CDC® CYBER 200 FORTRAN LANGUAGE 1.5

FOR USE WITH
CDC® CYBER 200
OPERATING SYSTEM 1.5

REFERENCE MANUAL

CONTROL DATA

60457040

CDC® CYBER 200 FORTRAN LANGUAGE 1.5

FOR USE WITH
CDC® CYBER 200
OPERATING SYSTEM 1.5

REFERENCE MANUAL

CONTROL DATA

60457040

REVISION RECORD	
REVISION	**DESCRIPTION**
A	Original release.
(7-2-79)	
B	This revision documents the CDC CYBER 200 FORTRAN language at release 1.5.
(8-22-80)	
C	This revision documents the CDC CYBER 200 FORTRAN language at release 1.5.1.
(11-15-80)	
Publication No. 60457040	

REVISION LETTERS I, O, Q AND X ARE NOT USED

Address comments concerning this manual to:

CONTROL DATA CORPORATION
Publications and Graphics Division
215 MOFFETT PARK DRIVE
SUNNYVALE, CALIFORNIA 94086

or use Comment Sheet in the back of this manual

LIST OF EFFECTIVE PAGES

New features, as well as changes, deletions, and additions to information in this manual are indicated by bars in the margins or by a dot near the page number if the entire page is affected. A bar by the page number indicates pagination rather than content has changed.

Page	Revision
Front Cover	-
Title Page	-
ii	C
iii/iv	C
v/vi	C
vii thru ix	B
x	C
xi	B
1-1 thru 1-3	B
2-1 thru 2-7	B
3-1 thru 3-4	B
4-1	B
4-2	B
5-1 thru 5-6	B
6-1 thru 6-7	B
7-1 thru 7-8	B
8-1 thru 8-5	B
9-1 thru 9-7	B
10-1 thru 10-3	B
11-1	C
11-2	B
11-3	B
11-4	C
11-5 thru 11-12	B
12-1	B
12-2	B
13-1 thru 13-17	C
14-1 thru 14-25	B
15-1 thru 15-12	C
16-1 thru 16-7	B
A-1	C
A-2	C
B-1 thru B-33	B
C-1 thru C-3	B
D-1 thru D-16	B
E-1 thru E-3	B
F-1	B
F-2	B
G-1 thru G-3	B
Index-1 thru Index-5	C
Comment Sheet	C
Mailer	-
Back Cover	-

PREFACE

This manual describes the CONTROL DATA® CYBER 200 FORTRAN programming language for use under control of the CDC® CYBER 200 Operating System running on the CDC® CYBER 200 Series computer system.

The reader of this manual should be familiar with the FORTRAN language and the CYBER 200 Series computer system. Familiarity with vector processing concepts is highly desirable.

CYBER 200 FORTRAN is designed to comply with American National Standards Institute FORTRAN language, as described in X3.9-1966. Control Data extensions to the standard FORTRAN language are indicated by shading. Example programs or parts of programs are shaded if they contain lines using extensions to the ANSI standard.

Related information of interest can be found in the listed publications.

Publication	Publication Number
CDC CYBER 200 Assembler Version 3 Reference Manual	60457050
CDC CYBER 203 Computer Hardware Reference Manual	60256010
CDC CYBER 205 Computer Hardware Reference Manual	60256000
CDC CYBER 200 Operating System Version 1 Reference Manual, Volume 1 of 2	60457000
CDC CYBER 200 Operating System Version 1 Reference Manual, Volume 2 of 2	60457010

CDC manuals can be ordered from Control Data Corporation, Literature and Distribution Services, 308 North Dale Street, St. Paul, Minnesota 55103.

This product is intended for use only as described in this document. Control Data cannot be responsible for the proper functioning of undescribed features or parameters.

CONTENTS

APPENDIXES

INDEX

FIGURES

TABLES

NOTATIONS

Certain notations are used throughout this manual that have consistent meanings. The notations are:

UPPERCASE	Uppercase letters in language forms indicate actual keywords.
lowercase	Lowercase letters in language forms indicate user-supplied character strings.
#	Numbers preceded by the pound sign are hexadecimal numbers.
numbers	All numbers in this manual are decimal unless preceded by a pound sign or otherwise denoted as hexadecimal numbers.
Δ	Delta represents a blank.
Shading	Shading indicates features that are Control Data extensions to the standard FORTRAN language. The parts of example programs that use language extensions are also shaded.

INTRODUCTION 1

The FORTRAN programming language for the CDC® CYBER 200 computer contains both CDC and unique CYBER 200 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 CDC® CYBER 200 FORTRAN extensions to standard FORTRAN allow the FORTRAN user to exploit the vector processing capabilities of the CYBER 200 Series computers. In CYBER 200 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

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 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 CYBER 200 FORTRAN compiler, the program is changed into a form that can be loaded and executed by the CYBER 200 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 until it 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 CYBER 200 System Input/Output (SIO). 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 CYBER 200 FORTRAN program is provided in figure 1-1.

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 lines 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 from 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	CYBER 200 FORTRAN statement, with blank characters ignored except in character and Hollerith constants, can appear anywhere within this field.
73 thru end of source time	Identification field, the contents of which are always ignored by the FORTRAN compiler, can contain any characters.

END LINES

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.

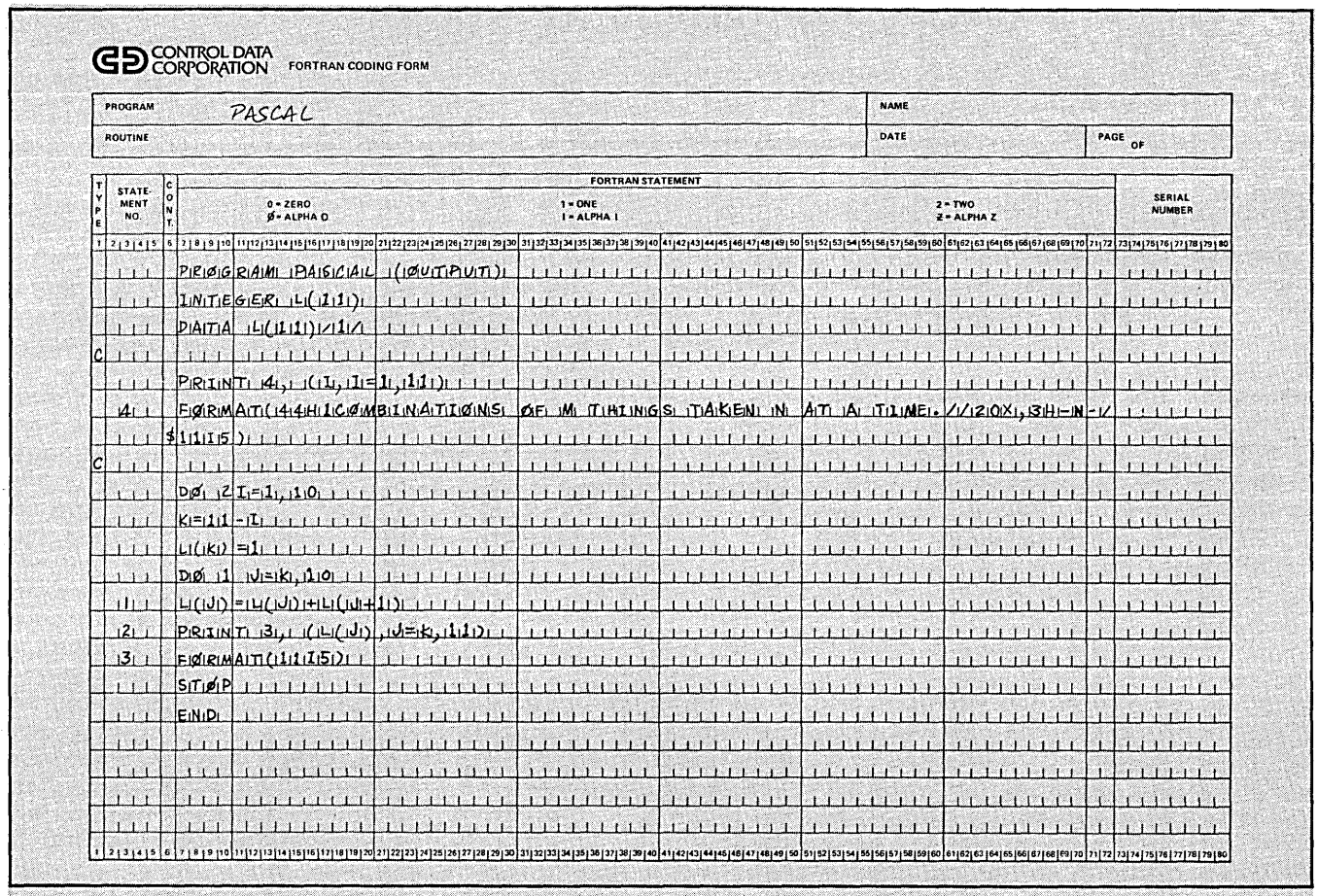
Form:

```
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.


CONTROL DATA CORPORATION FORTRAN CODING FORM

PROGRAM PASCAL NAME

ROUTINE DATE PAGE OF

TYPE	STATEMENT NO.	CONT.	FORTRAN STATEMENT (0 = ZERO Ø = ALPHA O; 1 = ONE I = ALPHA I; 2 = TWO Z = ALPHA Z)	SERIAL NUMBER
			PROGRAM PASCAL (OUTPUT)	
			INTEGER L(111)	
			DATA L(111)/111/	
C				
			PRINT 4, (I, I=1,111)	
	4		FORMAT(14H1COMBINATIONS OF M THINGS TAKEN N AT A TIME.//20X,3H-N-/	
		$	1115)	
C				
			DO 2 I=1,10	
			K=111-I	
			L(K)=1	
			DO 1 J=K,110	
	1		L(J)=L(J)+L(J+1)	
	2		PRINT 3, (L(J), J=K,111)	
	3		FORMAT(111(I5))	
			STOP	
			END	

Figure 1-1. Sample Coded FORTRAN Program

STATEMENTS

The statements in the CYBER 200 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.

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 be output.

Nonexecutable statements describe characteristics, arrangement, and format of data, as well as entry point 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.

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)

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.

Ordering of statements

Figure 1-2 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. Comment lines can appear anywhere within the program before the END line, except before statement continuation lines.

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 could 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 can 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.

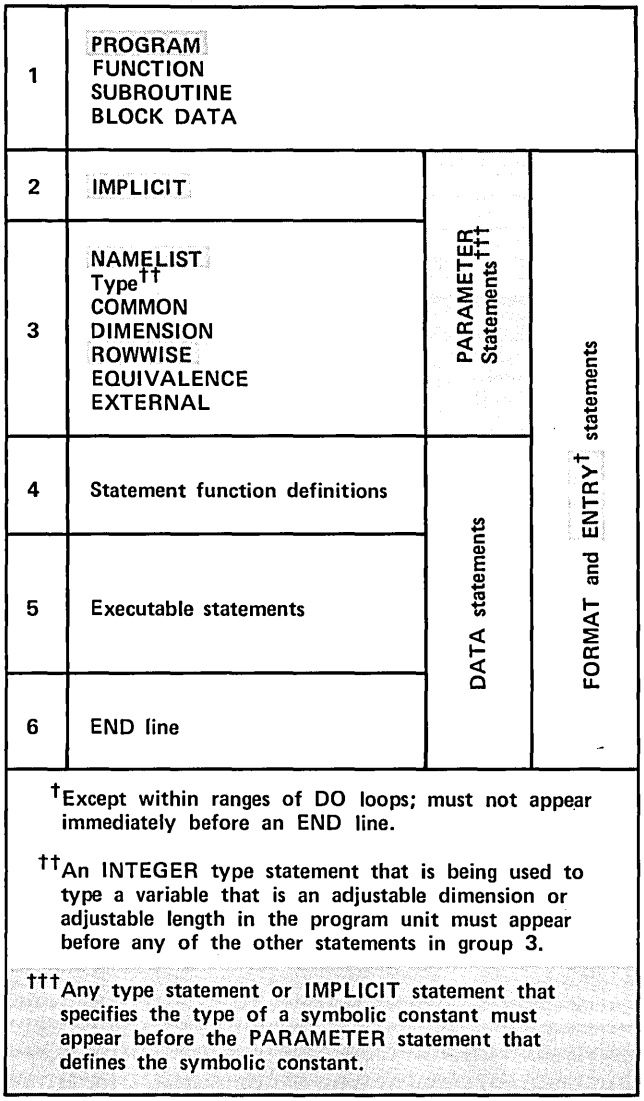


Figure 1-2. Ordering of Statements

The elements of a syntactically correct CYBER 200 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. CYBER 200 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, CYBER 200 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	Δ Blank or space = 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 : 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 Δ is used in this manual to denote a blank character that is not optional.

DATA ELEMENTS

Data can be represented in a CYBER 200 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 constant has one of nine data types:

- Integer
- Real
- Double-precision
- Complex
- Logical
- Hollerith
- Character
- Hexadecimal
- Bit

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 fullword the number occupies in memory has the 64-bit binary representation 0...010000100101.

SYMBOLIC CONSTANTS

A symbolic constant is a name that has a constant value. The value is specified in a PARAMETER statement. The type of a symbolic constant is specified implicitly by the first letter of the name, or explicitly by a type statement: the legal types for symbolic constants are integer, real, double-precision, complex, logical, character, and bit.

A symbolic constant can be used like any other constant except:

- A symbolic constant cannot appear as part of another constant. For example, if X is a real symbolic constant, (0.,X) is not a complex constant.
- A symbolic constant cannot appear in a FORMAT statement.

- A symbolic constant cannot appear as input data.
- A symbolic constant cannot appear in a PROGRAM statement.

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 or a descriptor. Descriptors are discussed in the vector programming section.

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
- 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.

An array can be a simple array or a 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:

$d_1,\ldots,d_n$

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 is 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 must be in common. A variable used in this way as an adjustable dimension must either be implicitly integer, or 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 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 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 through 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.

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 lowercase 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.

TABLE 2-2. ARRAY ELEMENT SUCCESSION FORMULAS

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.

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)

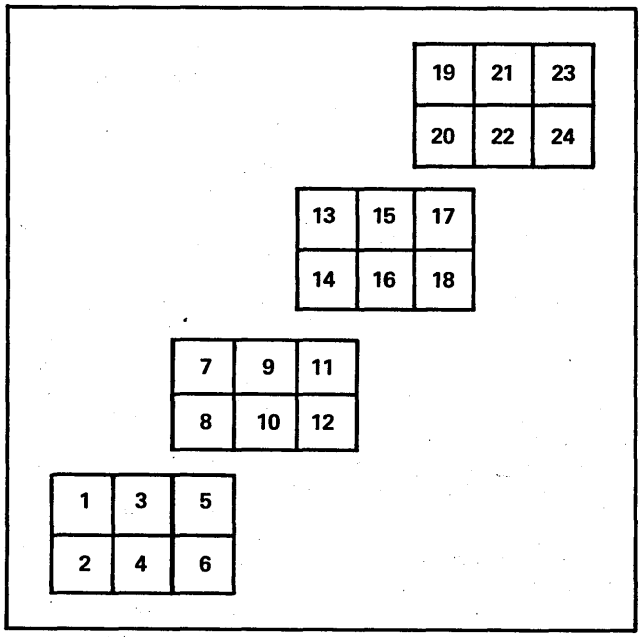


Figure 2-1. Conventional Ordering of Elements in a 3-Dimensional Array, A(2,3,4)

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 declared.

The data type of a variable, array, or function name is implied by the first letter of the name or it 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.

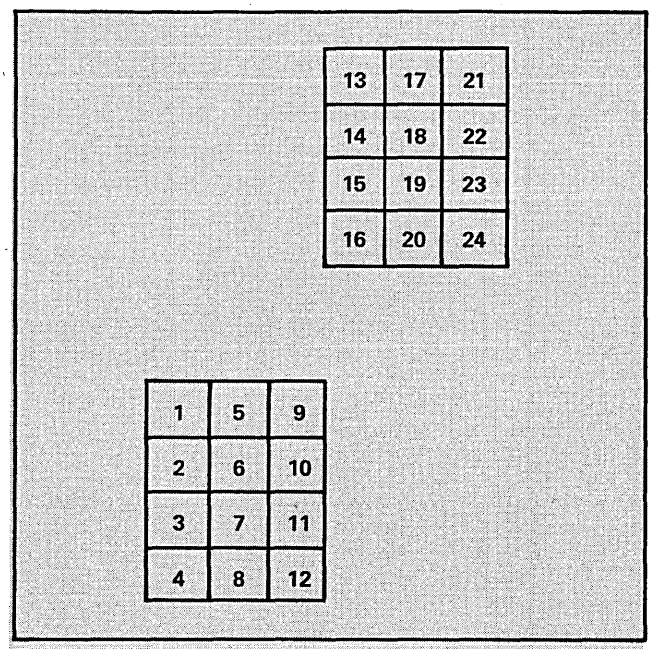


Figure 2-2. ROWWISE-Declared Array, A(2,3,4)

INTEGER ELEMENTS

An integer constant has the following form:

$d_1 d_2 \ldots d_m$

d_i A decimal digit (0 through 9); m is greater than or equal to 1 and less than or equal to 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 as shown in figure 2-3.

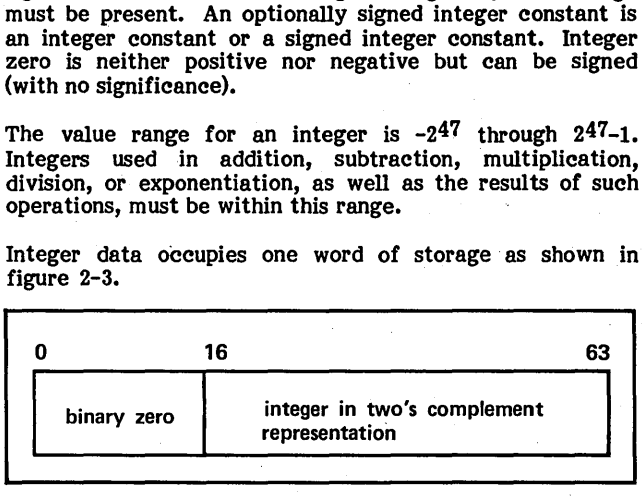


Figure 2-3. Integer Data Representation

A variable or array can be associated with the integer data type implicitly or explicitly, as described under Variables in this section.

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 .95370811543187E+8645. The smallest positive real number that can be represented is approximately .51921128456573E-8617. The precision retained in calculations involving real numbers is approximately 14 significant decimal digits.

Real data occupies one word of storage as shown in figure 2-4.

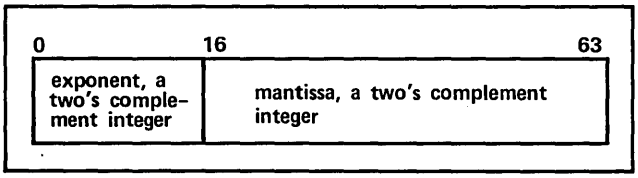


Figure 2-4. Real Data Representation

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 appropriate hardware reference manual contains more detailed descriptions of the hardware representations for numeric data.

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 .56119459376694461996204140 73D+8645. The smallest positive double-precision number that can be represented is approximately .51921128456573305570041353 39D-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).

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

3.14159265358979323846264 33D+0
```

Examples of signed double-precision constants:

```
+.25D+1      -.25D1      +2500D-3
```

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.

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 as shown in figure 2-5.

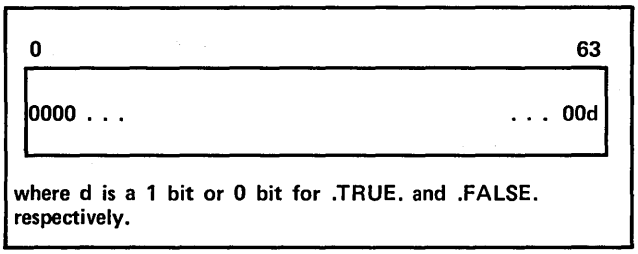


Figure 2-5. Logical Data Representation

A variable or array is associated with the logical data type by means of the IMPLICIT or the LOGICAL type declaration statement.

HOLLERITH ELEMENTS

A Hollerith constant is a string composed of an (unsigned) integer constant followed by the letter H or R, 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:

mRs

mHs

m An (unsigned) integer constant less than or equal to 255 and nonzero.

R Right-justified with binary zero fill.

H Left-justified with blank fill.

s A string of exactly m characters included in the 64-character ASCII subset (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 affect how Hollerith data is stored; however, when used as an actual argument in a subroutine call or function reference, a constant is aligned on a fullword boundary and extended with blanks on the right so that it occupies a whole number of words.

R constants are limited to one word and are right-justified with binary zero fill.

Examples of Hollerith constants:

19HRESULT NUMBER THREE 5H12345

5HΔΔΔΔΔ1H,
5R12345

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 apostrophes. If an apostrophe (') is required within the string as one of the characters, it must be prefixed with another apostrophe. The character blank is a significant character in a character constant.

Form:

'$c_1c_2 \ldots c_m$'

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. Character data is not left-justified and blank-filled as is Hollerith data.

Examples of character constants:

'RESULT NUMBER THREE' '12345' 'ΔΔΔΔΔ' ','

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 apostrophes. The 16 hexadecimal digits are the digits 0 through 9 and the letters A through F.

Form:

$X'h_1h_2 \ldots h_m'$

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 alignment 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.

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 apostrophes.

Form:

$B'b_1b_2 \ldots b_m'$

b_i A bit (0 or 1); m is less than or equal to 255.

Bit data uses m contiguous bits; the alignment is not significant. The digits 0 and 1 each correspond to one bit in storage.

Examples of bit constants:

B'0' B'10101111' B'000000000000001'

Bit constants are restricted to use in subprogram references, 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.

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 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.
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 evauation 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 are. In this situation, the execution of F might 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)
- Double-precision
- Real
- Integer

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 parenthesis in the same expression. Furthermore, if X and Y are arithmetic expressions, 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 bases and exponents 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

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, 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 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-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:

- A character 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:

```
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 comparison 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.

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

The periods are part of the operators and must appear. 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

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
```

.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:

```
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 can 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, is evaluated as if it were written (X.OR.((Y.AND.Z).OR.W)). If the user had intended (X.OR.Y).AND.(Z.OR.W), the parentheses must 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.

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	

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 side and right 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.

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.

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.

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.

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.

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

- Real part

  Real part of a complex value.

- Imaginary part

  Imaginary part of a complex value.

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.

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 on the right with blanks until it matches the length of var; then expr is assigned to var. If var is shorter than expr, expr is truncated on the right until it matches the length of var; then expr is assigned to var.

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 blanks.

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
  I=1
  LOG2 = I .EQ. 0
  ```

  LOG2 is assigned the value .FALSE. because I does not equal 0.

- ```
  LOGICAL NSUM, VAR
  BIG = 200.
  VAR = .TRUE.
  NSUM = BIG .GT. 200. .XOR. VAR
  ```

  NSUM is assigned the value .TRUE.

- ```
  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)
  ```
  ```

The statements of a CYBER 200 FORTRAN program are in effect executed consecutively except when flow is altered by a flow control statement or by an exceptional condition (for example, end-of-file on input, or a data flag branch interrupt). The execution of a flow control statement alters, interrupts, terminates, or otherwise modifies the normal sequential flow of program execution.

Some flow 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.

Besides the CONTINUE statement, CYBER 200 FORTRAN contains four types of flow 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 type 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.

When the GO TO is executed, control is transferred such 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 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; in the second form of the statement, 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.

```
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 $L \leq 0$, control falls through to the statement immediately following the GO TO statement, in this case CAT = FUR + GRIN.

IF STATEMENT

The IF statements provide for transfer of control or for conditional execution of one or more statements.

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 depending on 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, logical IF statement, block IF statement, ELSE statement, ELSE IF statement, or END IF statement.

Upon execution of this statement, the logical expression expr is evaluated. 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. comparisons 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 64-bit comparison to take place during evaluation of the expression.

BLOCK IF

The block IF statement has the following form:

IF (expr) THEN

expr — Any logical expression.

Upon execution of this statement, the logical expression expr is evaluated. If the value of expr is false, control transfers to an ELSE or ELSE IF statement; if neither an ELSE nor an ELSE IF statement is present, control transfers to an END IF statement. If the value of expr is true, execution continues with the next executable statement after the block IF statement.

ELSE

The ELSE statement has the following form:

ELSE

The ELSE statement can be used with a block IF statement to provide an alternate path of execution for a block IF statement. An ELSE statement can have a statement label, but the label cannot be referenced in any other statement.

ELSE IF

The ELSE IF statement has the following form:

ELSE IF (expr) THEN

expr Any logical expression.

The ELSE IF statement can be used with a block IF statement to provide an alternate path of execution for a block IF statement or another ELSE IF statement, and to perform a conditional test. An ELSE IF statement can have a statement label, but the label cannot be referenced in any other statement. The effect of execution of an ELSE IF statement is the same as for the block IF statement.

END IF

The END IF statement has the following form:

END IF

The END IF statement terminates a block IF structure. Each block IF statement must have a corresponding END IF statement.

BLOCK IF STRUCTURES

Block IF structures provide for alternate execution of blocks of statements. A block IF structure begins with a block IF statement and ends with an END IF statement; it can contain an ELSE statement or one or more ELSE IF statements. Each IF, ELSE, or ELSE IF statement can be followed by a block of executable statements called an if-block.

An if-block can contain any number of executable statements; it can also contain no statements. Control can transfer out of an if-block from within the if-block; however, control cannot transfer into an if-block from outside the if-block.

A simple block IF structure is shown in figure 5-1. If the expression in the block IF statement is true, execution continues with the first statement in the if-block. If the expression is false, control transfers to the statement following the END IF statement.

```
IF (expr) THEN
    if-block
END IF
```

Figure 5-1. Simple Block IF Structure

A block IF structure that contains an ELSE statement is shown in figure 5-2. If the expression in the block IF statement is true, execution continues with the first executable statement in if-block-1. If a statement in if-block-1 does not transfer control elsewhere, control transfers to the statement following the END IF statement after execution of if-block-1.

```
IF (expr) THEN
    if-block-1
ELSE
    if-block-2
END IF
```

Figure 5-2. Block IF Structure With ELSE Statement

If the expression in the block IF statement is false, control transfers to the first statement in if-block-2. If a statement in if-block-2 does not transfer control elsewhere, control transfers to the statement following the END IF statement after execution of if-block-2.

A block IF statement can have no more than one associated ELSE statement.

A block IF structure that contains ELSE IF statements is shown in figure 5-3. If the expression in the block IF statement is true, execution continues with the first executable statement in if-block-1. If a statement in if-block-1 does not transfer control elsewhere, control transfers to the statement following the END IF statement after execution of if-block-1.

```
IF (expr) THEN
    if-block-1
ELSE IF (expr) THEN
    if-block-2
ELSE IF (expr) THEN
    if-block-3
END IF
```

Figure 5-3. Block IF Structure With ELSE IF Statements

If the expression in the block IF statement is false, control transfers to the first ELSE IF statement that is associated with the block IF statement. The expression in this ELSE IF statement is evaluated. If the expression is true, execution continues with the first executable statement in if-block-2. If a statement in if-block-2 does not transfer control elsewhere, control transfers to the statement following the END IF statement after execution of if-block-2.

If the expression in the first ELSE IF statement is false, control transfers to the second ELSE IF statement that is associated with the block IF statement. The expression in the second ELSE IF statement is evaluated in the same manner as in the first ELSE IF statement. Any number of ELSE IF statements can appear in a block IF structure.

An ELSE statement can also appear in this structure; however, it must follow the last ELSE IF statement. The if-block associated with the ELSE statement is executed if all of the logical expressions in the block IF statement and ELSE IF statements are false.

NESTING BLOCK IF STRUCTURES

A nested block IF structure is a block IF structure that appears in an if-block of another block IF structure. A nested block IF structure must appear entirely within an if-block. Control can transfer from an if-block of a nested block IF structure to the if-block of the outer block IF structure in which the nested block IF structure appears. Control cannot transfer from an if-block of an outer block IF structure to an if-block of a nested block IF structure, however. Nested block IF structures are shown in figure 5-4.

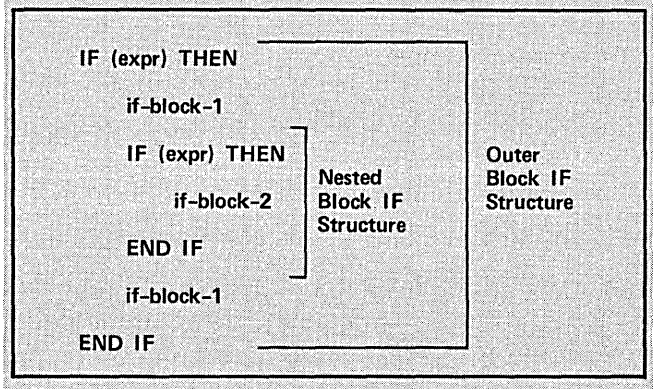


Figure 5-4. Nested Block IF Structure

A block IF structure can appear within the range of a DO loop, but the entire block IF structure must appear in the DO loop range. An END IF statement cannot be the terminal statement of a DO loop. A DO loop can appear in an if-block, but the entire range of the DO loop must appear in the if-block.

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 flow 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 block IF, ELSE, ELSE IF, or END IF statement
- A special call statement
- A DO statement
- A READ statement containing an ERR or END branch
- A CALL statement that passes a return label
- An arithmetic IF statement
- 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 5-5 are incorrect. The GO TO statement in figure 5-5 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 5-5. Incorrect: Entering Range of DO Before DO Execution

Execution of a DO statement causes the following sequence of operations:

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 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 allowed 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 5-6 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 5-6. DO Control Variable Reinitialization

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 5-7 gives an example of an incorrect transfer into the range of an inner DO. Since statement 500 in figure 5-7 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 5-7 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.) GO TO 500
      DO 500 K=1,10
      A(I,K)=SQRT(A(I,K))
  500 CONTINUE
```

Figure 5-7. Example of Incorrect Sharing of Terminal Statement

The range of a DO loop can contain a block IF structure, but the entire block IF structure must appear in the DO loop range. An END IF statement cannot be the terminal statement of a DO loop. A DO loop can appear in an if-block, but the entire range of the DO loop must appear in the if-block.

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.

PAUSE STATEMENT

The PAUSE statement has the following form:

PAUSE n

n Optional. A string of one to five decimal digits, or a character constant.

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

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 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, System Input/Output (SIO) module, System Request Language (SRL) module, assembly language 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, . . . ,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, 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.

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 5-8 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, 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, 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.

```
      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 5-8. Example of RETURN Statement

Specification statements are nonexecutable statements that 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 might not be required.

The nonexecutable data initialization statement is also described in this section.

TYPE STATEMENTS

Each variable, array, and function name that appears in a CYBER 200 FORTRAN program must be associated with a data type. Explicit type statements and implicit typing are the two ways to make this association.

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. 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.)

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. IMPLICIT statements must precede all other specification statements.

Form:

IMPLICIT $typ_1(list_1), \ldots, typ_m(list_m)$

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:

$v_1, v_2, \ldots, v_n$

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.

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 $v_1/d_1/, \ldots, v_n/d_n/$

REAL $v_1/d_1/, \ldots, v_n/d_n/$

DOUBLE PRECISION $v_1/d_1/, \ldots, v_n/d_n/$

COMPLEX $v_1/d_1/, \ldots, v_n/d_n/$

LOGICAL $v_1/d_1/, \ldots, v_n/d_n/$

CHARACTER*K $v_1*k_1/d_1/, \ldots, v_n*k_n/d_n/$

BIT $v_1/d_1/, \ldots, v_n/d_n/$

v_i A variable, array, array declarator, or function name.

d_i Optional. Represents the initial value for v_i. If omitted, the surrounding slashes must also be omitted. (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. If *K is omitted, a length of one byte is implied for every v not accompanied by a *k.

$*k_i$ 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 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 doubleword 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 $/blk_1/list_1/blk_2/list_2\ldots/blk_n/list$

n

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, 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.

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 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 fullword boundaries. Character and bit types can appear in a common block with other types, as long as the elements having the other types are not forced off fullword 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, 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 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, as 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 CYBER 200 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 by using an array element name containing a subscript that has 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 section 2. However, if neither of these forms is used, 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 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 fullword 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, as 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 part A of figure 6-1, 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 part B of figure 6-1. 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 to cause, directly or indirectly, 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.

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.

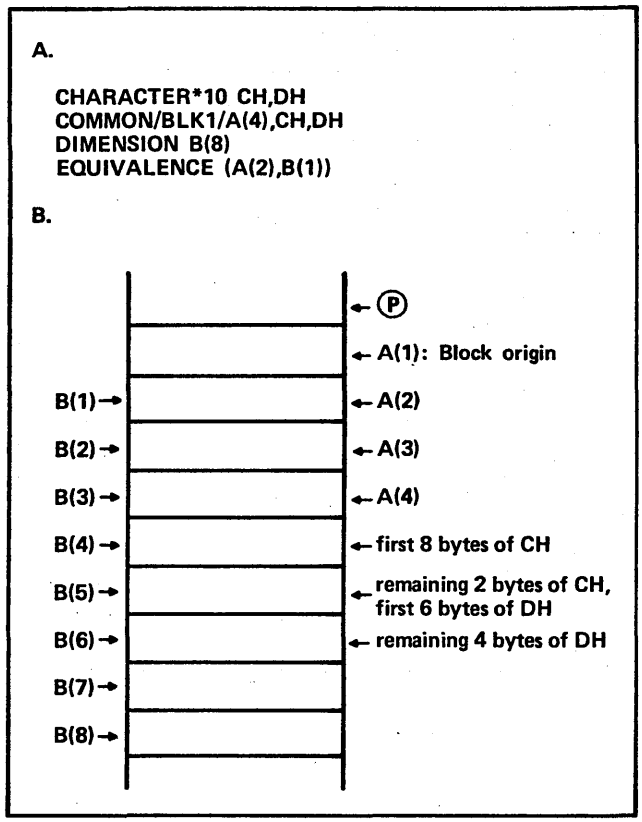


Figure 6-1. COMMON and EQUIVALENCE Statements

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.

TABLE 6-1. EXTERNAL DECLARATION OF A SUPPLIED FUNCTION

Type of Function	Declaration	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

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 can 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.

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/
```

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:

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

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.
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.

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 (H type) 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, 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.

If the variable is complex or double-precision, the H constant fills only the real part or first word of the variable. The imaginary part or second word is padded with zeros. Therefore, only one H constant is needed to initialize a complex or double-precision variable.

R constants used to initialize variables are padded with zeros on the left, or truncated on the right to fit in the variable. If the variable is complex or double-precision, the R constant fills only the real part or first word of the variable. The imaginary part or second word is filled with zeros. Therefore, only one R constant is needed to initialize a complex or double-precision variable.

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 compiler issues fatal diagnostic 374, TOO LITTLE DATA IN HEX OR BIT CONSTANT. If the constant or constant list bit length is too long, the compiler issues fatal diagnostic 375, 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 truncated on the left to fit the length of the variable, depending on whether the number of bits in the constant is 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'/
```

All of these DATA statements except the last one 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 fatal diagnostic 375, 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) can be initialized 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.

PARAMETER STATEMENT

The PARAMETER statement is a nonexecutable statement that specifies a value for a symbolic constant.

Form:

PARAMETER($name_1$=$value_1$,. . .,$name_n$=$value_n$)

$name_1$ The name of a symbolic constant.

$value_1$ A constant expression.

The constant expressions may contain previously declared symbolic constants. The operators that can be used in the expressions are +, -, *, and /.

The type of $value_i$ must be compatible with the type of $name_i$. Type conversion is performed for assignment statements. The type of a symbolic constant can be integer, real, double-precision, complex, logical, character, or bit. The type is determined by the first character of the name, as for variables, or by an explicit type statement. If the name appears in a type statement, the type statement must precede the PARAMETER statement. The length of a character symbolic constant is determined by the type, but the length of a bit symbolic constant is determined by the bit value in the PARAMETER statement.

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.

DEFINING PROGRAM UNITS AND FUNCTIONS STATEMENT

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 discussed in section 14. CALL and RETURN are discussed in section 5. 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 can be written. If a number of calculations are required to obtain several values, a subroutine subprogram can 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.

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 can 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)

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 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 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 Input/Output

Files can be specified in the PROGRAM statement by providing four parameters enclosed in brackets following the TAPE or UNIT specification. If the parameters are not specified, the default parameters are used.

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 a disk file with a default buffer size of three small pages.

Parameters must be supplied at the first reference within the PROGRAM statement and are not allowed for subsequent references to the same file. If TAPE7 is to be a 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 parameters given with TAPE7 are ignored and TAPE7 becomes a 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.

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

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 — 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.

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 section 4).

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 outside 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 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.

TABLE 7-1. DISTINGUISHING FUNCTIONS AND SUBROUTINES

	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.

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), 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, 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.

TABLE 7-2. CORRESPONDENCE OF ACTUAL TO DUMMY ARGUMENTS

Dummy Argument	Actual Argument
Simple variable	Scalar expression
Descriptor	Descriptor Descriptor array element Vector
Simple array	Simple array Array element (simple)
Descriptor array	Descriptor array 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 must contain at least one 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, . . . ,a_n)

CHARACTER FUNCTION f*m (a_1,a_2, . . . ,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 it can 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, 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 $s(a_1,a_2, \ldots ,a_n)$

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.

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.

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 CYBER 200 FORTRAN specification subprogram that can consist of only the following statements:

- BLOCK DATA statement
- IMPLICIT statements
- Explicit type statements
- EQUIVALENCE statements
- DIMENSION statements
- ROWWISE statements
- COMMON statements
- DESCRIPTOR statements
- DATA statements
- PARAMETER statements
- END statement

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

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.

Form:

ENTRY e $(a_1,a_2, \ldots ,a_n)$

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.

Control passes to the first executable statement following the ENTRY statement when the entry point name e is used in a CALL statement or function reference. In a subroutine subprogram, the entry point name e 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 e 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 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, CYBER 200 FORTRAN permits scalar function subprograms to have vector function entry points, and vector functions (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.

In a character function, the length of the entry point must be specified in a CHARACTER type statement and cannot be specified in the ENTRY 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.) 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

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.

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.

```
      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
      GO TO 300
      END
```

Figure 7-3. Multiple Entry Subroutine

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.

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. A function name that is not followed by an argument list is not a function reference.

```
      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) GO TO 300
      FRED = (C+A)/B
      RETURN
      END
```

Figure 7-4. Multiple Entry Function

The data processed by a CYBER 200 FORTRAN program can be constants in the program, or variables and arrays initialized with DATA statements, or can include variables and arrays whose values are read from input units at program execution time. When the program has produced results, CYBER 200 FORTRAN output statements can be used to send the results to specified output units. Input and output can be performed as frequently as necessary during the execution of a program.

The following types of input and output (I/O) statements are available in CYBER 200 FORTRAN:

Sequential	READ, WRITE, PRINT, PUNCH, ENCODE, and DECODE statements, with optional data format specifications
Buffer	BUFFER IN and BUFFER OUT statements (appendix G)
Namelist	READ, WRITE, PRINT, and PUNCH statements with a namelist group name that implies an I/O list and data format specifications
Concurrent	Q7BUFIN, Q7BUFOUT subroutine calls (section 13)

The legal record types for the types of input and output are given in table 8-1.

TABLE 8-1. LEGAL RECORD TYPES

Input/Output Statement Type	Record Type			
	Control Word	Undefined	Fixed Length	Record Mark
Formatted	Yes	No	Yes	Yes
Unformatted	Yes	No	Yes	No
BUFFER IN	Yes	No	Yes	No
BUFFER OUT	Yes	No	Yes	No
Q7BUFIN/OUT	Yes†	Yes†	Yes†	Yes†

†When using Q7BUFIN or Q7BUFOUT, any record type can be read or written, but the file is always treated as if the record type is undefined.

In addition to sequential, buffer, and namelist input/output, the unit positioning statements REWIND, BACKSPACE, and ENDFILE and the memory-to-memory data conversion statements ENCODE and DECODE are discussed in this section. Data conversion on input and output (the FORMAT statement) and the input and output lists for input/output statements are discussed in section 9.

All files or units referred to in an input/output statement, except for the standard output file OUTPUT, must be declared in a PROGRAM statement at the beginning of the main program. The default record length on ASCII card files is 80 characters. Record length on any ASCII file should not exceed 137 user-supplied characters. Record length can be changed with the Q8WIDTH subroutine described in section 13. The first character of a print file record is always used as carriage control and is not printed; the second character appears in the first print position (carriage control characters are listed in section 9). Additional requirements for input files and for the form of output files produced through FORTRAN are discussed in section 15.

Data moved by using input/output statements is always in a block that begins on a small page boundary and that has a length that is a multiple of small pages.

The following parameters are specified in input/output statement forms throughout this section to indicate the three basic components of input/output statements:

u	Logical unit number having an integer value of from 1 to 99 and associated with a particular file by means of the PROGRAM statement (see section 7).
fmt	Format designator; the statement label (having a value of 1 to 99999) of a FORMAT statement in the progam unit containing the input/output statement, or the name of an array containing the format specification.
iolist	List of variables and arrays to be input from or output to u according to fmt.

SEQUENTIAL INPUT STATEMENTS

To request that data be transferred into main memory, a READ statement is used. The formatted READ statement must be used for ASCII input, whereas the unformatted READ statement can be used to read data that does not require conversion from an external to the internal representation. The READ statement with implied device is a formatted read from the file INPUT.

FORMATTED READ STATEMENT

A formatted READ statement has the following form:

READ(u,fmt,END=m,ERR=n)iolist

END=m	Optional. End-of-file transfer parameter; m is a statement label in the same program unit.
ERR=n	Optional. Data transfer error parameter; n is a statement label in the same program unit.
iolist	Optional input list.

The END and ERR parameters can be in either order when both are present.

Execution of the formatted READ statement causes transfer of one or more records from the specified file u to the memory locations associated with the names in iolist, according to the format specified by fmt. The number of records transferred depends upon fmt and iolist. Conversion from the external to the internal form for the data takes place in accordance with the formatting.

Transfer on End-of-File

If a READ statement is executing when the next sequential record of input data is an end-of-file indicator, the variables in the input list become undefined, the end-of-file record becomes the preceding record, execution of the READ is abandoned, and control transfers to the statement label m specified by the END option in the READ statement. When no END option has been specified and an end-of-file is encountered on input, run-time error 13 is issued.

Data Transfer Errors

If a READ statement is executing when a data transfer error occurs, the variables in the input list become undefined, the record in error becomes the preceding record, execution of the READ is abandoned, and control transfers to the statement label n specified by the ERR option in the READ statement. If a data transfer error occurs on input when no ERR option has been specified, an appropriate diagnostic is issued.

A data transfer error is an abnormal condition such as a hardware parity error or reading of a tape record into too small a buffer area.

READ WITH IMPLIED DEVICE

Data can be transferred into memory without explicitly identifying the data's source.

Form:

READ fmt,iolist

iolist Optional input list.

Execution of this READ statement causes transfer of one or more records from the file INPUT to memory locations named in iolist, according to the format specified by fmt.

UNFORMATTED READ STATEMENT

An unformatted READ statement has the following form:

READ(u,END=m,ERR=n)iolist

m,n Optional statement labels, same as for formatted READ; when either is omitted, the entire parameter must be omitted.

iolist Optional input list.

Execution of the unformatted READ statement causes transfer of a single record of binary data from the specified file u to the memory locations associated with the names in iolist. In contrast to the formatted READ, no format designator is present in the statement and no data conversion takes place.

The size of the record read from the file u must match iolist exactly.

SEQUENTIAL OUTPUT STATEMENTS

To request that data be moved out of main memory, a WRITE, PRINT, or PUNCH statement is used. The formatted WRITE statement must be used to write ASCII output whereas the unformatted WRITE statement can be used to write binary data without converting it to ASCII. The PRINT statement is a formatted WRITE statement with the file OUTPUT implied. The PUNCH statement is a formatted WRITE statement with the file PUNCH implied; the file can be punched after program termination.

FORMATTED WRITE

A formatted WRITE statement has the following form:

WRITE(u,fmt)iolist

iolist Optional output list.

Execution of the formatted WRITE statement causes transfer of one or more records from the memory locations named in iolist to the specified file u according to the format specified by fmt. Hollerith data in fmt is also transmitted.

PRINT

Output data can be transferred to the file OUTPUT without explicitly indicating the file. This is done with the PRINT statement.

Form:

PRINT fmt,iolist

iolist Optional output list.

The statement causes transfer of one or more records, including any Hollerith data in the format specification fmt, from the memory locations associated with the names in iolist to OUTPUT according to fmt.

PUNCH

If PUNCH has been declared in the PROGRAM statement, data can be written on the file PUNCH with the PUNCH statement.

Form:

PUNCH fmt,iolist

iolist Optional output list.

The PUNCH statement causes transfer of data from the memory locations named in iolist, and Hollerith data in the associated FORMAT statement, to PUNCH, according to the format specified by fmt. PUNCH records are limited to 80 characters. After program execution is complete, PUNCH is a file that is suitable for punching (PUNCH is not punched automatically).

UNFORMATTED WRITE

An unformatted WRITE statement has the following form:

WRITE(u)iolist

iolist Output list; required.

Execution of the unformatted WRITE statement causes transfer of a single record, consisting of the sequence of values specified by iolist, to the file u. No data conversion takes place. If data is written by an unformatted WRITE and subsequently read by an unformatted READ, exactly what was written is read.

MEMORY-TO-MEMORY TRANSFER

The ENCODE and DECODE statements are used to reformat data in memory by transferring the data under format specification from one area of memory to another. The ENCODE statement is similar to a formatted WRITE statement and the DECODE statement is similar to a formatted READ statement. However, unlike a WRITE or READ statement, the source (for decoding) or destination (for encoding) of the data is a variable or array rather than an input or output file. Data is transferred internally with an ENCODE or DECODE statement; no files are involved.

ENCODE STATEMENT

An ENCODE statement has the following form:

ENCODE(cl,fmt,b)iolist

cl Length in number of 8-bit bytes of each encoded record.

fmt Label of a FORMAT statement in the same program unit, or the name of an array containing the format specification.

b Simple variable, array element, or array name that serves as the starting location of the encoded records.

iolist Optional output list.

Execution of an ENCODE statement causes the creation of one or more records, each having a length of cl characters. When iolist is present, the values of the elements in the list are written into memory, starting with b and according to the format conversion specified by fmt. The length of each record must be less than or equal to cl characters; if the record produced is shorter, it is extended on the right to cl with blanks.

The records created are stored adjacently and in the order of their creation, the first record beginning at b. At the inception of the encoding, the variable or array b must neither appear in the input/output list nor be associated through EQUIVALENCE statements or COMMON with any element of the input/output list. Furthermore, if fmt identifies an array, it must not be associated with b.

DECODE STATEMENT

A DECODE statement has the following form:

DECODE(cl,fmt,b)iolist

cl Length in number of 8-bit bytes of each of the records to be decoded.

fmt Label of a FORMAT statement in the same program unit, or the name of an array containing the format specification.

b Simple variable, array element, or array name that serves as the starting location of the area in memory from which the values decoded into iolist elements are taken.

iolist Optional input list.

Execution of a DECODE statement causes the reading of one or more records, starting at b, into the items in iolist, according to the format conversion specified by fmt. This action must not require more than cl characters of any record; however, if fewer than cl characters are required, the remaining characters are ignored. The list iolist must not include any elements of type bit and must not include descriptor names.

The records scanned by the execution of the DECODE statement partition the memory area, starting at b, into groups of cl characters. At the inception of the decoding, no element in the list can be associated with the variable or array b.

An example using ENCODE and DECODE is given in figure 8-1. LOC is an integer array having six elements. The call to the concurrent input/output routine Q7BUFIN transfers one small page of unformatted data into memory starting at LOC(1) and proceeding through the next 511 words. Q7WAIT is called to check the status of the buffering operation initiated by the Q7BUFIN call, and control branches to the statement labeled 666 if the operation has not completed normally or if not enough data was read in by the operation. The DECODE statement, under control of FORMAT statement 1, places the last four bytes of LOC(6) left-justified (A conversion) into TEMP, without change of form. The ENCODE statement, under control of FORMAT statement 2, packs the first four bytes of LOC(1) and the first four bytes of TEMP into NAME.

```
      DIMENSION LOC(6)
          .
          .
          .
      CALL Q7BUFIN (60,LOC,1)
          .
          .
          .
      CALL Q7WAIT (60, LOC, STATUS, 1, PCOUNT)
      IF (PCOUNT.LT.1 .OR. STATUS.NE.0)
     $    THEN GO TO 666
          .
          .
          .
      DECODE (8,1,LOC(6)) TEMP
      ENCODE (8,2,NAME) LOC(1), TEMP
          .
          .
          .
    1 FORMAT (4X,A4)
    2 FORMAT (2A4)
  666 CALL ERRMESS
```

Figure 8-1. Example Using ENCODE and DECODE Statements

Unless the conversion specified is F, E, or I, the conversion code in the FORMAT statement (fmt) determines the format of the encoded record or, for decoding, of the value assigned to the corresponding input/output list item. If F, E, or I conversion is specified, the type of the corresponding input/output list (iolist) item determines the format of the encoded record or of the value decoded into the iolist item. In figure 8-1, the FORMAT statements determine the formats of the values placed into TEMP and NAME, while the types of LOC and TEMP are not significant in the encoding and decoding operations.

NAMELIST INPUT AND OUTPUT

Formatted input and output of a group of variables and arrays having a single identifying name can be accomplished without using a format specification and without using an input/output list. Before the group is named in an input or output statement, a NAMELIST statement in the nonexecutable portion of the program unit must declare the group name and the group elements.

Form:

NAMELIST/g_1/v_1 . . . ,v_n . . . /g_p/w_1, . . . ,w_m

g_i Group's symbolic name, which must be unique within the program.

v_i,w_i Simple variables (excluding bit variables) and array names (excluding bit arrays and arrays having variable dimensions in the program unit).

The variables and arrays v_i and w_i, which cannot be of type bit, can belong to one or more namelist groups. If the statement occurs in a function subprogram, the variables can include the function name and any of the function secondary entry point names. The namelist group name g_i identifies the succeeding list of variables and arrays up to the next slash. The group name can be declared only once, and it cannot be used in the program unit for any purpose other than a namelist name.

The forms of namelist input and output statements are as follows:

READ(u,g_i,END=m,ERR=n)

READ g_i

WRITE(u,g_i)

PRINT g_i

PUNCH g_i

The optional END and ERR parameters are defined and used exactly as described for formatted and unformatted READ statements.

When an input or output statement references a namelist group name, data (in the formats described below) is transferred between memory and the specified file. If the specified group name is not found before the end of the file, execution continues at the statement labeled n if END=n was specified; otherwise, a run-time diagnostic is issued.

NAMELIST INPUT DATA

Data read by a namelist READ statement must contain only names listed in the referenced namelist group, but need not contain all of the names nor names in the order given in the defining NAMELIST statement. The values of the variables and arrays not included in the input data remain unchanged.

Form:

&gΔd_1,d_2, . . . ,d_n&END

g Group name, as defined in a NAMELIST statement.

d_i Data item having one of the following forms:

simple variable=constant
array name=constant, . . . ,constant
array name (integer constant
subscript expressions)=constant, . . . , constant

where a constant can be preceded by a repetition factor (an integer constant) followed by an asterisk.

The constants must be of the same type as the variable or array to the left of the equals sign. The number of constants in any item must be less than or equal to the number of elements in the array. Blanks must not appear between the ampersand and the group name, nor within names, constants, or the &END. However, a blank (indicated here by Δ) must appear between the group name and the first data item, and may appear where not forbidden. When a constant is logical, it can assume any of the following forms:

T	F
.T.	.F.
.TRUE.	.FALSE.
TRUE	FALSE

The first character of every record must be a blank. A data item can extend over more than one record, but this must be done in such a way that none of the variables, constants, array names, constants along with repeat specification, or the &END extends over more than one record. In this context, a complex constant is considered as two constants enclosed in parentheses and separated by a comma. Character elements are an exception in that they can extend over the record boundary.

On execution of an input statement that references a namelist group, the value to the right of each equals sign in the input record is assigned to the element to the left. When the left side is an array element name, the constants in the list on the right side are assigned to array elements in the array, starting with the element specified and continuing in the order in which the elements occur in memory (see section 2). An array name to the left of an equals sign names the first element of the array. The elements in the namelist group identified by g which have not been redefined when the &END is encountered are neither redefined nor undefined by the execution of the READ statement.

NAMELIST OUTPUT DATA

The entire sequence of records output by a namelist WRITE statement is in a form that is suitable to be input by a namelist input statement. The names and values of all variables and arrays in the group are output to OUTPUT, PUNCH, or the designated file.

Upon execution of a namelist WRITE or PRINT statement, a sequence of records with a character blank in the first character position of each record is transmitted. The remaining character positions are as follows:

First record: &g

where g is the namelist group name

One or more records:
$v_1=cl_1,v_2=cl_2,\ldots,v_n=cl_n$

where v_i is either a variable or array name, and cl_i is a constant or a list of constants separated by commas

Last record: &END

When v_i is an array name, the number of constants in the list cl_i is the number of elements in the array; the order of these constants is the order in which the elements occur in memory.

For character elements, output is in the form of character constants. The form of all other constants is as if the elements had been written with the format Ew.d (see section 9) where w and d are sufficient to preserve the precision of the elements. For complex elements, the format of the constant is as if the format specification ('(',Ew.d,',',Ew.d,')') had been used (section 9).

UNIT POSITIONING

REWIND, BACKSPACE, and ENDFILE statements can be used to adjust the current reading or writing position of a file when input/output is being performed by buffer, namelist, and sequential input/output statements. Files contain one or more records grouped as a totally ordered set. The initial position of a file is at the beginning of the file's first record. The end-of-file indicator, when present, follows the last record of information. The forms for the unit positioning statements all refer to u, which is an integer constant or simple integer variable specifying a logical unit number.

REWIND

Execution of a REWIND statement causes the file specified by u to be positioned at its initial point, the beginning-of-information, even when several ENDFILE statements were issued since the last REWIND.

Form:

REWIND u

If u is already at the beginning-of-information, the statement has no effect.

BACKSPACE

Execution of a BACKSPACE statement results in the positioning of the file in such a way that what had been the preceding record prior to the statement execution now becomes the next record.

Form:

BACKSPACE u

If u is at its initial point, BACKSPACE has no effect.

ENDFILE

Execution of the ENDFILE statement causes an end-of-file indicator to be written as the last record on the file.

Form:

ENDFILE u

Note that if a file is created by a FORTRAN run-time routine and an ENDFILE statement is executed first, the file is considered to be an SIO record mark file. Any attempt to perform unformatted or buffer input/output on the file results in a fatal error.

Input/output requests are initiated by using the statements discussed in the previous section. This section covers the details of specifying the format of the data being transmitted with these input and output statements. The interaction between the FORMAT statement and an input or output list, data conversion specification, and execution-time data formatting are covered here.

INPUT/OUTPUT LISTS

The list portion of an input or output statement specifies the items that are to be read or written and indicates the order of transmission. The items in the list are read or written sequentially. Although the list can contain any number of items separated by commas, the number of items should be compatible with the FORMAT statement specifications and (on input) with the amount of input data available.

An output list must accompany an unformatted WRITE. Otherwise, an input/output statement need not contain an input/output list. When no list is present on input, a record is skipped (unless the corresponding FORMAT statement contains an H specification). Only Hollerith information that appears in the format specification can be written when no list is present on output.

When an input/output list is present, the execution of an input or output statement continues as long as any list item remains to be processed. If there is insufficient data on input to give every element of the list a value, a run-time error results; whenever there is excess data in the input record, the excess is ignored.

LIST ITEMS

An input or output list item can be any of the following:

- Simple variable name
- Array element name
- Array name
- Implied DO list
- Descriptor name prefixed with an ampersand

Descriptors and type bit data elements cannot be input, nor can they be output by means of any unformatted output statement. They must not appear in input lists; they can appear in output lists only of formatted output statements.

All elements of an array can be specified with an unsubscripted array name list item. An unsubscripted array name output list item causes the elements of the array to be output in the order in which the element values are stored in memory, irrespective of the fact that the array has been specified to be a conventional array or rowwise array. Parts of arrays can be specified by means of an implied DO list item. Subscripts in an input/output list can be any valid subscript described for array element names in section 2.

Examples:

```
READ (2,100)A,B,C,D

READ (3,200)A,B,C(I),D(3,4),E(I,J,7),H

READ (4,101)J,A(J),I,B(I,J)

READ (2,202)DELTA
```

On input or output, the list is scanned and each variable in the list is paired with the field specification provided by the FORMAT statement. After one item has been input or output, the next format specification is taken together with the next element of the list, and so on until the end of the list. The correspondence between data in an input record and the format specification is shown in figure 9-1. In the figure, 100 is read into the variable L under the specification I3; 22 is read into M under the specification I2; 3456712 is read into N under the specification I7; and 1, 10, 11, and 0 are read into the four elements of the array K under the specification I2 (element K(1,1) = 1, K(2,1) = 10, K(1,2) = 11, and K(2,2) = 0).

Array declaration:

```
DIMENSION K(2,2)
```

Input statements:

```
      READ (5,20) L,M,N,K
20    FORMAT (I3,I2,I7,4I2)
```

Input record:

100	22	3456712	01	10	11	00

Figure 9-1. Example of Inputting Formatted Data

Attempting to read more data than is in the input stream produces an error, unless the END parameter (described in section 8) is used to test for the end of the file.

IMPLIED DO IN INPUT/OUTPUT LIST

Input and output of array elements can be accomplished by using an implied DO in the input/output list. A list of variables followed by a DO loop control variable is enclosed in parentheses to form a single implied DO list element.

For example,

```
READ (5,100)(A(I),I=1,3)
```

has the same effect as the statement

```
READ (5,100)A(1),A(2),A(3)
```

The general form for an implied-DO item is:

(list,i=m_1,m_2,m_3)

list — An input/output list in which i and implied DO items can appear. Variables, array elements, and subscripted or unsubscripted array names can appear more than once in list.

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 variable must not be used twice as a control variable within the set of parentheses that defines an implied DO item. When processing begins on the implied DO input/output list item, the control variables are set to their initial values; list is transmitted; then i is incremented by m_3 and if the value of i does not exceed m_2, list is again transmitted. The looping process repeats until the value of i exceeds m_2.

The control variable can appear in list. If it does appear in list (for example, in subscript expressions in the list, or as a list element), it assumes whatever value it has currently as the control variable.

Examples:

Implied DO	Transmitted
(A,B,C,I=1,4)	A,B,C,A,B,C,A,B,C,A,B,C
(I,R(I+1,1),C,I=1,3)	1,R(2,1),C,2,R(3,1),C,3,R(4,1),C
((R(I,J),I=1,2),J=1,2)	R(1,1),R(2,1),R(1,2),R(2,2)
(J,(R(I,J),I=1,2),Y(J), (B(K),K=1,2),J=1,3)	1,R(1,1),R(2,1),Y(1),B(1),B(2),2, R(1,2),R(2,2),Y(2),B(1),B(2),3, R(1,3),R(2,3),Y(3),B(1),B(2)

FORMAT STATEMENT

Data being moved, as a result of execution of an ENCODE/DECODE or a formatted input/output statement, changes in format during transferral. Data that has not been formatted consists of a string of the binary values that are in memory. Data that has been formatted consists of ASCII characters.

The nonexecutable FORMAT statement or a variable format specification is required to specify the formatting of data being moved with ENCODE/DECODE and formatted input/output statements between a file and main memory. Unlike most other nonexecutable statements, a FORMAT statement can appear anywhere in the program unit in which it is referenced. More than one input/output statement can reference a single FORMAT statement.

Form:

sn FORMAT (q_1 $s_1r_1f_1$ x_1...x_{n-1} $s_nr_nf_nq_2$)

sn — A statement label that must appear; the label referred to in an input/output statement in the same program unit.

q_1,q_2 — Optional; one or more slashes indicating record boundaries.

s_i — Optional scale factor for E, F, G, and D conversion codes.

r_i — Optional unsigned integer constant that serves as a repeat count for the field specification; when omitted, a count of 1 is assumed.

f_i — A field specification indicating one of fourteen types of data conversion and editing codes; or, a list of the form:

($q_1s_1r_1f_1x_1$...$x_{n-1}s_nr_nf_nq_2$)

with the restriction that parentheses can be nested to only three levels in a FORMAT statement.

x_i — A comma, or one or more slashes indicating record boundaries; optional following an H specification, ' specification, or X specification.

The format specification is enclosed in parentheses. No more than two additional levels of parentheses can be nested within the outermost set of parentheses. Blanks within the format specification are not significant except in Hollerith (nH) and apostrophe field specifications.

Example:

```
      READ (5,100) INK, NAME, ARE
  100 FORMAT (10X,I4,I2,F7.2)
```

Generally, each item in the input/output list of an input/output statement should correspond to a single field specification in the specified FORMAT statement. However, complex variables always correspond to two F, E, G, or Z field specifications. Also, arrays in an input/output list must correspond to as many field specifications as there are array elements.

The FORMAT statement usually specifies the type of conversion that is performed for input data without any regard for the type of the variable that subsequently receives the value. Nevertheless, the data type of the variable in the input/output list should match that of the field specification, because no conversion takes place upon assignment to (or transmittal from) an input/output list element except with G, F, E, and I data conversion codes.

Example:

```
      INTEGER N
      READ (5,100)N
  100 FORMAT (F10.2)
```

In the above example, a real number is read, converted to integer, and assigned to the variable N.

Repetition of individual field specifications (except for H, ', X, and T editing specifications) or of groups of specifications (delimited by parentheses) is indicated through use of the repeat count r. If the input/output list warrants it, the same conversion is repeated the number of times specified by r. When a group is to be repeated, an integer constant precedes the left parenthesis to indicate how many times the group is to be repeated; a repeat count of r has an effect identical to concatenating r copies of the field specification string that composes the group. The repeat count r cannot exceed 255.

FORMAT CONTROL

Execution of a formatted input/output or ENCODE/DECODE statement initiates format control. Format control depends on information provided jointly by an element of the input/output list and an element in the format specification. The format specification, like the input/output list, is interpreted from left to right (except for the effects of repeat counts). The field specifications in a FORMAT statement determine how many characters are read from input or written on output.

When more field specifications than input/output list elements are given, some field specifications are not used. When fewer field specifications than input/output list elements are given, a new record starts and control moves to the group repeat specification of the group terminated by the last preceding right parenthesis. If no group exists, control returns to the first left parenthesis of the format specification. This action has no effect on the scale factor described later in this section.

On initiation of a formatted read, format control is also initiated and one record is read. Thereafter, additional records are read only as the format specification demands. Such action must not require more characters than a record can contain, which is 137 characters at the maximum. A slash in the format specification demands that a new record start. Any unread characters remaining in the current input record when a slash is encountered in the format specification are ignored.

When a formatted WRITE is executed, records are built and output according to the interaction of format specification and the output list. A slash in the format specification demands that building of the current record terminate, the record be transmitted, and a new record be started. Termination of format control also causes the current record to be written.

When all elements of the input/output list have been processed, the format control terminates.

DATA CONVERSION

Ten types of data conversion codes are available, each of which causes conversion of ASCII data to a particular internal data format or vice versa. The editing codes H, ', X, and T are covered later in this section.

Forms:

Iw	Decimal integer conversion.
Ew.d	Single-precision floating-point conversion, with exponent.
Fw.d	Single-precision floating-point conversion, without exponent.
Gw.d	Single-precision floating-point conversion, with or without exponent; character conversion; logical conversion; integer conversion.
Dw.d	Double-precision floating-point conversion with exponent.
Lw	Logical conversion.
Aw	Character conversion.
Rw	Character conversion.
Zw	Hexadecimal conversion.
Bw	Bit conversion (on output only).
w	Field width in number of character positions in the external record, including any leading blanks, + or - signs, decimal point, and exponent; a nonzero (unsigned) integer constant.
d	Number of digits to the right of the decimal point in the field; an (unsigned) integer constant.

The field width w must be specified for all conversion codes. For the form w.d (except for G when associated with integer, logical, or character type items), the d must always be specified, even when it is zero. Also, the field width w must always be greater than d.

A type complex list item should correspond to two single-precision floating-point (real) field specifications: the first is for the real part and the second is for the imaginary part of the complex item.

CONVERSION SPECIFICATION

In numeric input conversions (F, E, D, G, and I), leading blanks in the input record are not significant, while other blanks are treated as zero characters. Plus signs can be omitted. With the F, E, G, and D input conversion codes, a decimal point in the input field overrides the decimal point specification supplied by the field specification.

The output field is right-justified for all output conversions. If the number of characters produced by the conversion is smaller than the field width w, leading blanks are inserted in the output field. For I format, if the number of characters produced by an output conversion is greater than the field width, the first character in the field is an asterisk, and the asterisk is followed by the rightmost w-1 characters that should have been output. For D, E, and F formats, if the number of characters produced by an output conversion exceeds the field width, the entire field is filled with asterisks. With output conversions, the external representation of a negative value is signed, while a positive value is not signed.

For F, E, and I conversions, the type of list element determines the internal representation, while the conversion code determines the external representation. For example, an external field read under F conversion into an integer list variable is converted to integer, whereas an integer list item written out under F conversion appears as a real value.

The types of input/output list items to which each of the conversion codes can correspond are shown in table 9-1. For example, the A conversion code can interpret input data as ASCII data which can then be assigned to a variable of any type except bit.

TABLE 9-1. INPUT/OUTPUT CONVERSIONS

Conversion Code	Data Type						
	Logical	Integer	Real	Double Precision	Complex	Character	Bit
F	n/a	x	x	x	x	n/a	n/a
E	n/a	x	x	x	x	n/a	n/a
D	n/a	n/a	n/a	x	n/a	n/a	n/a
G	x	x	x	x	x	x	n/a
I	n/a	x	x	x	x	n/a	n/a
A	x	x	x	x	x	x	n/a
R	x	x	x	x	x	x	n/a
L	x	n/a	n/a	n/a	n/a	n/a	n/a
Z	x	x	x	x	x	x	n/a
B	n/a	n/a	n/a	n/a	n/a	n/a	x

x indicates permitted conversions; n/a indicates type and conversion combination not allowed.

†Can also be used to output descriptors and double descriptors.

††Allowable conversion for output only.

I Conversion

The numeric field specification Iw indicates that the external field is to occupy w sequential character positions as an integer (including a possible plus or minus sign). On input, the value of the input list item corresponding to Iw appears internally as integer, real, or double-precision data, according to the type of the item in the input list. As input, the integer must be in the form of an optionally signed constant, except that embedded and trailing blanks are interpreted as zeros.

E and F Conversions

The numeric field specification Ew.d or Fw.d indicates that the respective external field is to occupy w sequential character positions (including any decimal point, exponent, or sign), where the part following the decimal point consists of d digits. On input, the value of the input list item corresponding to Ew.d or Fw.d appears internally as integer, real, or double-precision data, according to the type of the input list item. As input, a real number consists of an optional sign, followed by a string of digits that can contain a decimal point. This basic form can be suffixed with an exponent in any of the following forms:

- Signed integer constant
- The letter E suffixed with an optionally signed integer constant
- The letter D suffixed with an optionally signed integer constant

With the E and F conversion codes, an exponent that uses D is interpreted identically to an exponent using E and to an exponent that is expressed as a signed integer constant.

Output conversions specified by the E and F floating-point conversion codes differ. For output with F conversion, the real number produced consists of optional leading blanks, a minus sign if the internal value is negative, and a string of digits containing a decimal point. Together these represent the internal value modified by any established scale factor and rounded to d fractional digits.

For output with E conversion, the forms (for a scale factor of zero) are:

$b.a_1 \ldots a_d E\pm ee$	For values where the magnitude of the exponent is less than 100.
$b.a_1 \ldots a_d E\pm eee$	For values where the magnitude of the exponent is 100 to 999.
$b.a_1 \ldots a_d E\pm eeee$	For values where the magnitude of the exponent is greater than 999.

b is a minus sign if the number is negative, and blank if the number is positive.

$a_1 \ldots a_d$ are the d most significant digits of the value correctly rounded.

e is a digit of the decimal exponent.

A scale factor of n shifts the decimal point so that the fractional part of the number $(a_1 \ldots a_d)$ is multiplied by 10^n and the decimal exponent is reduced by n. If n is less than or equal to zero, there are exactly -n leading zeros with d+n significant digits after the decimal point. If n is greater than zero, there are exactly n significant digits to the left of the decimal point and d-n to the right of the decimal point.

G Conversion

The field specification Gw.d is used to input and output real, integer, character, and logical data. On output, it indicates that the external field is to occupy w sequential character positions; according to the type of the output list item-character, integer, or logical - Gw.d conversion is identical to an Aw, Iw, or Lw conversion code, respectively. On input, the value of the input list item corresponding to Gw.d appears internally as character, integer, logical, or real data, according to the type of the input list item.

For output of real data with a Gw.d conversion code, the method of representation in the external field is a function of the magnitude of the real data being converted. Let m be the magnitude of the internal data. The following tabulation exhibits a correspondence between m and the equivalent resulting method of conversion:

Magnitude of Data	Equivalent Conversion
$0.1 \leq m < 1$	F(w-4).d,4X
$1 \leq m < 10$	F(w-4).(d-1),4X
.	.
.	.
.	.
$10^{d-2} \leq m < 10^{d-1}$	F(w-4).1,4X
$10^{d-1} \leq m < 10^{d}$	F(w-4).0,4X
Otherwise	sEw.d, where s is a scale factor

In the tabulation, the effect of any scale factor is suspended unless the magnitude of the data is outside the range that permits effective use of the F conversion.

D Conversion

The numeric field specification Dw.d indicates that the external field is to occupy w sequential character positions, the fractional part of which consists of d digits. The value of the corresponding input/output list item appears internally as double-precision data.

As input, a double-precision number looks like a real number, only more digits can be retained during conversion than for the E, F, or G conversions.

For output, the representation of a double-precision number is the same as for E conversion, except that the character D, rather than E, is in the exponent.

L Conversion

The logical field specification Lw indicates that the external field occupies w positions that as a unit indicate truth or falsity. The value of the external field is stored on input as logical data.

As input, logical data consists of leading blanks, an optional decimal point, T (for true) or F (for false), and optional trailing characters.

For output, logical data consists of w-1 blank characters followed by the character T or F.

A and R Conversions

The character field specifications Aw and Rw indicate that the respective external fields occupy w sequential character positions in the external record. The value of the corresponding input/output list item appears internally as character data; the list item has already been explicitly or implicitly specified to have a length k.

When k equals w, the input field characters are assigned directly to, or transmitted from the input/output list item.

On input, if k is shorter than the number of characters in the input field (that is, w), only the rightmost k characters are assigned to the input list item; the leftmost w-k characters are ignored. If k is longer than w, w characters are left-justified in the list item with blank character fill to their right for the A conversion; and w characters are right-justified in the list item with binary zero fill to their left for R conversion.

On output, if k is shorter than the number of characters in the output field (that is w), the k characters of the output list item are output with w-k blank characters preceding. If k is longer than w, the leftmost w characters in the output list item are output for the A conversion; and the rightmost w characters in the list item are output for R conversion.

Z Conversion

The hexadecimal field specification Zw indicates that the external field occupies w positions, where each character position is a hexadecimal digit. The value of a corresponding input list item after assignment appears internally as hexadecimal data.

On input, w hexadecimal digits are transmitted to the associated list element, right-justified and binary zero-filled. Leading as well as embedded and trailing blanks in the input field are treated as zeros. If w is greater than the number of hexadecimal digits that can be represented in the list element, the rightmost part of the field is used and the input string is truncated on the left.

On output, the binary value of the corresponding output list item interpreted as hexadecimal digits is transmitted to the output field, right-justified and blank-filled. If w is less than the number of hexadecimal digits, the rightmost w digits are output.

Descriptors can be output, but not input, by using a PRINT, formatted WRITE, or ENCODE statement. A descriptor or descriptor array element name prefixed with an ampersand can appear in the output list, and must correspond to a Z specification in the appropriate FORMAT statement. Example:

```
      BIT B(3000)
      DIMENSION X(100)
      DESCRIPTOR D(3)
      .
      .
      .
      DO 100 I = 1,3
  100 ASSIGN D(I), X(1;100)
      .
      .
      .
      PRINT 50, (&D(I),I=1,3)
   50 FORMAT (1X,Z16)
```

An attempt to output descriptors by using any other type of conversion specification produces run-time diagnostics.

B Conversion

The bit field specification Bw indicates that the output field occupies w character positions, including w-1 blank characters followed by either a 0 or a 1. The value of the corresponding output list item, which must have been declared to be type bit, appears internally as a bit constant; output of bit arrays must be accomplished with a repeated B specification.

Bit data can only be printed or encoded with the B format. All other field specifications produce run-time diagnostics.

EDITING CODES

The edit field specifications, unlike the conversion codes, do not correspond to input/output list items. Instead, each interacts directly with the input or output record.

X Specification

The nX specification is used to skip characters or to generate blank characters. On input, n sequential characters of the input record are skipped. On output, n sequential blank characters are placed in the output record. The comma following X in the specification list is optional.

H and ' Specifications

The nH (or Hollerith) specification is used to output strings of characters and to read a character string into an existing H field specification within a FORMAT statement. The apostrophe specification, $'h_1 \ldots h_n'$, where h_i is a character, can be used as an alternate form of the H specification. For compatibility with FORTRAN Extended, the * can be used in place of the apostrophe. See appendix G. The comma following the Hollerith or ' specification in the specification list is optional.

On output, the n characters immediately following the nH specification (or the character string between the apostrophes) in the format specification are placed in the output record. Any characters in the ASCII subset (see appendix A), including blanks and apostrophes, are acceptable within the character string. However, when the apostrophe notation is used, any apostrophe character within the string must be immediately preceded with another apostrophe; only one of the apostrophes appears on the output record.

On input, the n characters immediately following the nH specification (or the character string between the apostrophes) in the format specification are replaced with n sequential characters from the input record. Any subsequent uses of the same format specification for output cause the n characters to be placed in the output record. An apostrophe specification which contains two consecutive apostrophes cannot be used on input. When a Hollerith or apostrophe specification is part of a format specification contained in an array, characters in the input stream are skipped, and no change is made to the Hollerith or apostrophe specification in the array.

T Specification

The Tp specification is used to indicate that the next external field is to begin at character position p in the external record, where character position 1 is the first in the record, position 2 is second in the record, and so on. Conversion, under format control, continues at character position p until another T specification is encountered or until processing begins on the next external record.

Character position p can be either greater than or less than the character position currently being processed, but it must not exceed the record length. The maximum value that p can ever correctly assume is 137.

On output, if the same character position is defined more than once, action on the latest definition takes effect. Because of the carriage control character, a character position p in the external record becomes character position p-1 in the print line.

SCALE FACTORS

The scale factor nP is used to change the position of a real number's decimal point when the number is input or output. Scale factors can precede D, E, F, and G specifications except when G is used on integer, logical, or character type items. The scale factor n, which is an optionally signed integer constant, affects conversion differently for each kind of conversion code.

A scale factor of zero is established when a reference is made to a FORMAT statement; it holds for all F, E, G, and D field specifications until another scale factor is encountered. Once a scale factor is specified, it remains in use for all D, E, F, and G specifications in that FORMAT statement until another scale factor is encountered. To nullify this effect for subsequent D, E, F, and G specifications in the statement, a zero scale factor 0P must precede a specification.

For F, E, G, and D input conversion with no exponent in the external field, and for F output conversions, the scale factor sets the externally represented number to the internal representation of the number, times 10 raised to the n^{th} power.

For F, E, G, and D input with an exponent in the external field, the scale factor has no effect.

For E and D output, the basic real constant part of the output quantity is multiplied by 10^n and the exponent is reduced by n.

For G output, the effect of the scale factor is suspended unless the magnitude of the data is outside the range that permits effective use of F conversion (see G Conversion in this section). If E conversion is required, the scale factor has the same effect as with E output.

PRINTER CARRIAGE CONTROL

When an output record is sent to a line printer, the first character of the record is used for carriage control and is not printed. For output directed to the card punch or any device other than the line printer, control characters are not required (the first character of a record is not treated as a carriage control character).

Although other characters might be available for a particular installation, the following values are standard for FORTRAN carriage control for line printers:

Character	Action Before Printing
Δ (blank)	Single line feed
0 (zero)	Double line feed
1 (one)	Feed to first line of next page
+	No line feed

Failure to specify a carriage control character can cause unexpected results because the first character of output data would be used as the carriage control character. Carriage control characters are required at the beginning of every record that is to be printed, including new records introduced by means of a slash. Carriage control characters can be generated by any means, such as an X or H editing specification.

EXECUTION-TIME FORMAT SPECIFICATION

No format statement is necessary for an input/output statement if an array has been created that contains the appropriate format specification. This array can be defined in any of the following ways:

- The format specification can be included in a DATA statement.
- The format specification can be included in a character assignment statement.
- The format specification can be created with the aid of ENCODE statements.
- At execution time, the format specification can be read in, under the A field specification, as ASCII data.

The format specification must be enclosed in parentheses, but it must not be preceded by the word FORMAT and a statement label. Once defined, the format specification array can be used by READ, WRITE, PRINT, PUNCH, ENCODE, or DECODE statements. More than one statement can reference the array.

The name of the array containing the specifications is used in place of the FORMAT statement number in the input or output statement. For example, assume the following format specification:

```
(E9.2,F8.2,I7,2E20.3,I4)
```

The array IVAR could be defined as follows by using a character assignment statement:

```
CHARACTER*24 IVAR(1)
IVAR(1) = '(E9.2,F8.2,I7,2E20.3,I4)'
```

A subsequent output statement in the same program could then refer to these format specifications as:

```
WRITE (2,IVAR) A,B,I,C,D,E,J
```

which produces exactly the same result as the statements:

```
      WRITE (2,10) A,B,I,C,D,E,J
10    FORMAT (E9.2,F8.2,I7,2E20.3,I4)
```

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 CYBER 200 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.

The array assignment statement is not a part of the explicit vector programming capability of CYBER 200 FORTRAN; it is a DO loop notation.

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 (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 m3 is omitted, the colon immediately preceding it must also be omitted and a value of 1 is assumed for the incrementation value.

* 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. 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. 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, 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 by using the subarray notation.

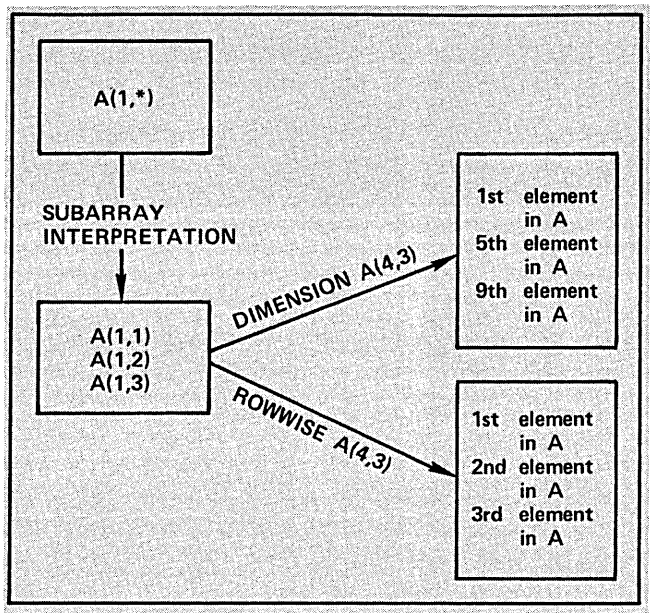


Figure 10-1. Meaning of a Subarray

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)
```

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.

Detailed in this section are the ways that a user can introduce machine vector instructions into the object code for a FORTRAN program. Any of the forms described here can be used in the same program with the previously described FORTRAN features.

AUTOMATIC VECTORIZATION

Automatic vectorization is a process by which the FORTRAN compiler translates an iterative, sequential procedure into parallel procedures. The aim of the process is to utilize the capabilities of the CYBER 200 hardware to produce optimal object code, without requiring alteration of FORTRAN programs that do not use the extensions of CYBER 200 FORTRAN, and without necessitating that a problem be reconceptualized in terms of parallel processes. Automatic vectorization of a FORTRAN program is selected by including the V compile option in the FORTRAN system control statement that requests compilation of the program.

Under the V option, CYBER 200 vector instructions are generated for DO loops that have certain characteristics. The object code generated for a loop that is accepted by the vectorizer consists of vector instructions rather than scalar instructions. If a loop is rejected by the vectorizer, the compiler attempts to transform the loop into a call to one of the supplied STACKLIB routines.

Automatic vectorization can be used with any FORTRAN program, including FORTRAN programs that do not use any of the extensions of CYBER 200 FORTRAN. However, because of the restrictiveness of the conditions for vectorization, summarized in table 11-1, it might not be possible for the vectorizer alone to achieve the degree of vectorization desired. As an alternative, the programmer can elect to use other methods, in conjunction with the V compile option or not, to specify vector operations explicitly.

GENERAL CHARACTERISTICS OF VECTORIZABLE DO LOOPS

A simple vectorizable DO loop is shown at (3) in figure 11-1. The range of a vectorizable loop can contain assignment statements, CONTINUE statements, and DO statements. An input/output statement or IF statement, for example, is not acceptable in a loop that is to be vectorized.

The initial, terminal, and incrementation parameters of the DO statement of a vectorizable loop must have certain characteristics. The incrementation parameter, if present, must be 1; an incrementation value of 2, for example, causes the loop not to be vectorized. Secondly, FORTRAN allows the parameters to be constants or variables; however, a variable initial, terminal, or incrementation parameter does prohibit the vectorization of any containing DO loop. For instance, the vectorizable loop defined at (3) has a variable terminal parameter. Loop (2) contains loop (3) and, consequently, cannot be vectorized. Thirdly, the iterative count of a loop or entire

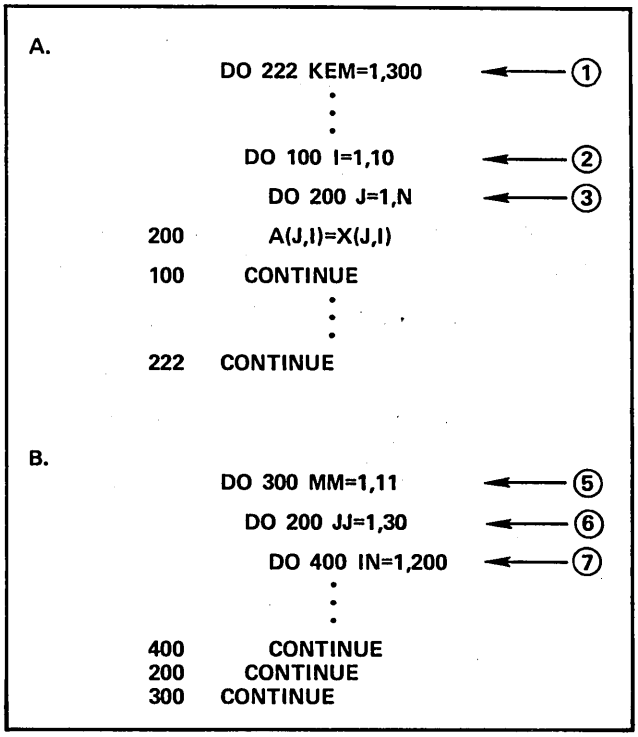


Figure 11-1. Form of Vectorizable DO Loops

nest of loops must be less than or equal to $2^{16}-1$ (that is, 65535). By this criterion, loops (7) and (6) in part B of figure 11-1 can be vectorized, depending on the range of the innermost loop; but loop (5) cannot be vectorized (because 30 * 200 * 11 = 66000).

When the initial or terminal parameter of a loop is a variable, the dimensions of the loop-dependent array references within the loop are used to determine the largest possible iterative count through which the loop can pass, and this count is used to decide if the loop can be vectorized.

The U compile option can be selected for unsafe vectorization. When U is selected, the compiler vectorizes loops that contain dummy arrays, even if the terminal value of the loop is variable. The optimization is considered unsafe because the presence of a variable dimension might cause the iterative loop count to exceed 65535.

The U compile option also enables vectorization of loops that contain an equivalenced data element on the left side of an assignment statement.

If a loop cannot be vectorized (loop (2) in figure 11-1, for instance), then a loop containing the nonvectorizable loop cannot be vectorized either. By this criterion, loop (1) is nonvectorizable.

TABLE 11-1. CRITERIA FOR VECTORIZABLE LOOPS

Can Appear in DO Loop	Must Not Appear in Any Part of DO Loop
Vectorizable loops nested within the loop.	Nonvectorizable loop nested in the loop.
Loop incrementation value of 1.	Loop incrementation value that is not 1 (this does not apply to the CYBER 200 Model 205).
Total iteration count less than 2^{16} for a nest of loops.	Total iteration count greater than or equal to 2^{16} for a nest of loops.
CONTINUE statement.	Any control statement besides DO and CONTINUE.
Arithmetic operators +, -, *, /, and **, logical operators.	Relational operators.
Real, integer, and logical data elements.	Any data element that has a type other than real, integer, or logical.
	Any input, output, or memory transfer statements.
References or calls to the following functions and subroutines: ABS, ACOS, ALOG, ALOG10, ASIN, ATAN, COS, EXP, FLOAT, IABS, IFIX, SIN, SQRT, and TAN.	References and calls to functions and subroutines other than ABS, ACOS, ALOG, ALOG10, ASIN, ATAN, COS, EXP, FLOAT, IABS, IFIX, SIN, SQRT, and TAN.
	Any data elements appearing on the left side of an assignment statement which have appeared in EQUIVALENCE statements.
Any scalar assignment statement whose right side is a real, integer, or logical expression.	Vector assignment statements.
Loop-dependent subscripts having one of the forms c, c+n, c-n, or c*n, where c is a control variable and n is an integer constant. The c*n form is not valid on the STAR 100 or the CYBER 200 Model 203.	Loop-dependent subscripts not of one of the forms c, c+n, c-n, or c*n, where c is a control variable and n is an integer constant. The c*n form is not valid on the STAR 100 or the CYBER 200 Model 203.
References to dummy arrays, so long as the terminal value of the loop is constant.	References to any dummy array when the terminal value of the loop is variable (can be vectorized if the U option is selected).
Loop-independent subscripts.	

ASSIGNMENT STATEMENTS IN VECTORIZABLE DO LOOPS

Operators in assignment statements in a vectorizable loop can be any of the arithmetic or logical operators. The use of relational operations within a loop causes the loop not to be vectorized.

The type of an operand occurring in the range of a vectorizable loop can be integer, real, or logical. A vectorizable loop containing a logical assignment statement is shown in figure 11-2.

```
      LOGICAL A, C, R
      DIMENSION A(50000), C(50000), R(49999)
      DO 999 X=2,50000
      R(X-1) = (A(X-1) .AND. A(X)) .OR. (C(X-1) .AND. C(X))
999   CONTINUE
```

Figure 11-2. Vectorizable Loop #1

References to dummy arrays cause a loop with a variable terminal value to be nonvectorizable, unless the U option is selected. A loop that is vectorizable only if the U option is selected is shown in figure 11-3.

```
      FUNCTION F(OFFS,F1,N)
      DIMENSION OFFS(N),F1(N)
      DO 3 I=1,N
    3 OFFS(I)=F1(I)+5.0
      F=OFFS(2)
      RETURN
      END
```

Figure 11-3. Vectorizable Loop #2 (U option)

Function and array references can appear in the range of a vectorizable loop. However, function references are restricted to references to the predefined functions ABS, IABS, FLOAT, IFIX, SQRT, EXP, ALOG, ALOG10, SIN, COS, TAN, ACOS, ASIN, and ATAN. References in a loop to other predefined functions, or to any user-defined function, yields a nonvectorizable loop. Loop-dependent array references are subject to several restrictions. Loop-independent array references are considered to be scalars in the context of automatic vectorization.

The left side of an assignment statement appearing in the range of a vectorizable DO loop must be a loop-dependent array reference or a scalar reference. A vector or descriptor on the left side makes the loop nonvectorizable. A loop-dependent array reference is an array reference with at least one loop-dependent subscript expression. For example, the left sides of the assignment statements in figures 11-1, 11-2, 11-3, and 11-4 are all loop-dependent array references.

```
      DIMENSION A(10,10), B(10,10)
      DO 10 I=1,10          <-------- (1)
        DO 20 J=1,10,2      <-------- (2)
        A(J,I) = B(J,J)
   20 CONTINUE
   10 CONTINUE
```

Figure 11-4. Vectorizable Loop #3

A scalar reference is defined to be a simple variable or a loop-independent array reference. Scalars appearing on the left side of an assignment statement are subject to certain restrictions in order for the containing loop to be vectorized. These restrictions are as follows:

- If a reference to the scalar appears before the first definition within the loop, the loop is not vectorizable.
- If the scalar is defined within the loop and appears in embedded loops, the loop is not vectorizable.
- If the scalar is an array element, every reference to that array in the range of the loop must have that same subscript.

Figure 11-5 shows an example of both an unvectorizable loop (1), and a vectorizable loop (2).

```
      DIMENSION A(10,10), B(10,10), C(10,10)
      DO 01 I=1,10
      T = A(I,1) + B(I,1)
      C(I,1) = C(I,1)*T + A(I,1)/T
      DO 2 J=1,10
      T = A(J,I) + B(J,I)
      C(J,I) = A(J,I)*T + B(J,I)/T
    2 CONTINUE
    1 CONTINUE
```

Figure 11-5. Vectorizable Loop #4

LOOP-DEPENDENT ARRAY REFERENCES IN VECTORIZABLE LOOPS

The form of a subscript expression in a loop-dependent array reference is restricted to the control variable of the loop or of a containing loop, or a control variable plus or minus a constant:

```
c
c+n
c-n

c   Control variable
n   Integer constant
```

Figures 11-1 and 11-2 contain subscript expressions that properly can appear in a vectorizable loop.

As the control variable passes through the range of values, the loop-dependent subscripts of array references must increase by a constant amount. If the array references are not contiguous, the loop should not contain embedded loops; otherwise it will not be vectorized.

In figure 11-4, the subscripts of array A are increasing in increments of 2, and the subscripts of array B are increasing in increments of 22. Therefore, loop 2 is vectorized, but loop 1 is not.

Although loop-dependent array references to a particular array can appear on both sides of assignment statements in the range of a DO loop, in certain cases this could inhibit vectorization of the loop. When an array reference appears on the right side of an assignment statement, elements of the array are being accessed. When an array reference appears on the left side, elements of the array are being defined. For any particular array, if the array portion that is referenced by the assignment statements in the range of the loop overlaps the portion that is defined, there is potentially a feedback situation which cannot be described in terms of vector operations. Because of the parallel nature of vector operations, vectors are not suitable for use in describing any iterative procedure containing feedback. The compiler is not always able to determine that a feedback situation does not exist. If there is a possibility of feedback, the containing loop will not be vectorized.

In the case where there is an overlap of referenced and defined array elements within the range of a loop, feedback occurs only if at least one of the array elements in the overlap is defined in an iteration of the loop and is then referenced during a subsequent iteration. Given the declaration and initialization statements:

```
DIMENSION A(5)
DATA A/1,2,3,4,5/
```

the following is an illustration of feedback. The program segment:

```
      DO 1 I=1,4
      A(I+1)=A(I)*2
    1 CONTINUE
```

consists of a DO loop in whose range lie a CONTINUE terminal statement and an assignment statement containing two loop-dependent array references. The array elements referenced and defined by successive iterations of the loop are as follows:

Referenced	Defined
A(1)	A(2)
A(2)	A(3)
A(3)	A(4)
A(4)	A(5)

Elements A(2), A(3), and A(4) constitute the overlap. On the first iteration of the loop, A(2) is defined to be 1. On the second iteration, A(2) is accessed (and is used to define A(3)). The result of completing execution of the DO loop is that the five elements of A have the respective values 1, 2, 4, 8, and 16. A vectorizer interpretation of the same loop would be to assign the $(i-1)^{th}$ element multiplied by 2 to the i^{th} element of A (where i ranges from 2 to 5), in which case the result would be the values 1, 2, 4, 6, and 8 respectively for the five elements of A. The loop is not vectorizable.

The program segment:

```
      DO 1 I=2,5
      A(I-1)=A(I)*2
    1 CONTINUE
```

is an example of a loop in which there is overlap but no feedback. The array elements referenced and defined by successive iterations of this loop are as follows:

Referenced	Defined
A(2)	A(1)
A(3)	A(2)
A(4)	A(3)
A(5)	A(4)

Again, the overlapping elements are A(2), A(3), and A(4). However, no element is defined on one iteration to be accessed on a successive iteration, as happened in the previous example. Therefore, the results of executing this DO loop would be identical to that of a vectorizer interpretation of the loop. The loop is vectorizable.

AUTOMATIC RECOGNITION OF STACKLIBABLE LOOPS

When the V option is selected, the vectorizer attempts to vectorize all DO loops as described under vectorization. However, when an innermost loop is rejected by the vectorizer (that is, the loop cannot be vectorized), the rejected loop is transformed into a call to a compiler-supplied STACKLIB routine or inline vector macro code if the loop is one of the following types:

```
      DO 1 I=L,M
    1 X(I) = X(I=1) + Y(I)
      DO 2 I=L,M
    2 X(I) = Y(I) + X(I-1)
      DO 3 I=L,M
    3 S = S+X(I)
      DO 4 I = L,M
    4 S = X(I) + S
      DO 5 I = L,M
    5 S = S+X(I)*Y(I)
      DO 6 I = L,M
    6 S = X(I)*Y(I) + S
      DO 7 I=L,M
    7 S = S+X(I)*X(I)
      DO 8 I = L,M
    8 S = X(I)*X(I) + S
      DO 9 I = L,M
    9 S = S+X(I)**2
      DO 10 I = L,M
   10 S = X(I)**2+S
```

In all of the above loops, X and Y represent distinct 1-dimensional arrays of type real which must not have appeared in an EQUIVALENCE statement. S represents a simple real variable. All of the above loops must contain only one assignment statement of the form described above. CONTINUE statements are allowed. The loop increment parameter can either be 1 or not explicitly specified in the DO statement. Variables L and M represent any DO loop initial and final value parameters. The variable I represents any DO loop control variable.

Loops 1 and 2 are converted to calls to a STACKLIB routine that performs addition recursively. Loops 3 and 4 are converted to calls to a STACKLIB routine that performs summation. Loops 5 through 10 are converted to calls to a STACKLIB routine that computes a dot product. If the object mainframe is the CYBER 200 Model 205, loops 2 through 10 are transformed into inline vector macro code.

AUTOMATIC VECTORIZATION MESSAGES

The vectorizer indicates on the source listing how many loops were encountered in the routine, how many loops were vectorized, and how many loops were transformed into calls to STACKLIB routines. For loops that could not be vectorized, a message is issued that indicates the first impediment to vectorization that was encountered by the compiler. The compiler analyzes a loop for vectorization from the bottom to the top; therefore, the diagnostic might not reflect the impediment to vectorization with the lowest source line number. See appendix B for a complete list of the vectorizer messages.

The source listing also indicates which loops were transformed into calls to STACKLIB routines. An example of a source listing for a program compiled with the V option is shown in figure 11-6.

EXPLICIT VECTORIZATION

Whether or not the V compile option has been selected, vector instructions are produced in the object code when vectors and special calls (section 13) are used. Vectors differ from the nonvector (scalar) elements of CYBER 200 FORTRAN in that many values (rather than a single value) are specified with the intention of using those values as a series of operands for a single operation, such as addition or assignment.

```
 00001            PROGRAM VECTRIZE
 00002            DIMENSION A(100),B(100)
 00003            DO 1 I=1,100
 00004    1       A(I) = A(I) + B(I)
 00005            DO 2 I=2,100
 00006    2       A(I) = A(I-1) + B(I)
 00007            STOP
 00008            END


UNCOLLAPSABLE LOOPS              REASON FOR NON-VECTORIZATION
------------------               ----------------------------
  LINE 00005                       LINE 00006   RHS ARRAY HAS POSSIBLY RECURSIVE PROPERTIES

STACKLIBED LOOPS
----------------
  LINE 00005

NUMBER OF LOOPS IN ROUTINE    = 0002

NUMBER OF VECTORIZABLE LOOPS = 0001

NUMBER OF STACKLIBED LOOPS    = 0001
      NO ERRORS
```

Figure 11-6. Vectorizer Output

The data elements used in vector programming are:

Vector	An ordered set of scalar elements; semicolon notation can be used to specify it.
Descriptor	A pointer to a vector.

Only the explicit forms for vectors and sparse vectors are unlike the forms of other FORTRAN data elements. The elements of a vector that is part of an array can also be treated as scalar array elements; the way that the data is to be processed (that is, as a vector or as an array) is determined by the notation used to reference it.

VECTORS

A CYBER 200 FORTRAN vector can be defined on a previously declared scalar array or on the dynamic space area and is delimited by a base address and length. A vector that is defined on an array is denoted by the name of the array, the location in the array of the vector's first element, and the number of elements encompassed by the vector.

Form:

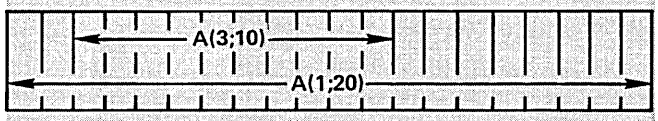
a(s;n)

a	The name of a real, integer, complex, double-precision, or bit array.
s	The subscript which designates the vector's first element (the base address); a subscript consisting of from one to seven subscript expressions, depending upon the number of dimensions in the array a.
n	The length, an integer expression (nonnegative). The length must not exceed 65535 elements (2^{16}-1) for integer, real, or bit vectors, and must not exceed 32767 elements (2^{15}-1) for complex or double-precision vectors.

The elements in the array a, starting with the element a(s) and continuing for n contiguous elements in memory, belong to the vector a(s;n). A semicolon must be used to separate the base address from the length. The user must ensure that the length of the vector is within the bounds of the array, as no compile-time or run-time check is made for this.

Example:

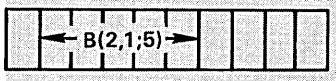
A(20) is the array declarator. Then the following are two of the possible vectors within the 20 elements of A:



Example:

If B(2,5) is the array declarator for a rowwise array, the following is a possible vector within the 10 elements of B:

The array elements within the vector, if referenced individually, would be B(2,1), B(2,2), B(2,3), B(2,4), and B(2,5). If B had been declared columnwise, the vector B(2,1;5) would have been the following:



and the array elements referenced individually would be B(2,1), B(1,2), B(2,2), B(1,3), and B(2,3).

An array element that is part of a vector can belong to more than one vector or can be used and modified individually as though it were not part of any vector. Vectors cannot be input or output by using the semicolon notation; a vector can be input or output only as the part of the array that embodies it. However, the semicolon notation for vectors can be used in the DATA statement to initialize vectors.

A vector that is a subset of the dynamic space area instead of an array is not named by using the semicolon notation described above, but can be indicated only by a descriptor that is assigned to point to it. To create such a vector, a descriptor ASSIGN statement must be used.

The type of a vector is the type of the array a. Vectors of type logical and character are not allowed, and the allowable occurrences of vectors of type complex and double-precision are highly restricted.

Vectors can be organized into arrays of vectors only in terms of descriptor arrays. A descriptor array element is a descriptor.

DESCRIPTORS

Conceptually, a descriptor is a pointer to a vector. A descriptor element in CYBER 200 FORTRAN introduces one level of indirection in referring to vectors. By means of the descriptor ASSIGN statement, a descriptor can be made to point to various vectors during program execution. Also, at compile time, a descriptor can be initialized to point to a particular vector by means of the DATA statement.

A descriptor is identified by a symbolic name. The descriptor must be specified explicitly or implicitly to have the same type as that of the vector to which it points. Descriptors can never be type double-precision.

Descriptors can occur individually in a program or they can be organized into arrays. In either case, every descriptor or descriptor array name must appear in a DESCRIPTOR specification statement. Also, before it is used, every descriptor must be defined by a descriptor ASSIGN statement or DATA statement.

Except in an output statement, an occurrence of a descriptor in the executable portion of a program refers to a vector. In an output list the name must be preceded by an & and refers to the descriptor itself. Descriptors can occur in all vector expressions, as well as in ASSIGN statements, DATA statements, argument lists, COMMON statements, and EQUIVALENCE statements.

At the machine level, a descriptor occupies a fullword as shown in figure 11-7.

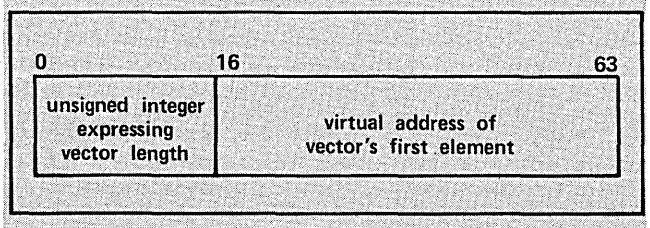


Figure 11-7. Descriptor Representation

The length is in words for integer, real, and complex descriptors. The length is in bits for bit descriptors.

Descriptors can be output using the FORMAT statement Z specification. Note that a descriptor must be prefixed with an ampersand when the descriptor appears in an input/output list.

EXPRESSIONS

A vector expression includes at least one of the following:

- Descriptor
- Descriptor array element
- Vector
- Vector function reference

Scalar data elements are also allowed in vector expressions. Operator precedences and the rules for resolving differences in operand data types for scalar and nonscalar expressions are identical.

This section gives the formation and evaluation rules for the four kinds of nonscalar expressions:

- Vector arithmetic

  Yields numeric-valued vectors; occurs in vector arithmetic and vector bit assignment statements.

- Vector relational

  Yields logical-valued vectors; occurs in vector bit assignment statements.

- Vector bit

  Yields bit-valued vectors; occurs in vector bit assignment statements.

VECTOR ARITHMETIC EXPRESSIONS

The discussion about scalar arithmetic expressions in section 3 also applies to vector arithmetic expressions, with the exception that an operand in a vector arithmetic expression can be a vector, descriptor, descriptor array element, vector function reference, constant, simple variable, scalar array element, or scalar function reference. Also, the use of vector complex and double precision entities is more restricted than is the use of scalar entities.

Evaluation of an expression that contains vector operands produces a vector result rather than a scalar result. The vector operands in an expression should always be the same length. Pairs of elements that are in corresponding positions in two vector operands (where correspondence is measured from the first element of the vector) are operated upon, and the result of the operation is placed in the corresponding position of a result vector that should have the same length as the vector operands.

Evaluation of a binary operation involving one vector operand and one scalar operand proceeds with the scalar operand being paired with each of the vector operand elements. Unary operations also apply to all of the elements of a vector operand.

For complex vector arithmetic expressions, the following restrictions apply:

- Operands can be integer, real, or complex. Double-precision operands are not allowed.
- Exponentiation is not allowed; the operators in a complex vector expression can be only +, -, *, and /.

For double-precision vector arithmetic expressions, the following restrictions apply:

- The expression must consist of either a double-precision vector or a reference to a FORTRAN-supplied double-precision vector function. No operators are allowed.
- The expression can appear only in a vector arithmetic assignment statement of type double-precision.

Given the declarations:

```
DESCRIPTOR D1, SCRP, RZLT
DIMENSION SCRP(3,3), VR(100), R(100)
DATA D1/VR(1;50)/, SCRP(3,1)/VR(1;100)/
```

the following are examples of vector arithmetic expressions:

- VR(1;100)

  Current values of the 100 consecutive elements in the array VR.

- D1

  Current values of the first 50 consecutive elements in the array VR.

- D1 + N

  A vector formed by adding the value of the scalar N to each element of VR(1;50).

- -(D1 + N)/2.**M

  A vector formed by adding N to D1, negating, then performing a divide by 2**M on each element of VR(1;50).

- SCRP(3,1)

  Current values of the 100 consecutive elements in the array VR.

- VEXP(R(10; 52); RZLT)

  Vector function reference.

VECTOR RELATIONAL EXPRESSIONS

A vector relational expression consists of a relational operator flanked by two expressions. The relational operators are:

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

The periods are part of the operators and must appear.

A vector relational expression has one of the following forms:

$sae\ op\ vae_1$

$vae_1\ op\ sae$

$vae_1\ op\ vae_2$

sae	A scalar arithmetic expression of type real or integer, but not of type complex or double-precision.
op	One of the relational operators.
vae_i	A vector arithmetic expression of type real or integer, but not of type complex or double-precision.

A vector relational expression, which always contains one or more vector data elements, evaluates to a bit vector of truth values represented by bits 0 and 1. (In contrast, evaluation of a scalar relational expression results in a single logical value.)

When both operands for a relational operation are vectors, the operation compares successive elements of one vector operand with corresponding elements of the other vector operand. If the specified relation holds between the pair of elements, the operation sets (assigns 1 to) the corresponding bit in the result bit vector. If the relation does not hold, the operation clears (assigns 0 to) the corresponding bit in the result bit vector. When one operand is a vector and the other a scalar, the scalar is compared with each element of the vector during evaluation of the expression.

Given the declaration:

```
DESCRIPTOR D1
```

the following are examples of vector relational expressions:

- X+Y/3.*Z .LT. VR(1;100)

  Bit vector having a length of 100 bits, where the i^{th} bit is 1 if the i^{th} element of the real vector VR(1;100) is greater than or equal to the value of the scalar arithmetic expression X+Y/3.*Z, and is 0 otherwise.

- D1 .NE. D1*2

  Bit vector having the length of the vector that D1 points to, where the i^{th} bit is 1 if an element of the vector is nonzero, and is 0 otherwise.

- VR(1;89) .EQ. VR(2;89)

  Bit vector having a length of 89 bits, where the i^{th} bit is 1 if the i^{th} element of the real vector VR(1;89) equals the i^{th} element of the real vector VR(2;89), and is 0 otherwise.

- R(10;20) .GE. 0.34

  Bit vector having a length of 20 bits, where the i^{th} bit is 1 if the i^{th} element of the real vector R(10;20) is greater than or equal to 0.34, and is 0 otherwise.

- VR(1;25)*3. .EQ. (VC(1;25)-ST+REM)

  Bit vector having a length of 25 bits, where the i^{th} bit is 1 if the i^{th} element of the real vector VR(1;25), multiplied by 3, is equal to the i^{th} element of the real vector VC(1;25) plus the value of REM-ST, and is 0 otherwise.

BIT EXPRESSIONS

A bit expression can be a vector relational expression, bit vector, bit descriptor, bit vector function reference, bit descriptor array element, bit constant, bit variable, bit array element, or a bit expression enclosed in parentheses. If B and C are vector bit expressions, then B followed by a logical operator followed by C is also a vector bit expression.

The operators used in vector 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 section 3, the precedences of the operators are the same as for logical operators in logical expressions.

A vector relational expression evaluates to an entire bit vector of logical results. Logical operations on bit vectors are performed on either corresponding elements of two vector operands or a scalar operand paired with successive elements of a vector operand.

Given the declarations:

```
BIT VC(100), BV, BV2, BVA
DESCRIPTOR BV, BV2, BVA
```

the following are examples of vector bit expressions:

- BV

  Value of the bit vector that BV points to.

- BV .OR. BV2

  Bit vector in which the i^{th} element is 1 unless the i^{th} elements of the vector values of BV and BV2 are 0, in which case the i^{th} element is 0.

- ((BV .OR. BV2) .OR. BVA)

  Bit vector in which the i^{th} element is 1 unless the i^{th} elements of the values of (BV .OR. BV2) and BVA are 0, in which case the i^{th} element is 0.

- (VC(1;25) .GE. 5.431) .AND. (VC(1;25) .LT. 5.931)

  Bit vector in which the i^{th} element is 1 if the i^{th} elements of the values of the two vector relational expressions are 1, and is 0 otherwise.

- B "101"

  Bit constant.

- B1

  Bit variable.

- .NOT . (B1 .AND.B2)

  Bit expression.

EXECUTABLE STATEMENTS

CYBER 200 FORTRAN vector programming statements include two assignment statements, one ASSIGN statement, and one specification statement. The types of vector expressions appearing in the assignment statements described here are shown in table 11-2.

TABLE 11-2. EXPRESSION TYPES THAT CAN APPEAR IN AN ASSIGNMENT STATEMENT

Expression \ Assignment Statement	Vector Arithmetic	Vector Bit
Vector arithmetic	x	x†
Vector relational		x
Vector bit		x

†A vector arithmetic expression can be a primary in a vector relational expression which in turn can be a primary in a vector bit expression.

DESCRIPTOR ASSIGN STATEMENT

At execution time, the descriptor ASSIGN statement can be used to associate a vector with a descriptor. The descriptor ASSIGN statement has two forms.

First form:

```
ASSIGN p,q
```

p A descriptor, or a descriptor array element.

q A vector, descriptor, or descriptor array element.

The type of p must agree with the type of q. The descriptors and vectors can be of type integer, real, bit, or complex; double-precision vectors cannot be assigned to a descriptor.

Execution of this form of the descriptor ASSIGN statement causes p to point to the vector q (if q is a vector) or to the vector that q points to (if q is a descriptor or descriptor array element). After the ASSIGN has been executed, the operand p points to a vector.

The use of the first form of the descriptor ASSIGN statement is illustrated in part A of figure 11-8. In the figure, a bit array is declared and given initial values. The rows of the bit array are then indicated to be vectors in the first descriptor ASSIGN statement within the DO. After the descriptor array element Y(I) has been given a bit vector value, the second ASSIGN statement assigns the same bit vector value to the descriptor array element Y(16-I). The result of completing the DO loop is the configuration shown in part B of figure 11-8.

Memory for a vector need not be allocated at compile time as happens when, for example, a vector name is used in the first form of the descriptor ASSIGN statement. The following descriptor ASSIGN statement form can be used to allocate dynamic space for a vector at execution time, and to cause a descriptor or descriptor array element to point to it.

A.

```
      DIMENSION Y(15),A(64,8)
      BIT Y,A
      DESCRIPTION Y
      DATA A/ . . .
      DO 10 I=1,8
        ASSIGN Y(I),A(1,I;64)
        ASSIGN Y(16-I),Y(I)
10    CONTINUE
```

B.

1	points to A(1,1;64)
2	points to A(1,2;64)
3	points to A(1,3;64)
4	points to A(1,4;64)
5	points to A(1,5;64)
6	points to A(1,6;64)
7	points to A(1,7;64)
8	points to A(1,8;64)
9	points to A(1,7;64)
10	points to A(1,6;64)
11	points to A(1,5;64)
12	points to A(1,4;64)
13	points to A(1,3;64)
14	points to A(1,2;64)
15	points to A(1,1;64)

Array Y

Figure 11-8. Example of Descriptor ASSIGN

Second form:

ASSIGN p, .DYN.e

p A descriptor, or a descriptor array element.

e An integer scalar expression that indicates a quantity of dynamic space in words or bits.

After execution of this second form of the descriptor ASSIGN statement, p points to a vector consisting of dynamic space having a length of e words if p is real or integer, e bits if p is bit, or 2e words if e is complex. Before the value of the dynamic space pointer is incremented, the current value of the dynamic space pointer is assigned as the base address of the vector pointed to by p.

Dynamic space is managed as a stack. After an assignment to dynamic space has been made with a descriptor ASSIGN statement, the dynamic space pointer (the location of the top of the stack) is incremented. The next space available is made to begin at the first doubleword boundary. Subsequently, the execution of a FREE statement or a RETURN statement releases all dynamic space allocated by descriptor ASSIGN statements in the program unit.

FREE STATEMENT

Execution of the FREE statement (or completion of program unit execution) reverses the effect of a descriptor ASSIGN statement in which a reference .DYN. to the dynamic space pointer appears. The FREE statement resets the dynamic space pointer to the value it had before execution of the first descriptor ASSIGN statement in the program unit. All space assigned through the use of descriptor ASSIGN statements is released; if more than one such assignment was made, all are reversed.

Form:

FREE

VECTOR ARITHMETIC ASSIGNMENT STATEMENT

A vector arithmetic assignment statement has the following form:

v=e

v A vector of type integer, real, complex, or double-precision; or a descriptor or descriptor array element of type integer, real, or complex.

e A vector arithmetic expression, or a scalar arithmetic expression.

The value of e is assigned to v . When v is double-precision, e can only be a double-precision vector or a reference to a predefined CYBER 200 FORTRAN double-precision vector function (listed in appendix E).

If e evaluates to a scalar, that scalar is stored into every element of v; but if e evaluates to a vector, the first element of e is stored into the first element of v, the second element of e is stored into the second element of v, and so on. If the type of v differs from that of e, type conversion takes place, during assignment, to the type of v. Type conversion rules are given in table 11-3.

Examples, given the following declarations:

```
DOUBLE PRECISION RES(30), DPN, EXN(30)
DESCRIPTOR D1, DEX
DATA DEX, D1/RES(1;20), EXN(1;15)/
```

- RES(1;30) = DSQRT(DPN)

  Replace the value of the i^{th} element of RES(1;30) with the value returned by the predefined function reference DSQRT(DPN). The sequential equivalent is:

```
      DO 3 I = 1,30
    3 RES(I) = DSQRT (DPN)
```

- RES(1;30) = EXN(1;30)

  Replace the value of the i^{th} element of RES(1;30) with the i^{th} element of EXN(1;30). The sequential equivalent is:

```
      DO 3 I = 1,30
    3 RES(I) = EXN(I)
```

TABLE 11-3. CONVERSION RULES FOR VECTOR ASSIGNMENT

V	e			
	Integer	Real	Double-precision	Complex
Integer	No conversion	Fix	Not applicable	Fix real part and discard imaginary part
Real	Float	No conversion	Not applicable	Use real part and discard imaginary part
Double-precision	Not applicable	Not applicable	No conversion	Not applicable
Complex	Float and use for real part; zero imaginary part	Use for real part; zero imaginary part	Not applicable	No conversion

- DEX = -(D1+N)/2.**M

  Cause DEX to point to the vector value of the vector arithmetic expression -(D1+N)/2.**M. The sequential equivalent is:

```
      DO 3 I = 1,15
3     EXN(I) = -(EXN(I)+N)/2.**M
      ASSIGN DEX, EXN(1;15)
```

- D1 = -30.0

  Replace the value of the i^{th} element of the vector pointed to by D1 with -30.0. The sequential equivalent is:

```
      DO 3 I = 1,15
3     EXN(I) = -30.0
      ASSIGN D1, EXN(1;15)
```

BIT ASSIGNMENT STATEMENT

A bit assignment statement has the following form:

v=e

v A bit vector, bit descriptor, bit descriptor array element, bit variable, or bit array element.

e A bit expression.

Execution of the statement causes the complete evaluation of e and the storing of the result into v. If v is longer than the result, the remaining rightmost bits of v are padded with zeros.

Examples, given the following declarations:

```
BIT BV(60), BD
DESCRIPTOR BD
```

- BV(1;60) = BD

  Replace the value of the i^{th} element of BV(1;60) with the value of the i^{th} element of the vector pointed to by BD.

- BD = BD .OR. BV(1;60)

  Cause BD to point to the vector value of the vector bit expression BD .OR. BV(1;60).

- BV = B'1'

  Set each element of the array BV to 1. The sequential equivalent is:

```
      DO 4 I = 1,60
4     BV(I) = B'1'
```

- BV = B'100'

  Set each element of the array to BV 1. (Note that the bit constant B '100' is truncated to B '1' since each element of the bit array BV is assigned separately.) The sequential equivalent is:

```
      DO 5 I = 1,60
5     BV(I) = B'100'
```

- BV (1,60) - B '1'

  Set the bit vector to 1 one and 59 zeros.

- BV (1,60) = B '101'

  Set the bit vector BV to 101 followed by 57 zeros.

DECLARATIONS

Vector programming adds one specification statement to the list of nonexecutable statements that can appear at the beginning of a CYBER 200 FORTRAN program unit.

DESCRIPTOR STATEMENT

The DESCRIPTOR statement has the following form:

DESCRIPTOR $v_1,v_2,\ldots,v_n$

v_i A variable, array declarator, or array name, of type real, integer, bit, or complex.

All variables in the DESCRIPTOR statement list are declared to be descriptors, and any array or array declarator list element specifies a descriptor array. For example, the statement pair:

```
DESCRIPTOR A,B,C(3,4)
REAL A,B(6,2),C
```

specifies A to be a real descriptor, and B and C to be descriptor arrays having 12 type real descriptor array elements each.

The type of v_i must be established with an explicit type declaration statement, or by the first-letter rule. Although vectors can be double-precision, a descriptor cannot be double-precision.

INITIALIZING DESCRIPTORS AND VECTORS

The nonexecutable DATA statement, described in section 6, can be used to place initial values in vectors and descriptors before the program begins executing. Double-precision vectors cannot be initialized in a DATA statement, although the double-precision array or individual array elements can be so initialized.

As described in section 6, a data initialization statement consists of pairs of lists; a list of variables is paired with a list of constants used as the initial values for the variables. Besides scalar list elements, the list of variables can include vectors, descriptors, descriptor arrays, and descriptor array elements.

For vectors, a vector name in the variable list must contain only integer constant subscript expressions and vector length specification. The number of constant list elements corresponding to the name must be equal to the length of the vector. For example, if a vector name in the variable list is A(1;10), then 10 consecutive constant list elements must correspond to the vector name. (This is similar to the way that arrays can be initialized in a DATA statement.)

For descriptors and descriptor array elements, a descriptor in the variable list must correspond only to a vector, which must contain only integer constant subscript expressions and vector length specification.

The repeat count specification in a DATA statement (section 6) can be used to specify the repeated use of a vector for initialization of more than one descriptor or descriptor array element. The data types of corresponding variable list and constant list items must, in the above cases, be the same.

Examples of initializing vector descriptors are given in section 16.

VECTOR FUNCTION SUBPROGRAMS

Vector function subprograms are defined in almost the same way that other function subprograms are defined. The differences lie in the argument list form, the number of data types available for vector function results, and the fact that the function name must appear in a DESCRIPTOR statement in the function subprogram.

Form:

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

- t — Optional declaration of the type of f. When present, t can be INTEGER, REAL, BIT, or COMPLEX but cannot be DOUBLE PRECISION.
- f — The function's symbolic name.
- a_i — Dummy argument. The possible dummy arguments here are the same as for scalar functions, n must be greater than or equal to 1.

The function name f must appear in a DESCRIPTOR statement within the function subprogram. If t is not specified, f can appear in a type statement or be typed implicitly. The semicolon in the dummy argument list is required to separate the input list from the dummy output argument, which is represented by the asterisk.

Refer to Function Subprograms in section 7 for a more detailed discussion of function names and function subprogram program units.

REFERENCING VECTOR FUNCTIONS

A vector function is referenced when the name of the function, followed by an actual argument list enclosed in parentheses, appears in an arithmetic expression in an arithmetic assignment statement. The actual arguments that can correspond to a dummy argument are shown in section 7.

The actual argument list in the function reference is divided into two parts by a semicolon. Input arguments precede the semicolon and are separated by commas; they can be scalar expressions, vectors, descriptors, or descriptor array elements. A single output argument of the function follows the semicolon and can be a vector, descriptor, or descriptor array element. The output argument must be the same data type as the function. CYBER 200 FORTRAN permits double-precision output arguments to be used in references only to some predefined CYBER 200 FORTRAN vector functions (listed in appendix E).

SECONDARY ENTRY POINTS

Vector function subprograms, like scalar function subprograms, can have multiple entry points defined for them. The ENTRY statement (described in section 7) specifies that the first executable statement following the ENTRY statement is a secondary entry point. More than one entry point can be declared in a subprogram; also, a scalar or vector function subprogram can have both scalar and vector secondary entry points.

Form:

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

e — The symbolic name of the entry point.

a_i — Dummy argument. The possible dummy arguments here are the same as for scalar ENTRY statements, n must be greater than or equal to 1.

Like the function name, the entry point name must appear in a DESCRIPTOR statement within the scalar or vector function subprogram. Again, the semicolon separates the dummy input argument list from the dummy output argument which is represented by the asterisk.

The statements made in section 7 with respect to referencing secondary entry point names apply to the referencing of the entry point names defined in a vector function subprogram.

This section outlines the calling, prologue, epilogue, and file initialization conventions observed by the CYBER 200 FORTRAN compiler, and describes in particular the possibilities for interfacing with predefined FORTRAN functions. Four points are elaborated upon:

- The FORTRAN compiler generates a standard prologue and epilogue sequence in the object code for subroutine and function subprograms.
- Except for some of the predefined FORTRAN functions, the FORTRAN compiler generates a standard calling sequence for all external procedures.
- In the appropriate environment, a fast calling sequence is generated by the FORTRAN compiler for calls to predefined FORTRAN functions.
- Input/output in a FORTRAN subprogram assumes that the files referenced by the input/output statements have been opened in the main program.

In this section all numbers designating registers are in hexadecimal; the # before a register number is a reminder of this.

PROLOGUE AND EPILOGUE

The FORTRAN compiler generates a prologue and an epilogue for subroutine calls and function references. For a non-zero-swap routine, the prologue code performs the following functions:

1. Saves the values in registers #1A through #1F.
2. Saves the values of the caller's registers and loads the registers with the values of the routine's registers.
3. Restores the values in registers #1A (return address), #1E (data base address), and #1F (DFBM table pointer).
4. Updates the values in registers #1B (dynamic stack address), #1C (current stack pointer), and #1D (previous stack pointer).
5. Clears the length field of register #1F for the Data Flag Branch Manager.

For a non-zero-swap routine, the epilogue code performs the following functions:

1. Saves the values of the routine's registers and loads the registers with the values of the caller's registers.
2. If the length field of register #1F is nonzero, restores the data flag branch register mask conditions of the caller, preserving the free data flags
3. Returns control to the address specified in register #1A.

A non-zero-swap routine is a routine that requires the values currently in the registers to be saved upon entering the routine. A zero-swap routine is a routine that does not require the values in all registers to be saved upon entering the routine. The compiler generates a zero-swap routine if all of the following conditions are met:

- Option O or Z was specified.
- There are no calls or function references (other than to FORTRAN routines that can be generated in-line).
- There are no input/output statements.
- There are no vectors used through either explicit or automatic vectorization.
- The generated code can be reasonably executed using only registers #3 through #13, and possibly registers #17, #18, and #19.
- There are no special calls.

STANDARD CALLING SEQUENCE

In general, FORTRAN observes the subroutine linkage conventions and register conventions described in volume 2 of the CYBER 200 Operating System reference manual. When a user-written FORTRAN procedure calls an external procedure, such as one written in assembly language, the standard calling sequence (in machine language) generated during compilation for this call is essentially as follows:

#78xx001E	Load register #1E with the address, xx, of the callee data base.
#78yy0017	Load register #17 with parameter list descriptor yy.
#361A00zz	Branch to the entry point zz of the called procedure after setting a return location in register #1A.

In the above instructions, registers #1E, #17, and #1A are the conventional data base register, parameter descriptor register, and return register respectively; xx contains the callee data base address, yy contains the descriptor of the parameter list, and zz contains the procedure entry point address. All of the other global and environment registers are initialized by the operating system.

If the procedure is a function, a function result is expected, on return, in register #18 (for a one-word result) or in registers #18 and #19 (for a two-word result). Specifically, logical, integer, and real functions return their results in register #18. Complex and double-precision functions return their results in registers #18 and #19. Character functions return the address of the result in register #18. Vector functions return results in the result vector; register #18 must have been preset by the caller to the address where the result vector is to be placed.

FAST CALLS

Many of the FORTRAN-supplied functions have a fast call entry point as well as a standard call entry point. The FORTRAN compiler generates a fast call to any of these functions unless the function name appears in an EXTERNAL statement in the calling program. The standard call entry point name is the function name; appendix E contains a list of the equivalent fast call entry point names (not all FORTRAN-supplied functions have fast call entry points). FORTRAN does not generate fast calls to procedures which have user-supplied names.

The difference between standard and fast calling sequences is the method by which parameters are passed to the called procedure. Whereas parameters in a standard call are passed via a parameter list in memory, fast call parameters are passed in the lower area of the register file. Fast call parameters are passed in temporary registers #3, #4, #5, and #6, as required by the number and length of the function parameters. Results are returned as for functions called with the standard calling sequence.

A fast call to a scalar function with one argument could appear as follows:

#78xx001E	Load register #1E with the address of the callee data base.
#78yy0003	Load register #3 with the function's actual argument.
#361A00zz	Branch to the fast call entry point of the called function and set a return location in register #1A.

In the above instructions, xx contains the callee data base address, yy contains the function parameter, and zz contains the procedure function entry point address. Function parameters must be loaded in consecutive registers, beginning with register #3 and in the order specified in the function descriptions given in section 14.

Placing the name of a FORTRAN-supplied function name in an EXTERNAL statement suppresses generation of fast calling sequences for references to the function. That is, a standard calling sequence is generated for any function whose name appears in an EXTERNAL statement; in particular, predefined functions which would otherwise be referenced with a fast calling sequence (i.e., those functions having only an external version as listed in appendix E) are referenced using the standard calling sequence.

FILE INITIALIZATION

One purpose of the PROGRAM statement is to initialize files on which input/output is to be performed during program execution, including the files referenced in subprograms of the program. The PROGRAM statement parameter list informs the FORTRAN compiler that the files listed are to be created if they do not exist already and are to be opened for input/output. Only output with PRINT statements can be performed when no PROGRAM statement is used.

When the main program referencing a FORTRAN subprogram that performs input/output has been written in assembly language or implementation language (IMPL) instead of in FORTRAN, no PROGRAM statement exists to perform the required file initialization. In this case the assembly language or IMPL program must initialize the files explicitly. Initialization is performed by setting up register #3, then referencing the entry point FT_INIT.

Only the file OUTPUT is opened if register #3 is set to all 0 bits. Register #3 can alternatively be set to a descriptor pointing to a character string that is a PROGRAM statement file information parameter list, not including parentheses. The length field (bits 0 through 15) of register #3 must be the length in characters of the character string, and the address portion (bits 16 through 63) must be the virtual bit address of the string's first character.

The following types of FORTRAN-supplied subroutines can be called from a CYBER 200 FORTRAN program:

- Special calls

  Used to place specific CYBER 200 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.

CYBER 200 FORTRAN SPECIAL CALLS

CYBER 200 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 appropriate 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($a_1, \ldots, a_n$)

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 CYBER 200 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 CYBER 200 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 when indexed branch instructions (#B0 through #B5) have a zero in bit 2 of the G field, the combined Y and B fields require only one argument; this argument is usually a label reference. When bit 2 of the G field contains a one, each field requires an argument and the second field must be zero or null. For a nonrelative branch, the Y and B fields represent two register designators: index and base address. In this case, the user must set the G field bit 2 to zero and use a 16-bit hexadecimal constant for the fourth argument or operand.

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 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 CYBER 200 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

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 halfword 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. Branch control bits 5 and 6 in the G field should be set accordingly. No checking is done to verify that the instruction branches in the correct direction.

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 halfword offset from the special CALL statement, to the statement tagged by the label. If the resultant halfword 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 halfword registers). Registers modified by branch instructions cannot be referenced symbolically.

Literals

A literal can be a decimal, hexadecimal, bit, or character 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. Hollerith constants are not permitted in special calls.

EXAMPLES OF SPECIAL CALL USAGE

The call to Q8BSAVE shown in figure 13-1 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 13-1. Special CALL Statement

The calls in figure 13-2 produce identical results; each call enters the character string AB in register #41. 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 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 13-2. Q8ES Usage

The special calls in figure 13-3 generate the machine code shown in figure 13-4 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 13-3. Additional Q8 Usage

```
ES      R3,1
ES      R4,2
ADDX    R3,R4,R22
```

Figure 13-4. Generated Machine Code

If J has not been assigned any register by the compiler, the code shown in figure 13-5 would be generated.

```
ES      R3,1
ES      R4,2
ADDX    R3,R4,T1
STO     (DATA BASE, RELATIVE
        LOCATION OF J),T1
```

Figure 13-5. Additional Generated Code

DATA FLAG BRANCH MANAGER

The data flag branch manager (DFBM) is a FORTRAN run-time and CYBER 200 library routine. A data flag branch is a hardware function of the CYBER 200 computers. 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, 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
- Reduction of the job interval timer to zero (cannot occur unless the program sets the JIT explicitly or calls the FORTRAN-supplied routine SECOND)
- 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 CYBER 200 central processor, is formatted as shown in figure 13-6. 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 CYBER 200 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 CYBER 200 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. CYBER 200 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

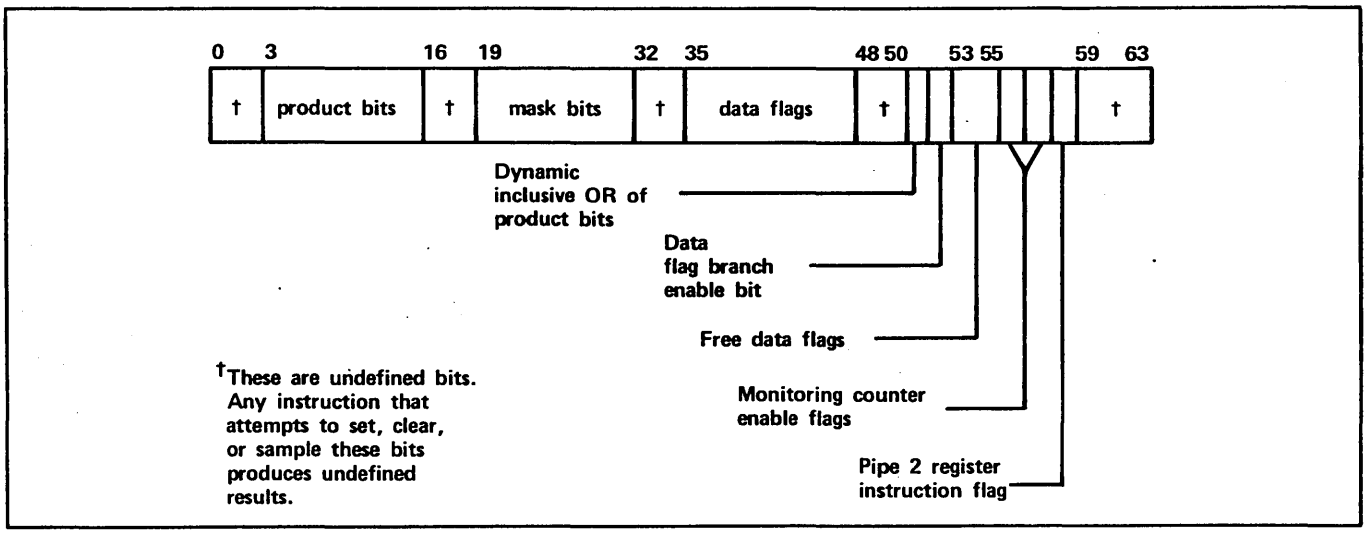


Figure 13-6. Data Flag Branch Register Format

CYBER 200 hardware clear and set bit 52. When both bit 51 and bit 52 are set, the CYBER 200 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 13-1. 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.

TABLE 13-1. 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.

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 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 CYBER 200 input/output.

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 DFBM interrupt does not return control to the operating system. 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 CYBER 200 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)
2. SFT (bit 3)
3. BKP (bit 15)
4. ORX (bit 12)
5. ORD (bit 8)
6. SRT (bit 13)
7. IND (bit 14)
8. FDV (bit 9)
9. EXO (bit 10)
10. RMZ (bit 11)
11. SSC (bit 5)
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 13-1 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; a DFBM interrupt 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 13-2. 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 13-2. 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 can 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 postmortem 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 CYBER 200 Operating System reference manual.

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 13-7. 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, reenabling data flag branches.
5. Calls the class III interrupt-handling routine.

```
ERROR 124 EXECUTION INTERRUPTED IN INDEF AT LINE 5
DATA FLAG BRANCH - INDEFINITE RESULT - REGISTER 1 ADDRESS 000000012260

DATA FLAG BRANCH REGISTER

00000000 01000010 00011000 01000111 00000000 01001010 00010000 00100000

                      SFT JIT SSC DUF THZ ORO FDV EXO AM7 ORA 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 13-7. DFB Register Dump Example

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 13-8. 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.

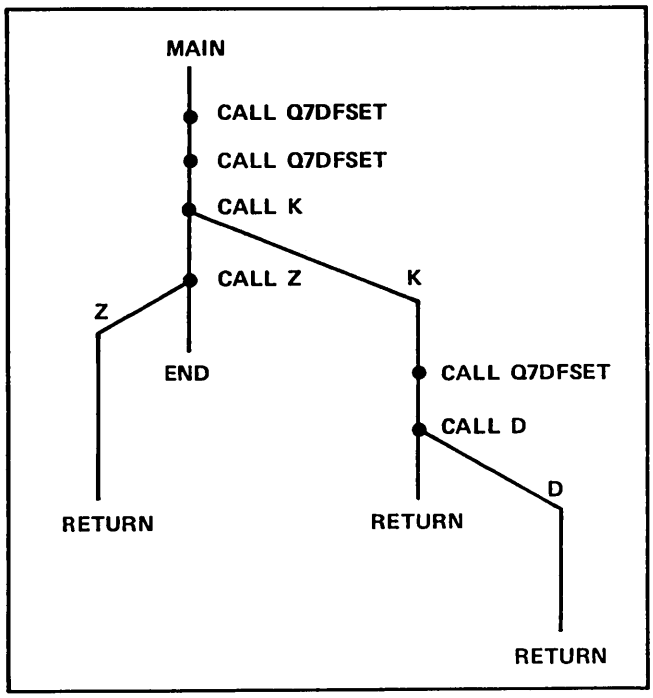


Figure 13-8. Scope of Selected Conditions

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 by 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 13-1. The designator must be enclosed in apostrophes. A designator from table 13-1 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 that USRRTN is 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 13-1.

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 13-1.

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 13-1. A designator from table 13-1 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 causes DFB register bits 25 and 30 to be cleared at the time that DFBM restarts the interrupted FORTRAN program. The second call causes 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.

The initial call to the FORTRAN-supplied routine SECOND in a FORTRAN program invokes routine Q7DFCL1 to specify a special JIT interrupt-handling routine. If a user-provided JIT interrupt routine is also specified in the same program, an interrupt processing conflict occurs and fatal run-time error 140 or 141 is issued. To force the program to execute, the System Error Processor routine SEP can be invoked before the initial call to SECOND to reset the error class to warning. When the JIT interrupt condition occurs, DFBM branches to the most recently specified JIT interrupt-handling routine. If the user-provided routine is the most recently specified JIT interrupt-handling routine, the results of SECOND are undefined.

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 assembly language 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, 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 assembly language subroutines of a FORTRAN program by 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 13-9. An array I was declared and initialized by using the two statements:

```
DIMENSION I(20)
DATA      I/5*7,15*12/
```

and then by 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 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 CYBER 200 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, 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.

```
        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 13-9. MDUMP Output

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, 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 INPUT/OUTPUT SUBROUTINES

The mass storage input/output subroutines for concurrent input/output transmit data in an optimal manner between main memory and files on mass storage. No buffers are required and no structuring information is processed when a concurrent input/output 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.

The four concurrent input/output 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 input/output routines must be declared in the PROGRAM statement. 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 input/output 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 input/output 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. The user is not notified of error conditions, but 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 input/output 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 input/output 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, 12288 words), the array should be placed on a large page to obtain the input/output efficiency that is derived from using concurrent input/output. 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 65536 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 input/output requests must be issued by the system to transfer that array. When multiple explicit input/output requests are issued, concurrent processing ceases after the first of the multiple requests completes and cannot resume during the remainder of the input/output for that call. If the array did not overlap a large page boundary, a single explicit input/output 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 input/output routines. The program then should also contain the statement:

```
COMMON/ANAME/BIGRAY(10240),RA2(51200)
```

which declares an array BIGRAY with 10240 words and an array RA2 with 51200 words to reside in the labeled common block ANAME. After the program is compiled (by using the system control statement FORTRAN.), loading is performed by 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 input/output 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 input/output 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).

Q7BUFIN

The Q7BUFIN subroutine transfers data from a mass storage file to an array in main memory. 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 input/output 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. 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 input/output 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. Input/output 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. While data transmission is 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 input/output 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 input/output operation:

- 0 = Normal completion
- 1 = Physical end-of-file reached
- 2 = Data transfer error due to hardware failure
- 3 = Input/output operation not yet completed

ret — Optional. Integer constant or integer variable specifying action to be taken upon return from Q7WAIT call:

- 0 = If input/output is in progress at time of call, program should wait (computation should cease) until input/output is completed normally or abnormally. Default.
- 1 = If input/output is 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 input/output 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.

A call to Q8WIDTH is only valid for files with control word, record mark, or fixed-length record types. A call to Q8WIDTH must precede any other access to the file.

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.

Q8NORED SUBROUTINE

The subroutine Q8NORED enables a user to suppress file size reduction. Files created by a program are initially 128 small pages long, but are reduced to minimal size upon completion of the program. If Q8NORED is used, the file will remain 128 small pages long.

The format of the Q8NORED subroutines is:

CALL Q8NORED
CALL Q8NORED($u_1, \ldots, u_n$)

u_i logical unit number of a file

If no parameters are present, no files will be reduced.

SUPPLIED SUBROUTINES

A number of predefined subroutines are provided with the CYBER 200 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'000054F4A3B933BD'.

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.

Because SECOND uses the job interval timer (JIT), a user program that manipulates the job interval timer invalidates the returned result. Furthermore, a user program that attempts to perform JIT interrupt processing conflicts with SECOND and causes the program to abort. See the description of the data flag branch manager.

TIME

This subroutine generates the same result as the TIME function described in section 14. 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 13-3 and table 13-4.

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).

TABLE 13-3. STACKLIB CALLS WITH FORWARD COUNT

Description	Type	STACKLIB Call With Sample Arguments	Equivalent Statement Contained in the Loop DO xxI = 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))
Sum of vector elements	Dyadic	CALL Q8DA0000(S,A(2),N-1)	S=S+A(I)
Dot product of 2 vectors	Triadic	CALL Q8DC0000(S,A(2),B(2),N-1)	S=S+A(I)*B(I)
Dot product of 1 vector	Triadic	CALL Q8DC0010(S,A(2),N-1)	S=S+A(I)*A(I)
Multiply, add	Triadic	CALL Q8AM201(A(2),B(2),C,D(2),N-1)	A(I)=B(I)+C*D(I)
Multiply, add	Triadic	CALL Q8AM101(A(2),B(2),D(2),C,N-1)	A(I)=B(I)+D(I)*C
Multiply, add	Triadic	CALL Q8MA200(A(2),B(2),C,D(2),N-1)	A(I)=B(I)*C+D(I)
Multiply, add	Triadic	CALL Q8MA400(A(2),C,B(2),D(2),N-1)	A(I)=C*B(I)+D(I)
Multiply, subtract	Triadic	CALL Q8SM201(A(2),B(2),C,D(2),N-1)	A(I)=B(I)-C*D(I)
Multiply, subtract	Triadic	CALL Q8SM101(A(2),B(2),D(2),C,N-1)	A(I)=B(I)-D(I)*C

TABLE 13-4. STACKLIB CALLS WITH BACKWARD COUNT

Description	Type	STACKLIB Call With Sample Arguments	Equivalent Statement as Contained in the Loop DO xxI = 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-1),B(N-1),S,A(N),N-1)	A(J)=(B(J)*S)+A(J+1)
Multiply add, recursive v1, scalar v4	Triadic	CALL Q8MA412(A(N-1),S,B(N-1),A(N),N-1)	A(J)=(S*B(J))+A(J+1)
Multiply add, recursive v4, scalar v1, reverse order	Triadic	CALL Q8AM143(A(N-1),A(N),B(N-1),S,N-1)	A(J)=A(J+1)+(B(J)*S)
Multiply add, recursive v4, scalar v2, reverse order	Triadic	CALL Q8AM243(A(N-1),A(N),S,B(N-1),N-1)	A(J)=A(J+1)+(S*B(J))
Subtract multiply, recursive v1, reverse order	Triadic	CALL Q8SM013(A(N-1),B(N-1),C(N-1),A(N),N-1)	A(J)=B(J)-(C(J)*A(J+1))
Subtract multiply, recursive v2, reverse order	Triadic	CALL Q8SM023(A(N-1),B(N-1),A(N),C(N-1),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-1),S,A,(N),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-1),S,T,A(N),N-1)	A(J)=S/(T+A(J+1))
Multiply, add, recursive v1, reverse order	Triadic	CALL Q8AM013(A(N-1),B(N-1),C(N-1),A(N),N-1)	A(J)=(B(J)*C(J))+A(J+1)
Multiply, add, recursive v2, reverse order	Triadic	CALL Q8AM023(A(N-1),B(N-1),A(N),C(N-1),N-1)	A(J)=(B(J)*A(J+1))+C(J)

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 1=forward count; 2 or 3=backward count; 0 or 2=forward order of operations; 1 or 3=reverse order of operations).

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) (f) 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) (f) v2(ind) (s) v1(ind)
```

The (f) and (s) 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) (f) v2(irev) (s) 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.

A group of predefined functions is provided with the CYBER 200 FORTRAN compiler. These functions, listed and described in this section, perform 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, suffixed with an appropriate list of actual arguments, as a data element in an arithmetic or logical expression. The actual arguments 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.

IN-LINE AND EXTERNAL

The functions fall into three categories - functions that when referenced:

- Cause in-line code to be generated during compilation.
- Cause transfer of program control to an external module during execution.
- Can generate either in-line code or a transfer to an external module.

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 external 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.

Appendix E contains a list of the functions categorized into the three types.

SCALAR AND VECTOR

Each function also falls into one of two categories according to the generated result:

- Scalar result - one set of computations that result in one value
- Vector result - one or more sets of computations that result in one or more values

Also, CYBER 200 FORTRAN provides a group of Q8-prefixed functions that perform more involved manipulations with vectors than the V-prefixed functions.

The functions, function reference form, and type of arguments and result are given in table 14-1. In the table, the letters a and b are used for scalar arguments, the letter v is used for vector arguments, the letter c is used for control vectors, the letter i is used for index vectors, and the letter u is used for vector results. Scalar arguments can be general scalar expressions. Vector arguments must be vectors or descriptors.

TABLE 14-1. FORTRAN-SUPPLIED FUNCTIONS

Function	Function Reference	Type of Arguments (other than c and i)†††	Type of Result
Obtain absolute value of argument	ABS (a)	Real	Real
	IABS (a)	Integer	Integer
	DABS (a)	Double	Double
	VABS (v;u)	Real	Real
	VIABS (v;u)	Integer	Integer
Truncate argument	AINT (a)	Real	Real
	INT (a)	Real	Integer
	DINT (a)	Double	Double
	IDINT (a)	Double	Integer
	VAINT (v;u)	Real	Real
	VINT (v;u)	Real	Integer
Calculate remainder: $(a_1 - [a_1/a_2]\, a_2)$†	AMOD (a_1,a_2)	Real	Real
	MOD (a_1,a_2)	Integer	Integer
	DMOD (a_1,a_2)	Double	Double
	VAMOD (v_1,v_2;u)	Real	Real
	VMOD (v_1,v_2;u)	Integer	Integer

TABLE 14-1. FORTRAN-SUPPLIED FUNCTIONS (Contd)

Function	Function Reference	Type of Arguments (other than c and i)†††	Type of Result
Choose the largest value from among two or more arguments	AMAX0 (a_1,a_2, . . .)	Integer	Real
	AMAX1 (a_1,a_2, . . .)	Real	Real
	MAX0 (a_1,a_2, . . .)	Integer	Integer
	MAX1 (a_1,a_2, . . .)	Real	Integer
	DMAX1 (a_1,a_2, . . .)	Double	Double
Choose the smallest value from among two or more arguments	AMIN0 (a_1,a_2, . . .)	Integer	Real
	AMIN1 (a_1,a_2, . . .)	Real	Real
	MIN0 (a_1,a_2, . . .)	Integer	Integer
	MIN1 (a_1,a_2, . . .)	Real	Integer
	DMIN1 (a_1,a_2, . . .)	Double	Double
Float argument (convert from integer to real or double-precision)††	FLOAT (a)	Integer	Real
	DFLOAT (a)	Integer	Double
	VFLOAT (v;u)	Integer	Real
Fix argument (convert from real to integer)††	IFIX (a)	Real	Integer
	VIFIX (v;u)	Real	Integer
Transfer sign from second argument to first (second must not be zero)	SIGN (a_1,a_2)	Real	Real
	ISIGN (a_1,a_2)	Integer	Integer
	DSIGN (a_1,a_2)	Double	Double
	VSIGN (v_1,v_2;u)	Real	Real
	VISIGN (v_1,v_2;u)	Integer	Integer
Calculate positive difference between two arguments: (a_1 - MIN(a_1,a_2))	DIM (a_1,a_2)	Real	Real
	IDIM (a_1,a_2)	Integer	Integer
	DDIM (a_1,a_2)	Double	Double
	VDIM (v_1,v_2;u)	Real	Real
	VIDIM (v_1,v_2;u)	Integer	Integer
Convert from double-precision to real††	SNGL (a)	Double	Real
		Real	Real
	VSNGL (v;u)	Double	Real
Obtain real part of complex argument	REAL (a)	Complex	Real
	VREAL (v;u)	Complex	Real
Obtain imaginary part of complex argument	AIMAG (a)	Complex	Real
	VAIMAG (v;u)	Complex	Real
Convert from real to double-precision††	DBLE (a)	Real	Double
		Double	Double
	VDBLE (v;u)	Real	Double
Express two real arguments in complex form	CMPLX (a_1,a_2)	Real	Complex
	VCMPLX (v_1,v_2;u)	Real	Complex
Obtain conjugate of a complex argument	CONJG (a)	Complex	Complex
	VCONJG (v;u)	Complex	Complex
Obtain double-precision product of two real arguments	DPROD (a_1,a_2)	Real	Double
Sum vector's elements	Q8SSUM (v) or	Real	Real
	Q8SSUM (v,c)	Integer	Integer
Calculate product of vector's elements	Q8SPROD (v) or	Real	Real
	Q8SPROD (v,c)	Integer	Integer
Calculate dot product of two vectors	Q8SDOT (v_1,v_2)	Integer	Integer
		Real	Real

TABLE 14-1. FORTRAN-SUPPLIED FUNCTIONS (Contd)

Function	Function Reference	Type of	
		Arguments (other than c and i)†††	Result
Count elements having value of 1	Q8SCNT (v)	Bit	Integer
Obtain length of vector	Q8SLEN (v)	Real Integer Complex	Integer Integer Integer
Find minimum value in vector	Q8SMIN (v) or Q8SMIN (v,c)	Real Integer	Real Integer
	Q8SMINI (v) or Q8SMINI (v,c)	Real	Integer
Find maximum value in vector	Q8SMAX (v) or Q8SMAX (v,c)	Real Integer	Real Integer
	Q8SMAXI (v) or Q8SMAXI (v,c)	Real	Integer
Find first pair of elements satisfying the specified relation	Q8SEQ (v_1,v_2)	Real Integer	Integer Integer
	Q8SGE (v_1,v_2)	Real Integer	Integer Integer
	Q8SLT (v_1,v_2)	Real Integer	Integer Integer
	Q8SNE (v_1,v_2)	Real Integer	Integer Integer
	Q8VEQI (v_1,v_2;u) Q8VGEI (v_1,v_2;u) Q8VLTI (v_1,v_2;u) Q8VNEI (v_1,v_2;u)	Real Real Real Real	Integer Integer Integer Integer
Test data flag branch register	Q8SDFB (a,b)	Integer	Logical
Mask values in two vectors into result vector	Q8VMASK (v_1,v_2,c;u)	Real Integer	Real Integer
Merge values in two vectors into result vector	Q8VMERG (v_1,v_2,c;u)	Real Integer	Real Integer
Delete selected elements from vector	Q8VCMPRS (v,c;u)	Real Integer	Real Integer
Insert zero-elements into vector	Q8VXPND (v,c;u)	Real Integer	Real Integer
Merge every i-th element into result vector	Q8GATHP(v,i,n;r)	Real Integer	Real Integer
Index into vector argument to select elements for result vector	Q8VGATHR (v,i;u)	Real Integer	Real Integer
Replace every i-th element in result vector	Q8SCATP(v,i,n;r)	Real Integer	Real Integer
Scatter elements of vector argument into indexed result vector elements	Q8VSCATR (v,i;u)	Real Integer	Real Integer
Store selected elements in result vector	Q8VCTRL (v,c;u)	Real Integer	Real Integer

TABLE 14-1. FORTRAN-SUPPLIED FUNCTIONS (Contd)

Function	Function Reference	Type of Arguments (other than c and i)†††	Type of Result
Reverse the element order in a vector	Q8VREV (v;u)	Real Integer	Real Integer
Create vector whose adjacent elements differ in value by one specified amount	Q8VINTL (a_1,a_2;u)	Real Integer	Real Integer
Compute a polynomial at several values	Q8VPOLY (v_1,v_2;u)	Real	Real
Compute the differences between adjacent elements of a vector	Q8VDELT (v;u)	Real	Real
Create a bit pattern	Q8VMKZ (a_1,a_2;u) Q8VMKO (a_1,a_2;u)	Integer Integer	Bit Bit
Average each pair of adjacent elements of a vector	Q8VADJM (v;u)	Real	Real
Average each pair of corresponding elements for two vectors	Q8VAVG (v_1,v_2;u)	Real	Real
Compute the average differences between corresponding elements for two vectors	Q8VAVGD (v_1,v_2;u)	Real	Real
Exponential: e^a	EXP(a) DEXP(a) CEXP(a) VEXP(v;u) VCEXP(v;u)	Real Double Complex Real Complex	Real Double Complex Real Complex
Natural logarithm: $\log_e a$	ALOG(a) DLOG(a) CLOG(a) VALOG(v;u) VCLOG(v;u)	Real Double Complex Real Complex	Real Double Complex Real Complex
Common logarithm: $\log_{10} a$	ALOG10(a) DLOG10(a) VALOG10(;u)	Real Double Real	Real Double Real
Sine	SIN(a) DSIN(a) CSIN(a) VSIN(v;u) VCSIN(v;u)	Real Double Complex Real Complex	Real Double Complex Real Complex
Cosine	COS(a) DCOS(a) CCOS(a) VCOS(v;u) VCCOS(v;u)	Real Double Complex Real Complex	Real Double Complex Real Complex
Tangent	TAN(a) DTAN(a) VTAN(v;u)	Real Double Real	Real Double Real
Cotangent	COTAN(a)	Real	Real
Arctangent	ATAN(a) DATAN(a) VATAN(v;u)	Real Double Real	Real Double Real
Arctangent of a/b	ATAN2(a,b) DATAN2(a,b) VATAN2(v_1,v_2;u)	Real Double Real	Real Double Real

TABLE 14-1. FORTRAN-SUPPLIED FUNCTIONS (Contd)

Function	Function Reference	Type of Arguments (other than c and i)†††	Type of Result
Arcsine	ASIN(a)	Real	Real
	DASIN(a)	Double	Double
	VASIN(v;u)	Real	Real
Arccosine	ACOS(a)	Real	Real
	DACOS(a)	Double	Double
	VACOS(v;u)	REal	Real
Hyperbolic sine	SINH(a)	Real	Real
	DSINH(a)	Double	Double
Hyperbolic cosine	COSH(a)	Real	Real
	DCOSH(a)	Double	Double
Hyperbolic tangent	TANH(a)	Real	Real
	DTANH(a)	Double	Double
Square root	SQRT(a)	Real	Real
	DSQRT(a)	Double	Double
	CSQRT(a)	Complex	Complex
	VSQRT(v;u)	Real	Real
	VCSQRT(v;u)	Complex	Complex
Modulus: $(x^2+y^2)^{1/2}$ where x is the real part and y is imaginary part of the argument	CABS(a)	Complex	Real
	VCABS(v;u)	Complex	Real
Insert or extract bits	Q8SINSB(a,m,n,b)	Real	Typeless
		Integer	Typeless
	Q8SEXTB(a,m,n)	Real	Typeless
		Integer	Typeless
Random number	RANF(d)	(dummy)	Real
Time of day	TIME(d)	(dummy)	Character*8
Date	DATE(d)	(dummy)	Character*8
CPU time in seconds since job start	SECOND(d)	(dummy)	Real

†[x] is defined as the sign of x times the largest integer less than or equal to |x|. The results are not defined when the second argument is zero.

††Provides the same effect as the implied conversion in assignment statements.

†††Each control vector c is type bit, and each index vector i is type integer.

FUNCTION DESCRIPTIONS

The following descriptions are listed in strict alphabetical order. However, since a naming convention uses the letter V as a prefix to scalar function names to produce the corresponding vector function names, all functions with vector results can be found under V and Q8V. If a vector function input argument can be a vector, it is implied that it can also be a descriptor or descriptor array element. Also, except for some of the vector functions, none of the functions alters the values of its arguments. The mathematical values of some of the mathematical functions can be infinite.

A generic function generates a real or integer result, depending on the mode of the argument. For instance, the Q8SSUM function is a generic function.

A typeless function generates a result that is not converted for use as an argument or for assignment. For instance, the Q8SINSB function is a typeless function.

ABS(a)

For a real number x, ABS(x) computes the absolute value |x|.

ACOS(a)

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)

For a real number x, AINT(x) computes [x], where [A] is the sign of A times the largest integer less than or equal to |A|. AINT returns a real result even though its value is always integral.

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.

For a given real number x, ALOG(x) is calculated as follows.

For x outside the range:

$$((2)^{1/2}/2 \leq x < (2)^{1/2})$$

let:

$$x = 2^n * w$$

where:

$$1/2 \leq w < 1$$

and n is an integer that satisfies the equation.

Also, let:

$$t = (w - (2)^{1/2}/2)/(w + (2)^{1/2}/2)$$

Then:

$$\log_e(x) = (n - 1/2) * \log_e(2) + \log_e((1 + t)/(1 - t))$$

For x in the range:

$$((2)^{1/2}/2 \leq x < (2)^{1/2})$$

let:

$$t = (x - 1) / (x + 1)$$

Then:

$$\log_e(x) = \log_e((1 + t) / (1 - t))$$

In either case:

$$\log_e((1 + t) / (1 - t)) = 2t \sum_{n=0}^{6} c_n t^{2n}$$

where:

$$c_0 = 1.00000000000000000172016224 * 10^0$$

$$c_1 = 3.33333333332761817688528 3 * 10^{-1}$$

$$c_2 = 2.00000000309807789089930 7 * 10^{-1}$$

$$c_3 = 1.42857079946082734726139 8 * 10^{-1}$$

$$c_4 = 1.11117183115434280671900 0 * 10^{-1}$$

$$c_5 = 9.06093565817935371721425 4 * 10^{-2}$$

$$c_6 = 8.41918657586305313753481 7 * 10^{-2}$$

If a zero or negative argument is received, a data flag branch occurs inside the routine.

ALOG10(a)

This computes the logarithm of a real number. The result is a real number that is accurate to approximately 45 bits.

For a given real number x greater than zero:

$$\log_{10}(x) = \log_{10}(e) * \log_e(x)$$

where the natural logarithm is computed as described for the function ALOG.

If a zero or negative argument is received, a data flag branch occurs inside the routine.

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|.

ASIN(a) AND ACOS(a)

These 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.

For a given real number x:

$\text{asin}(x) = \text{asin}(u)$ if $0 \leq x \leq 1/2$, where $u = x$

$\text{asin}(x) = \text{pi}/2 - 2 * \text{asin}(u)$ if $1/2 < x \leq 1$

where $u = (1 - x/2)^{1/2}$

$\text{asin}(x) = -\text{asin}(-x)$ if $-1 \leq x < 0$

and asin(u) is calculated from a polynomial of degree 22.

Also:

$\text{acos}(x) = \text{pi}/2 - \text{asin}(x)$.

If an argument of magnitude greater than 1 is received, a data flag branch occurs inside the routine.

ATAN(a)

This 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).

For a given real number x:

$\text{atan}(x) = \text{sign}(x) * \text{atan}(v)$

where:

$v = |x|$

Then:

$\text{atan}(v) = \text{atan}(r) + c$

where r and c are:

If $0 \leq v < p$, $r = v$ and $c = 0.0$

If $p \leq v < 2^{1/2} - 1$, $r = (v - p)(1 + v * p)$
and $c = \text{pi}/16$

If $2^{1/2} - 1 \leq v < 1$, $r = (v - t)/(1 + v * t)$
and $c = 3\text{pi}/16$

If $1 \leq v < 2^{1/2} + 1$, $r = (v * t - 1)/(v + t)$
and $c = 5\text{pi}/16$

where:

$p = \tan(\text{pi}/16)$ and $t = \tan(3\text{pi}/16)$

Then:

$\text{atan}(r) = r - r * q$

where:

$q = (C_0 + C_1 * Z^1 + C_2 * Z^2 + \ldots + C_6 * Z^6$

$Z = r^2$

$C_0 = 0.999999999999998$

$C_1 = 0.333333333330652$

$C_2 = 0.199999998910139$

$C_3 = 0.142856976561312$

$C_4 = 0.111099001318911$

$C_5 = 0.904542314114089 * 10^{-1}$

$C_6 = 0.683464392415994 * 10^{-1}$

ATAN2(a,b)

This 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.

For given real numbers x and y, the result is in the range:

-pi to -pi/2 if $x < 0$, $y \leq 0$

-pi/2 to 0 if $x \geq 0$, $y < 0$

0 to pi/2 if $x \geq 0$, $y \geq 0$

pi/2 to pi if $x < 0$, $y > 0$

ATAN2(x,y) computes the arctangent as follows:

$\text{atan}(x/y) = \text{sign}(x) * \text{pi}/2$ if $y = 0$

$\text{atan}(x/y) = \text{sign}(x) * \text{ATAN}(x/y)$ if $y > 0$

$\text{atan}(x/y) = \text{pi} - \text{ATAN}(x/y)$ if $y < 0$, $x \geq 0$

$\text{atan}(x/y) = \text{ATAN}(x/y) - \text{pi}$ if $y < 0$, $x < 0$

The result is greater than or equal to zero for $x \geq 0$, and negative for $x < 0$. (ATAN is the function that computes the arctangent of a real number.)

If unacceptable arguments are received, the message:

X=Y=0.0

is issued, the result is set to indefinite, and a normal exit is taken from ATAN2.

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.

For a given complex number:

$x = u + iv$

the result is:

$(u^2 + v^2)1/2 + 0i$

where the square root function is evaluated by the machine instruction SQRT.

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.

For a given complex number $x=u+iv$, CCOS(x) is computed as follows. If $|u|>.110534964875444 * 10^{15}$ or if $v>19905.80$, the result is set to indefinite, an error message is issued, and a normal exit is taken from CCOS.

Otherwise, the complex result is:

$r + is$

where:

$r = \cos(u) * (e^{v} + e^{-v})/2$

$s = -\sin(u) * (e^{v} - e^{-v})/2$ for $|v| \geq 0.5$

$$s = -\sin(u) * v * \sum_{n=0}^{5} c_n v^{2n} \text{ for } |v| < 0.5$$

where:

$c_0 = .999999999999999811672 * 10^{0}$

$c_1 = .166666666666721232395 * 10^{0}$

$c_2 = .833333333307759961 * 10^{-2}$

$c_3 = .1984127027907999 * 10^{-3}$

$c_4 = .275569807356154 * 10^{-5}$

$c_5 = .251726188251 * 10^{-7}$

The real-valued sine, cosine, and exponential functions are evaluated as described for the SIN, COS, and EXP routines.

If an unacceptable argument is received, one of the messages:

ABS (REAL PART) TOO LARGE

IMAG. PART TOO LARGE

is issued, both real and imaginary parts of the result are set to indefinite, and a normal exit is taken from CCOS.

CEXP(a)

This computes the exponential of a complex number. The result is a complex number that is accurate to approximately 45 bits.

For a given complex number $x=u+iv$, the procedure for calculating CEXP(x) is as follows.

If $u>19905.80$, or if $|v|>.110534964875444 * 10^{15}$, both the real and imaginary parts of the result are set to indefinite, an error message is issued, and a normal exit is taken from CEXP.

Otherwise, the complex result is:

$r + is$

where:

$r = \cos(v) * e^{u}$

$s = \sin(v) * e^{u}$

The real-valued sine, cosine, and exponential functions are evaluated as described for the functions SIN, COS, and EXP.

If the function argument is out of range, one of the messages:

REAL PART TOO LARGE

ABS (IMAG PART) TOO LARGE

is issued, the result is set to indefinite, and a normal exit is taken from CEXP.

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.

For a given complex number $x=u+iv$, the procedure for calculating CLOG(x) is as follows.

The complex result is:

$r + is$

where:

$r = \log_e((u^2 + v^2)^{1/2})$

$s = \arctan(v/u)$

The real-valued log and arctangent functions are evaluated as described for the functions ALOG and ATAN2. The square root is computed by the machine instruction SQRT.

The message:

ZERO ARGUMENT

is issued if the argument is 0.+i0., the result is set to indefinite, and a normal exit is taken from CLOG.

CMPLX(a_1,a_2)

This constructs a complex number from two real numbers. CMPLX(x,y) assigns x to the real part of the result and y to the imaginary part of the result.

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 the 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)

See SIN for a description of the COS function.

COSH(a)

This 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. For a given real x:

$$\cosh(x) = (e^x + e^{-x})/2.0 .$$

If an unacceptable argument is received, the message:

ARGUMENT TOO LARGE

is issued, the result is set to indefinite, and a normal exit is taken from COSH.

COTAN(a)

This 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.

For a given real number x, cotan(x) is calculated as follows.

Let:

sign = sign (x)

r = x * 4 /pi;

n = [r];

z = r - n (where $z \geq 0$ and < 1);

s = n modulo 8;

k = s if $0 \leq s \leq 3$ and k = (s - 4) if $4 \leq s \leq 7$; and

if k = 1 or k = 3 then z = z - 1

if k = 1 or k = 2 then sign = - sign

z = sign * z

The values of cotan(x) corresponding to the values of k are:

k	cotan(x)
0	1/tan(z)
1	tan(z)
2	1/tan(z)
3	1/tan(z)

In any case, tan(y) is approximated by:

$$\tan(y) = y * \sum_{n=0}^{12} c_n y^{2n},$$

where c_n are constants as defined for the TAN function.

If an unacceptable argument is received, a data flag branch occurs.

CSIN(a)

This computes the sine of a complex number. The result is a complex number accurate to approximately 45 bits.

For a given complex-valued x = u + iv, CSIN(x) is computed as follows.

If abs(u) $> .110534964875444 * 10^{15}$ or v > 19905.80, the result is set to indefinite, the appropriate error message is issued, and a normal exit is taken from CSIN.

Otherwise, the complex result is:

r + is

where:

$r = \sin(u) * (e^v + e^{-v})/2$

$s = \cos(u) * (e^v - e^{-v})/2$ for $|v| \geq 0.5$

$$s = \cos(u) * v * \sum_{n=0}^{5} c_n v^{2n} \text{ for } |v| < 0.5$$

The values for c_n are as given in the CCOS routine.

Real-valued sine, cosine, and exponential functions are evaluated as described for the functions SIN, COS, and EXP respectively.

If an unacceptable argument is received, one of the messages:

ABS (REAL PART) TOO LARGE

IMAG. PART TOO LARGE

is issued, both real and imaginary parts of the result are set to indefinite, and a normal exit is taken from CSIN.

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.

For a given:

x = u + iv

taking the square root of x produces the result:

r + is

where r and s have one of the following sets of values:

r = b and s = c * sign(v) if $u \geq 0$

r = c and s = b * sign(v) if $u < 0$

r = 0 and s = 0 if u = 0 and v = 0

The values of b and c are defined as follows:

$b = ((a + |u|)/2)^{1/2}$

c = |v| /(2 * b)

where:

a = (u2 + v2)1/2

The square root function is computed by means of the machine instruction SQRT.

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.

For a given double-precision number x:

DASIN(x) = DATAN(y)

where:

$y = x/(1.0 - x^2)^{1/2}$ for $|x|<1.0$

arcsin(x) = pi/2.0 for x = 1.0

arcsin(x) = - pi/2.0 for x = -1.0

Also:

DACOS(x) = pi/2.0 - DATAN(y) for $|x|<1.0$

arccos(x) = 0.0 for x = 1.0

arccos(x) = pi for x = -1.0

(DATAN is the function that computes the arctangent of a double-precision number.)

If an unacceptable argument is received, the message:

ARGUMENT .GT. ONE

is issued, the result is set to indefinite, and a normal exit is taken from DASIN or DACOS.

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.

For two double-precision numbers x and y, the result is in the range:

-pi to -pi/2 if $x<0$, $y\leq 0$

-pi/2 to 0 if $x\geq 0$, $y<0$

0 to pi/2 if $x\geq 0$, $y\geq 0$

pi/2 to pi if $x<0$, $y>0$

Valid arguments for DATAN and DATAN2 lie in the interval $-0.47685405771593E+8645\leq x\leq +0.47685405771593E+8645$ (the largest allowable argument value is half of the largest allowable real number).

The arctan(x/y) is calculated as follows:

atan(x/y) = sign(x) * pi/2 if y = 0

atan(x/y) = atan(z) * sign(x) if $y>0$

atan(x/y) = pi - atan(z) if $y<0$ and $x\geq 0$

atan(x/y) = atan(z) - pi if $y<0$ and $x<0$

where:

$z = |x/y|$

and atan(z) is calculated as follows:

atan(z) = atan(v) if $|x|\leq|y|$

atan(z) = pi/2 - atan(v) if $|y|<|x|$

where:

$v = |t_1|/|t_2|$

with t_1 and t_2 being the two double-precision arguments and:

$|t_1|\leq|t_2|$

where atan(v) is calculated as follows:

atan(v) = atan(r) + c

where atan(r) is computed from a telescoped Taylor-Maclauren power series, and where r and c are as defined for the function ATAN.

If unacceptable arguments are received, the message:

X=Y=0.0

is issued, the result is set to indefinite, and a normal exit is taken from DATAN or DATAN2.

DATAN2(a,b)

See DATAN for a description of the DATAN2 function.

DATE(d)

This queries the system as to the date, and returns a result of type CHARACTER *8 in the following format:

mm/dd/yy

mm Pair of decimal digits expressing the month.

dd Pair of decimal digits expressing the day.

yy Pair of decimal digits expressing the year.

Within any particular routine, DATE must be consistently called as a subroutine or a function. For a function call, the argument is ignored.

DBLE(a)

This converts a real number to double-precision. The value of DBLE(x) is the same as the value of x. No error messages are issued by DBLE.

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.

For a given double-precision x:

$$\cosh(x) = (e^{x} - e^{-x})/2.0$$

If an unacceptable argument is received, the message:

ARGUMENT TOO LARGE

is issued, the result is set to indefinite, and a normal exit is taken from DCOSH.

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.

For a given x:

$$e^{x} = 2^{n} * e^{r1} * e^{r2}$$

where:

$$n = [x/\log_e(2) + .5]$$

$$r = r1 + r2 = x - n * \log_e(2)$$

r1 is the most significant part of r and r2 is the least significant part of r; and the

$$|r| \leq \log_e(2)$$

The factor e^{r1} is evaluated from a polynomial of degree 17. The polynomial was telescoped from a truncated Taylor-Maclauren power series.

The factor $e^{r2} = 1 + r2$.

If the function argument is out of range, the message:

ARGUMENT TOO LARGE, FLOATING POINT OVERFLOW

is issued, the result is set to indefinite, and a normal exit is taken from DEXP.

DFLOAT(a)

This converts an integer number to a double-precision number. The normalized integer number is the first word of the double-precision result, and the second word is set to real zero. The result is accurate to 94 bits.

DIM(a_1,a_2)

This computes the positive excess of one real number over another real number. DIM(x,y) returns the value x-y if x is greater than or equal to y, and returns a value of 0.0 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.

For a given double-precision number:

$$x = 2p * w$$

where:

$$(1/2)1/2 \leq w < 21/2$$

and p is an integer:

$$\log_e(x) = p * \log_e(2) + \log_e(w)$$

The term:

$$\log_e(w)$$

is initially approximated by:

$$a_0 = c_1 * v + c_3 * v^3 + c_5 * v^5 + c_7 * v^7$$

where:

$$v = (w - 1)(w + 1)$$

and c_n are as for the function ALOG.

An iteration must be performed to obtain accuracy. The iteration formula for:

$$f(a) = e^{a} - x = 0$$

is:

$$a_{n+1} = a_n - t$$

where:

$$r = x * e^{-a_n}$$

$$t = 1 - r$$

The final result with desired accuracy is:

$$a_2 = a_0 - t1 - t2 - t12 * (1/2 + (t1) / 2)$$

where t1 and r1 denote the most significant parts of t and r, while t2 and r2 denote the least significant parts of t and r.

If a zero or negative argument is received, one of the messages:

ZERO ARGUMENT

NEGATIVE ARGUMENT

is issued, the result is set to indefinite, and a normal exit is taken from DLOG.

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.

For a given double-precision number x greater than zero:

$$\log_{10}(x) = \log_{10}(e) * \log_e(x)$$

where the natural logarithm is computed as described for the function DLOG.

If a zero or negative argument is received, one of the messages:

ZERO ARGUMENT

NEGATIVE ARGUMENT

is issued, the result is set to indefinite, and a normal exit is taken from DLOG10.

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. The result is double-precision. Valid arguments for DMOD lie in the interval $-0.47685405771593E+8645 \leq x \leq +0.47685405771593E+8645$ (the largest allowable argument value is half of the largest allowable real number).

For given double-precision numbers x and y:

$$DMOD(x,y) = x - y * [x/y]$$

DPROD(a_1,a_2)

This computes the double-precision product of two real numbers. Valid arguments for DPROD lie in the interval $-0.47685405771593E+8645 \leq x \leq +0.47685405771593E+8645$ (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) returns one of the values $-|x|$, 0, or $|x|$ according as y is negative, zero, or positive, respectively.

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.

For a given double-precision x, the sine and cosine of x are computed as follows:

$$\cos(x) = \cos(r) * \cos(k * pi/2) \pm \sin(r) * \sin(k * pi/2)$$

$$\sin(x) = \cos(x - pi/2)$$

where:

$$n = [|x| * 2/pi + .5]$$

$$r = |(|x| - n * pi/2)|\ , r \leq pi/4$$

$$k = n \text{ modulo } 4\ , 0 \leq k \leq 3$$

Depending on k and on the sign of $|x|-n*pi/2$, cos(x) is equal to plus or minus the sin(r) or cos(r). Accordingly, sin(r) or cos(r) is evaluated and negated if necessary. The sin(r) and cos(r) are evaluated by polynomials of degree 21 and 20, respectively. These polynomials were telescoped from truncated Taylor-Maclauren power series of degree 25 and 24.

If an unacceptable argument is received, the message:

ARGUMENT TOO LARGE

is issued, the result is set to indefinite, and a normal exit is taken from DSIN or DCOS.

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.

For any given double-precision number x:

$$\sinh(x) = (e^x - e^{-x})/2.0$$

If an unacceptable argument is received, the message:

ARGUMENT TOO LARGE

is issued, the result is set to indefinite, and a normal exit is taken from DSINH.

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. An approximation to the square root is obtained by using the SQRT machine instruction; this number is accurate to 14 decimal places. One Newton approximation is done to double the accuracy of the number; the form is:

$$a2 = 1/2 * (a1 + x/a1)$$

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. Valid arguments for the DTAN function are in the interval -.110534964875444D+15<x<.110534964875444D+15.

For a given double-precision x, tan(x) = sin(x)/cos(x), where:

$$\sin(x) = \pm \sin(r) * \cos(k*pi/2) + \cos(r) * \sin(k*pi/2)$$

$$\cos(x) = \cos(r) * \cos(k*pi/2) \pm \sin(r) * \sin(k*pi/2)$$

$$r = |x-n*pi/2|$$

$$n = \text{floor}(x*2/pi+.5)$$

$$k = n \text{ modulo } 4,\ 0 \leq n \leq 3$$

Depending on k and on the sign of (x-n*pi/2), tan(x) is equal to plus or minus sin(r)/cos(r) or cos(r)/sin(r).

If an unacceptable argument is received, the message:

ARGUMENT TOO LARGE

is issued, the result is set to indefinite, and a normal exit is taken from DTAN.

DTANH(a)

This computes the hyperbolic tangent of a double-precision number and returns a double-precision result that is accurate to 90 bits.

For a given double-precision x:

$$\tanh(x) = (e^x - e^{-x})/(e^x + e^{-x}) \text{ for } |x|<18.0$$

$$\tanh(x) = 1.0 \text{ for } x = 18.0$$

$$\tanh(x) = -1.0 \text{ for } x = -18.0$$

EXP(a)

This computes the exponential of a real number. The result, accurate to approximately 45 bits, is a real number greater than or equal to zero.

For a given x, the mathematical method used for calculating EXP(x) is as follows. If x<-19842.031, the result is zero. If x >19905.7999999999, a data flag branch occurs inside the routine.

For all other values of x:

$$e^x = 2^k * 2^{m/16} * 2^{f/16}$$

where:

$$n = [16 * (x/\log_e(2))]$$

$$k = [n/16] \text{ if } x \geq 0$$

$$k = [n/16] - 1 \text{ if } x < 0$$

$$m = n \text{ modulo } 16 \text{ if } x \geq 0$$

$$m = 16 - (n \text{ modulo } 16) \text{ if } x < 0$$

$$f = (16 * (x/\log_e(2))) - n$$

The absolute value of f is ≥ 0 and <1.

The factor:

$$2^{m/16}$$

is obtained from a table.

The product:

$$2^k * 2^{m/16}$$

is obtained by adding the exponents.

The factor:

$$2^{f/16} = (q + f * p)/(q - f * p)$$

where:

$$q = q01 * f^2 + q00$$

$$p = p01 * f^2 + p00$$

$$q00 = .532832542630989 * 10^4$$

$$q01 = .1 * 10^1$$

$$p00 = .115416054573517 * 10^3$$

$$p01 = .361007098948762 * 10^{-2}$$

FLOAT(a)

This converts an integer number to a real number by normalizing the integer number.

IABS(a)

For an integer number x, IABS(x) computes the absolute value |x|.

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|.

IFIX(a)

This converts a real number to an integer number. IFIX, which is an alternative name for INT, computes the largest integer less than or equal to its real argument, retaining the sign of the argument.

INT(a)

For a real number x, INT(x) computes $[x]$, where $[A]$ is the sign of A times the largest integer less than or equal to $|A|$.

ISIGN(a_1,a_2)

This combines the absolute value of one integer number with the sign of another integer number; ISIGN(x,y) returns one of the values $-|x|$, 0, or $|x|$ according as y is negative, zero, or positive, respectively.

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.

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 computes one integer number modulo a second integer number and produces an integer result. MOD(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 $|A|$.

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.

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

Q8SEQ(v_1,v_2)

From among the pairs of corresponding elements in two real or two 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 is 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.

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 real, integer, or complex 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 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, could 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 could 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

Q8SMAXI(v) OR Q8SMAXI(v,c)

Like Q8SMAX, this finds the maximum from among the elements in a 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 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, could 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 could 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.

Q8SNE(v_1,v_2)

This is identical to Q8SEQ, except that Q8SNE searches for an element in x that is not equal to the corresponding element in y.

Q8SPROD(v) OR Q8SPROD(v,c)

This calculates the product of the elements in a 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, could 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 could 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

Q8SSUM(v) OR Q8SSUM(v,c)

This sums the elements in a 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, could 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 could 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

Q8VADJM(v;u)

This computes the averages of adjacent elements of the real input vector. For a given real vector x, Q8VADJM(x;r) forms the n^{th} element of the result vector r by adding the n^{th} 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;r) is:

r = 4. 4. 4. 4. 4.5 4.5 4.

Q8VAVG(v_1,v_2;u)

This computes the averages of corresponding elements of two 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;r) is the vector:

r = 5.15 5.7 9.5 4.955 0.55

Q8VAVGD(v_1,v_2;u)

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^{th} 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;r) is the vector:

r = 48. 45.5 45.5 42.5 43.

Q8VCMPRS(v,c;u)

This deletes selected elements from a 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;r) is the vector:

r = 5 5

Q8VCTRL(v,c;u)

This changes the values of only selected elements in a real or integer result vector, by 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;y) is the vector:

y = 9 9 9 9 9

Q8VDELT(v;u)

This computes the differences between the adjacent elements of the input vector. For a given real vector x, Q8VDELT(x;r) computes the n^{th} element of the result vector r by subtracting the n^{th} 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;r) is:

r = -2. 2. -2. 2. -1. 1. -2.

Q8VEQI(v_1,v_2;u)

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 real scalar. For given real vectors x and y, Q8VEQI(x,y;r) 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 n^{th} iteration, which is performed using the n^{th} element of x and which is a count of the number of unsuccessful compares that were made on this iteration, is placed in the n^{th} element of r.

Example:

Given:

x = 0. 1. 4. 5. 4.

y = -1. 0. 3. 5. 4.

the value of Q8VEQI(x,y;r) is the vector:

r = 1 5 4 3 4

Q8VGATHP(v,i,n;r)

This creates a real or integer vector, by using the elements in another vector of the same data type to provide the values. Selection of values is controlled by the second parameter, i, an integer scalar constant stride.

For a real or integer vector x and a constant stride i, the result vector r consists of every ith element of x, starting with the first. The lengths of x and r are ignored; the number of items gathered is controlled by the third parameter, n.

Example:

Given:

x = 10 19 11 15 0 9 2 5

i = 2

n = 4

the value of Q8VGATHP(x,i,n;r) is the vector:

r = 10 11 0 2

Q8VGATHR(v,i;u)

This creates a real or integer vector, by 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;r) is the vector:

r = 3 9 0 9 11 10 10

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.

Q8VINTL(a_1,a_2;u)

This forms a 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;r) 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;r) 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;u)

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;u)

Q8VMASK(x,y,c;r) 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 in 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;r) is:

r = 1 2 19 1 2 19 1 2 19

Q8VMERG(v_1,v_2,c;u)

This merges the elements in two real or two integer vectors, under control of a bit control vector, into a single result vector. Q8VMERG(x,y,c;r) 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 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;r) is the vector:

r = 10 11 5 4 12

Q8VMKO(a_1,a_2;u)

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;r) is the bit vector:

r = 1110001110

Q8VMKZ(a_1,a_2;u)

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;r) 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;r) is the bit vector:

r = 0000000111

Q8VNEI(v_1,v_2;u)

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;u)

This computes a polynomial at several values. For given real vectors x and y, Q8VPOLY(x,y;r) 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;r) is the vector:

r = 36 9 15 48 101

The elements of r are computed as follows:

$r(1) = 10(-2^2) + 3(-2) + 2 = 36$

$r(2) = 10(-1^2) + 3(-1) + 2 = 9$

$r(3) = 10(1^2) + 3(1) + 2 = 15$

$r(4) = 10(2^2) + 2(2) + 2 = 48$

$r(5) = 10(3^2) + 3(3) + 2 = 101$

Q8VPOLY is not valid on the CYBER 205. When control statement parameters are used to select the CYBER 205 as the target machine for compilation, an error message is issued. If instruction scheduling was selected, it is terminated.

Q8VREV(v;u)

This reverses the order of the elements in a 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;r) is the vector:

r = 10 9 6 5 3 4

Q8VSCATP(v,i,n;r)

This changes the values of only selected elements in a real or integer result vector, by using the elements in another vector or a scalar of the same type to provide the new values. Selection of elements to be altered in the result vector is controlled by the second parameter, i, an integer scalar constant stride.

For a real or integer vector x and a constant stride i, every ith element, starting with the first, of the result vector r is altered. The length of r is ignored; the number of items scattered is controlled by the third parameter, n.

Example:

Given:

x = 0 50 -1 60 70

i = 2

n = 5

r = 9 9 9 9 9 9 9 9 9 9

the value of Q8VSCATP(x,i,n;r) is the vector:

r = 0 9 50 9 -1 9 60 9 70 9

Q8VSCATR(v,i;u)

This changes the values of only selected elements in a real or integer result vector, by 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 a single position in y, and not every element in y needs 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):

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

Q8VXPND(v,c;u)

This inserts additional elements having the value 0 (or 0.0) into a real or integer vector, under control of a bit control vector. The effect of the procedure is as though a Q8VMERG(x,n,c;y) 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;r) is the vector:

r = 0 5 5 0 0 0

RANF(d)

This returns a random number. The argument is ignored. 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'00004C65DA2C866D'. The seed can be obtained and reset with the subroutines RANGET and RANSET, respectively. The default value of the seed is X'000054F4A3B933BD'. A vector of random numbers can be returned with the subroutine VRANF.

REAL(a)

This returns the real part of a complex number as a real number; if x+iy is the complex number, REAL returns x.

SECOND(d)

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. Within any particular routine, SECOND must be consistently called either as a subroutine or a function. For a function call, the argument is ignored.

Because SECOND uses the job interval timer (JIT), a user program that manipulates the job interval timer invalidates the returned result. Furthermore, a user program that attempts to perform JIT interrupt processing conflicts with SECOND and causes the program to abort. See the description of the data flag branch manager.

SIGN(a_1,a_2)

This combines the absolute value of one real number with the sign of another real number; SIGN(x,y) returns one of the values -|x|, 0 or |x| according as y is negative, zero, or positive, respectively.

SIN(a) AND COS(a)

These compute the sine and cosine of a real number expressed in radians. The real number modulo 2 pi is used by the functions. The results are real numbers in the range -1 to 1, inclusive, and are accurate to approximately 45 bits.

For a given x, sin(x) and cos(x) are calculated as follows.

If $|x| > .110534964875444 * 10^{15}$, a data flag branch abort occurs in SIN or COS. Otherwise, the sine and cosine of x are calculated identically, differing only in the formula with which the value of k is selected:

$k = r_2$ modulo 4 for the sine or cosine of x

where:

$r_1 = |x| * 4/pi$

$r_2 = [r_1]$

The values of f(x) corresponding to the values of k are:

k	sin(x)	cos(x)
0	sin(z)	sin(1-z)
1	sin(1-z)	-sin(z)
2	-sin(z)	-sin(1-z)
3	-sin(1-z)	sin(z)

where $z = r_1 - r_2$, and $0 \leq z < 1$.

The sin(z) is approximated by the formula:

$$\sin(z) = z \sum_{n=0}^{8} s_n z^{2n}$$

where s_n has the following values:

$s_0 = .157079632679491 * 10^1$

$s_1 = -.645964097506246 * 10^0$

$s_2 = .796926262461656 * 10^1$

$s_3 = -.468175413530426 * 10^{-2}$

$s_4 = .160441184713148 * 10^{-3}$

$s_5 = -.35988432058822 * 10^{-5}$

$s_6 = .569213644231555 * 10^{-7}$

$s_7 = .668441770083272 * 10^{-9}$

$s_8 = .587299730858022 * 10^{-11}$

SINH(a)

This computes the hyperbolic sine of a real number and produces a real result that is accurate to 47 bits. For a given real number x:

$\sinh(x) = (e^x - e^{-x})/2.0$ for $|x| \geq 0.5$, and

$$\sinh(x) = \sum_{n=0}^{5} x^{(2n+1)}/(2n+1)! \text{ for } |x| < 0.5.$$

If an unacceptable argument is received, the message:

ARGUMENT TOO LARGE

is issued, the result is set to indefinite, and a normal exit is taken from SINH.

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.

SQRT(a)

This computes the square root of a real number and returns a real result that is accurate to approximately 45 bits. The square root function is computed by means of the machine instruction SQRT.

TAN(a)

This 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

$-0.276334121886E+14 < x < +0.276334121886E+14$

Note that:

$(2^{46}-1) * pi/8 = 0.276334121886E+14$

For a given real number x, TAN(x) is calculated as follows.

Let:

sign = sign(x)

$r = |x * 4/pi|$

$n = [r]$

$z = r - n$ (where $z \geq 0$ and < 1)

s = n modulo 8

k = s if $0 \leq s \leq 3$ and k = (s-4) if $4 \leq s \leq 7$

if k = 1 or k = 3, then z = z-1

if k = 1 or k = 2, then sign = -sign

z = sign * z

Then, the values of tan(x) corresponding to the values of k are:

k	tan(x)
0	tan(z)
1	1/tan(z)
2	1/tan(z)
3	tan(z)

In any case, tan(y) is approximated by:

$$\tan(y) = y * \sum_{n=0}^{12} c_n y^{2n},$$

where:

$c_0 = .785398163397450 * 10^0$

$c_1 = .161491024375903 * 10^0$

$c_2 = .398463131591972 * 10^{-1}$

$c_3 = .994872676029386 * 10^{-2}$

$c_4 = .248684703853100 * 10^{-2}$

$c_5 = .621642731342203 * 10^{-3}$

$c_6 = .155667818044739 * 10^{-3}$

$c_7 = .381715991884668 * 10^{-4}$

$c_8 = .110209742715057 * 10^{-4}$

$c_9 = .742132251112162 * 10^{-6}$

$c_{10} = .203719911897978 * 10^{-5}$

$c_{11} = -.589759669894803 * 10^{-6}$

$c_{12} = .232574075395844 * 10^{-6}$

If an unacceptable argument is received, a data flag branch occurs.

TANH(a)

This 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.

For a given real number x:

$$\tanh(x) = x * \sum_{n=0}^{5} c_n x^{2n} \text{ for } 0 \leq |x| \leq 0.12$$

where:

$c_0 = 1$

$c_1 = -1/3$

$c_2 = 2/15$

$c_3 = -17/315$

$c_4 = 62/2835$

$c_5 = -1382/155925$

The hyperbolic tangent is:

$\tanh(x) = (e^x - e^{-x})/(e^x + e^{-x}) = 1 - (2/(e^{2x} + 1))$
for $0.12 < |x| \leq 18.0$

where the exponential function is computed as described for the EXP function.

The hyperbolic tangent is:

$\tanh(x) = \text{sign}(x) * 1.0$ for $|x| > 18.0$

TIME(d)

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.

Within any particular routine, TIME must be consistently called either as a subroutine or a function. For a function call, the argument is ignored.

VABS(v;u)

For each element x in a real vector, VABS computes the absolute value x. The real result is accurate to 47 bits.

VACOS(v;u)

This computes the arccosine of each element in a real vector. The result real vector contains elements that are accurate to approximately 45 bits.

For a given real element x:

$\text{acos}(x) = \text{atan}((1 - x^2)^{1/2}/x)$

that is:

VACOS(x;y) = VATAN2(a,b;r)

where:

$a = (1 - x^2)1/2$

$b = x$

and VATAN2 is the function that computes the arctangent of ratios of elements in real vectors.

VAIMAG(v;u)

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;u)

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;u)

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;u)

This computes the logarithm of each element in a real vector, returning a result vector of real numbers accurate to approximately 45 bits.

For a real element x:

$$\log_{10}(x) = \log_{10}(e) * \log_e(x),$$

where the logarithm of x is computed as described for the function ALOG.

VAMOD(v_1,v_2;u)

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 $|A|$.

VASIN(v;u)

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}.

For a given real element x:

$$\text{asin}(x) = \text{atan}(x/(1 - x^2)^{1/2})$$

that is:

VASIN(x;y) = VATAN2(a,b;r)

where:

a = x and

$b = (1 - x^2)^{1/2}$

and VATAN2 is the function that computes the arctangent of ratios of elements in real vectors.

VATAN(v;u)

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}.

For a given real element x:

if $0 \leq x < 1$, then atan(x) = atan(z)

if $x > 1$, then atan(x) = pi/2 - atan(z)

if $x < 0$, then atan(x) = - atan(-x)

The arctangent is:

$$\text{atan}(z) = \text{atan}(z1) + \text{atan}(z2/(1 + z1^2 + z1 * z2))$$

where:

z = x if x<1

z = 1/x if x>1

The atan(z1) is obtained from a table lookup, and the arctangent of the second term is computed by a polynomial. The values z1 and z2 are the most significant and the least significant parts of z, respectively.

VATAN2(v_1,v_2;u)

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.

For a given pair of elements (x,y), the arctangent of x/y is calculated as follows:

atan(x/y) = sign(x) * atan(v) if y>0

atan (x/y) = pi - atan(v) if y<0 and x>0

atan(x/y) = atan(v) - pi if y<0 and x<0

where:

v = |x/y|

and atan(v) is calculated as follows:

atan(v) = atan(|x/y|) if $|x| \leq |y|$

atan(v) = pi/2 - atan(|y/x|) if |x|>|y|

$$\text{atan}(z) = \text{atan}(z1) + \text{atan}(z2/(1 + z1^2 + z1 * z2))$$

where z=|x/y| for $|x| \leq |y|$ and z=|y/x| for |x|>|y|; and z1 and z2 are the most and least significant parts of z, respectively.

VCABS(v;u)

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.

For a given complex element:

x = u + iv

the result is:

$(u^2 + v^2)^{1/2}$

where the square root function is evaluated by the machine instruction SQRT.

VCCOS(v;u)

See VCSIN for a description of the VCCOS function.

VCEXP(v;u)

This computes the exponential of each element in a complex vector, and produces a complex vector of results. For a given vector element:

$x = u + iv$

$e^x = e^u * \cos(v) + i * e^u * \sin(v)$

where the sine, cosine, and exponential are obtained from calls to the VSIN, VCOS, and VEXP routines, respectively.

VCLOG(v;u)

This computes the natural logarithm of each element in a complex vector, returning a complex result vector.

For a complex element:

$x = u + iv$

$\log_e(x) = \log_e(r) + i * a$

where:

$r = |x|$

$a = \arctan(u/v)$

The arctangent, modulus, and logarithm are computed as for the functions VATAN, CABS, and VALOG, respectively.

VCMPLX(v_1,v_2;u)

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;u)

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;u)

See VSIN for a description of the VCOS function.

VCSIN(v;u) AND VCCOS(v;u)

These compute the sine and cosine of each element in a complex vector. Each complex result is accurate to approximately 45 bits.

For a complex element $x = u + iv$:

$\sin(x) = \sin(u) * \cosh(v) + i * \cos(u) * \sinh(v)$

$\cos(x) = \cos(u) * \cosh(v) - i * \sin(u) * \sinh(v)$

where:

$\cosh(u) = (e^u + e^{-u}) * 0.5$

$\sinh(u) = (e^u - e^{-u}) * 0.5$

$e^{-u} = 1/e^u$

The real-valued sine, cosine, and exponential are computed by calls to VSIN, VCOS, and VEXP, respectively.

VCSQRT(v;u)

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;u)

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;u)

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;u)

This computes the exponential of each element in a real vector. VEXP returns a result vector of real numbers.

See EXP for a description of exponential function.

VFLOAT(v;u)

This constructs a real vector from an integer vector. Each integer vector element is normalized and assigned to the real vector.

VIABS(v;u)

For each element x in an integer vector, VIABS computes the absolute value |x|.

VIDIM(v_1,v_2;u)

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;u)

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|.

VINT(v;u)

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;u)

For each pair (x,y) of corresponding elements in two integer vectors, this combines the sign of y with the absolute value of x; the effect of VISIGN on each pair (x,y) is that of the expression ISIGN(x,y).

VMOD(v_1,v_2;u)

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|.

VREAL(v;u)

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;u)

For each pair (x,y) of corresponding elements in two real vectors, this combines the sign of y with the absolute value of x; 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;u) AND VCOS(v;u)

These compute the sine and cosine of each element in a real vector as described for the function SIN.

VSNGL(v;u)

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;u)

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 with the machine instruction VSQRT.

VTAN(v;u)

This computes the tangent of each element in a real vector. See TAN discussion for a description of the tangent function.

The system control statements accompanying a CYBER 200 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 CYBER 200 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_1 — 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 ABCEIKLMOR SUVYZ12, 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, and a period instead of 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) do not have to 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 object file is not built when the O=olist parameter without the B option 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. This optimization involves compile-time computable result propagation, redundant code elimination, and instruction scheduling. The E option is included in the O option. The E option effectively selects options P, 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 fullword (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 CYBER 200 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, I, and P.

P - PROPAGATION

The P option selects compile-time-computable result propagation.

R - REDUNDANT CODE ELIMINATION

The R option selects elimination of redundant code. The R option is included in the O and E options.

S - SUPPRESS DEBUG SYMBOL TABLE CREATION

The effect of this option is to suppress generation in the binary output of 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 and for a FORTRAN run-time routine to identify the source line in a user routine at which a run-time error occurred. The user must not 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 can be selected.

U - UNSAFE 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, the compiler cannot determine the number of iterations of the loop. Vectorization of such loops is considered unsafe because the loop count might exceed 65535, which is the maximum length of a vector. If a DO loop contains an assignment statement that has an equivalenced data element on the left side, the loop can be vectorized only if the U compile option is selected.

V - VECTORIZATION AND AUTOMATIC RECOGNITION OF STACKLIB LOOPS

Vectorization of certain CYBER 200 FORTRAN language constructs and automatic recognition and conversion of certain DO loops into calls to a stacklib routine are requested with the V compile option. The language constructs that fall under these categories are described in section 11.

Y - SYNTAX CHECK

A partial compilation can be performed to check the syntax of a FORTRAN program and any resulting diagnostics can be produced by selecting the Y compile option. The Y option can appear alone or with the L or S options (such as LY or SY); all other option combinations using Y 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 and 3 options. When 1, 2, or 3 is not selected, optimization is for the mainframe on which compilation is performed.

2 - CYBER 203 OPTIMIZATION

The 2 option selects optimization for the CYBER 203. The 2 option conflicts with the 1 and 3 options. When 1, 2, or 3 is not selected, optimization is for the mainframe on which compilation is performed.

3 - CYBER 205 OPTIMIZATION

The 3 option selects optimization for the CYBER 205. The 3 option conflicts with the 1 and 2 options. When 1, 2, or 3 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 source lines are numbered on the left. The source line 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, storage map, and register map appear on the output file following the source listing is:

- Cross-reference map
- Assembly listing
- Storage map and register map

Any diagnostics follow the storage and register maps.

CROSS-REFERENCE MAPS

When the C compile option is selected, from one to four cross-reference maps appear in the output for the program compilation. These maps appear immediately following the source program listing, or, when the L compile option is also selected, as the first listings in the output. The four cross-reference maps are:

- Statement label map
- Variable map
- Symbolic constant map
- Procedure map

The statement label map provides information about each statement label used in the program. See figure 15-1 for the format of the statement label map. If no statement labels are used in the program, the statement label map is not printed. Some uses of the statement label map include:

- Identifying unreferenced FORMAT statements and other unreferenced labeled statements
- Verifying that proper statement labels are specified in flow control statements
- Locating labeled statements in the program, and locating all references to a statement label

See figure 15-2 for an example of a statement label map.

```
STATEMENT LABEL MAP
--LABEL---DEFINED---REFERENCES

    lbl       def       refs
     .         .         .
     .         .         .
     .         .         .
```

lbl — A statement label that appears in the label field of a FORTRAN statement.

def — The source line number of the statement in which lbl appears in the label field.

refs — The source line numbers of all source lines that contain references to lbl.

Figure 15-1. Statement Label Map Format

```
FORTRAN R1.5 CYCLE FL1      BUILT 06/19/80 11:44    SOURCE LISTING              COMPILED 09/09/80 11:56 O=CAM          PAGE  1
   00001          PROGRAM PASCAL(OUTPUT)                                                          0001/00001
   00002          INTEGER L(11),ONE,A,B                                                           0001/00002
   00003          PARAMETER(ONE=1)                                                                0001/00003
   00004          IADD(A,B) = A + B                                                               0001/00004
   00005          DATA L(11) /ONE/                                                                0001/00005
         C                                                                                        0001/00006
   00006          PRINT 4, (I,I=1,11)                                                             0001/00007
   00007    4     FORMAT('1COMBINATIONS OF M THINGS TAKEN N AT ',                                 0001/00008
                X    'A TIME.' //20X,'-N-' /1115)                                               0001/00009
   00008          DO 200 I=1,10                                                                   0001/00010
   00009          K = 11 - I                                                                      0001/00011
   00010          L(K) = 1                                                                        0001/00012
   00011          DO 100 J=K,10                                                                   0001/00013
   00012  100     L(J) = IADD(L(J),L(J+1))                                                        0001/00014
   00013  200     PRINT 3, (L(J),J=K,11)                                                          0001/00015
   00014    3     FORMAT(1115)                                                                    0001/00016
         C                                                                                        0001/00017
   00015          STOP                                                                            0001/00018
   00016          END                                                                             0001/00019
```

```
FORTRAN R1.5 CYCLE FL1      BUILT 06/19/80 11:44    CROSS REF LISTING    PASCAL    COMPILED 09/09/80 11:56 O=CAM          PAGE  2

STATEMENT LABEL MAP
--LABEL---DEFINED---REFERENCES

  100         12          11
  200         13           8
  3           14          13
  4            7           6

VARIABLE MAP
--NAME-------BLOCK------TYPE------CLASS-------REFERENCES     A=ARGLIST, C=CTRL OF DO, I=DATA INIT, R=READ, S=STORE, W=WRITE

  A                     INTEGER   UNKNOWN        2       4       4
  B                     INTEGER   UNKNOWN        2       4       4
  I                     INTEGER   SIMPLE         6/W     6/C     8/C     9
  J                     INTEGER   SIMPLE        11/C    12      12      12      12      12      13      13/C
  K                     INTEGER   SIMPLE         9/S    10      11      13
  L                     INTEGER   ARRAY          2       5/I    10/S    12/S    12      12      12      12      13/W
  PASCAL                          PROGRAM        1

SYMBOLIC CONSTANT MAP
--NAME-------TYPE------VALUE-----------------------------------REFERENCES     S=DEFINITION LINE

  ONE         INTEGER    1                                        2       3/S

PROCEDURE MAP
--NAME-------TYPE-------CLASS-------------REFERENCES     D=STMT FN DEF, A=ARGLIST

  IADD        INTEGER    STAT FUNC              4/D    12
```

Figure 15-2. Compiler Output Example (Sheet 1 of 5)

```
FORTRAN R1.5 CYCLE FL1      BUILT 06/19/80 11:44    ASSEMBLY LISTING     PASCAL     COMPILED 09/09/80 11:56 0=CAM          PAGE    3

    LOCATION        MACHINE          LINE    SOURCE          ASSEMBLY REPRESENTATION
    COUNTER         INSTRUCTION      NUMBER  LABEL

                                             PASCAL     IDENT
                                                        ENTRY       PASCAL
                                     00001   A00006
    0000000         7D00151C                PASCAL     SWAP        ,C_#1A,CUR_STACK
    0000020         781C001D                           RTOR        CUR_STACK,PREV_STACK
    0000040         7818001C                           RTOR        DYN_SPACE,CUR_STACK
    0000060         3F181400                           IS          DYN_SPACE,5120
    0000080         2A1C0050                           ELEN        CUR_STACK,80
    00000A0         3E230700                           ES          ST_23,1792
    00000C0         631E2324                           ADDX        CALLEDATA,ST_23,ST_24
    00000E0         2A24004A                           ELEN        ST_24,74
    0000100         7D241400                           SWAP        ST_24,C_#20
    0000120         78660003                           RTOR        L_C00001_DESCR,PR_3
    0000140         7861001E                           RTOR        FT_INIT_DB,CALLEDATA
    0000160         361A0060                           BSAVE       RETURN,FT_INIT_ADR
    0000180         7818004A                           RTOR        DYN_SPACE,PI_DYNSP
                                     00006   A00007
    00001AC         78670004                           RTOR        L_F4_DESCR,PR_4
    00001C0         785B001E                           RTOR        FT_WTIPR_DB,CALLEDATA
    00001E0         361A005A                           BSAVE       RETURN,FT_WTIPR_ADR
    0000200         78520057                           RTOR        C_#1,I
                                             A00008
    0000220         78570003                 D00002    RTOR        I,PR_3
    0000240         785D001E                           RTOR        FT_WTIE_DB,CALLEDATA
    0000260         361A005C                           BSAVE       RETURN,FT_WTIE_ADR
    0000280         B4065752 00035357                  IBXLE,BRB   I,C_#1,D00002,C_#B,I
                                             A00009
    00002C0         7859001E                           RTOR        FT_WTTPR_DB,CALLEDATA
    00002E0         361A0058                           BSAVE       RETURN,FT_WTTPR_ADR
                                     00008   A00010
    0000300         78520057                           RTOR        C_#1,I
                                             A00011
    0000320         67535755                 C0009     D00003      SUBX        C_#B,I,K
    0000340         7F655552                 00010                 STO         IL_18_DESCR,K1,C_#1
    0000360         78550056                 00011                 RTOR        K,J
    ●               ●                                              ●           ●
    ●               ●                                              ●           ●
    ●               ●                                              ●           ●
    0000540         785F001E                                       RTOR        FT_WTIEA_DB,CALLEDATA
    0000560         361A005E                                       BSAVE       RETURN,FT_WTIEA_ADR
    0000580         7859001E                                       RTOR        FT_WTTPR_DB,CALLEDATA
    00005A0         361A0058                                       BSAVE       RETURN,FT_WTTPR_ADR
                                             A00015
    00005C0         B4065752 00155457                              IBXLE,BRB   I,C_#1,D00003,C_#A,I
                                     00015   A00016
    0000600         3E030000                                       ES          PR_3,0
    0000620         7863001E                                       RTOR        FT_STOP_DB,CALLEDATA
    0000640         361A0062                                       BSAVE       RETURN,FT_STOP_ADR
                                                                   END
```

Figure 15-2. Compiler Output Example (Sheet 2 of 5)

```
FORTRAN 81.5 CYCLE FL1     BUILT 06/19/80 11:44    REGISTER MAP          PASCAL    COMPILED 09/09/80 11:56 O=CAM                PAGE    5

REG.                  REG.                  REG.                  REG.                  REG.
 NO      NAME          NO      NAME          NO      NAME          NO      NAME          NO      NAME

 00   0   (MACHINE ZERO)   33   ST_33         66   L_C00001_DESCR   99   FR_99         CC   FR_CC
 01   DATA_FLAG_RETURN     34   ST_34         67   L_F4_DESCR       9A   FR_9A         CD   FR_CD
 02   TM_INTERUPT_ENTRY    35   ST_35         68   L_F3_DESCR       9B   FR_9B         CE   FR_CE
 03   PR_3                 36   ST_36         69   FR_69            9C   FR_9C         CF   FR_CF
 04   PR_4                 37   ST_37         6A   FR_6A            9D   FR_9D         D0   FR_D0
 05   PR_5                 38   ST_38         6B   FR_6B            9E   FR_9E         D1   FR_D1
 06   PR_6                 39   ST_39         6C   FR_6C            9F   FR_9F         D2   FR_D2
 07   PR_7                 3A   ST_3A         6D   FR_6D            A0   FR_A0         D3   FR_D3
 08   PR_8                 3B   ST_3B         6E   FR_6E            A1   FR_A1         D4   FR_D4
 09   PR_9                 3C   ST_3C         6F   FR_6F            A2   FR_A2         D5   FR_D5
 0A   PR_A                 3D   ST_3D         70   FR_70            A3   FR_A3         D6   FR_D6
 0B   PR_B                 3E   ST_3E         71   FR_71            A4   FR_A4         D7   FR_D7
 0C   PR_C                 3F   ST_3F         72   FR_72            A5   FR_A5         D8   FR_D8
 0D   PR_D                 40   ST_40         73   FR_73            A6   FR_A6         D9   FR_D9
 0E   PR_E                 41   ST_41         74   FR_74            A7   FR_A7         DA   FR_DA
 0F   PR_F                 42   ST_42         75   FR_75            A8   FR_A8         DB   FR_DB
 10   PR_10                43   ST_43         76   FR_76            A9   FR_A9         DC   FR_DC
 11   PR_11                44   ST_44         77   FR_77            AA   FR_AA         DD   FR_DD
 12   PR_12                45   ST_45         78   FR_78            AB   FR_AB         DE   FR_DE
 13   PR_13                46   ST_46         79   FR_79            AC   FR_AC         DF   FR_DF
 14   C_#20                47   ST_47         7A   FR_7A            AD   FR_AD         E0   FR_E0
 15   C_#1A                48   ST_48         7B   FR_7B            AE   FR_AE         E1   FR_E1
 16   C_1                  49   ST_49         7C   FR_7C            AF   FR_AF         E2   FR_E2
 17   C_PARM_DESCR         4A   PI_DYNSP      7D   FR_7D            B0   FR_B0         E3   FR_E3
 18   F_RET1               4B   P_DYNBAS      7E   FR_7E            B1   FR_B1         E4   FR_E4
 19   F_RET2               4C   L_TARVEC      7F   FR_7F            B2   FR_B2         E5   FR_E5
 1A   RETURN               4D   LEN_TARG      80   FR_80            B3   FR_B3         E6   FR_E6
 1B   DYN_SPACE            4E   V_TEMP1       81   FR_81            B4   FR_B4         E7   FR_E7
 1C   CUR_STACK            4F   V_TEMP2       82   FR_82            B5   FR_B5         E8   FR_E8
 1D   PREV_STACK           50   V_TEMP3       83   FR_83            B6   FR_B6         E9   FR_E9
 1E   CALLEDATA            51   V_TEMP4       84   FR_84            B7   FR_B7         EA   FR_EA
 1F   ON_UNIT              52   C_#1          85   FR_85            B8   FR_B8         EB   FR_EB
 20   DATABASE             53   C_#8          86   FR_86            B9   FR_B9         EC   FR_EC
 21   PARM_DESCR           54   C_#A          87   FR_87            BA   FR_BA         ED   FR_ED
 22   ST_22                55   K             88   FR_88            BB   FR_BB         EE   FR_EE
 23   ST_23                56   J             89   FR_89            BC   FR_BC         EF   FR_EF
 24   ST_24                57   I             8A   FR_8A            BD   FR_BD         F0   FR_F0
 25   ST_25                58   FT_WTTPR_ADR  8B   FR_8B            BE   FR_BE         F1   FR_F1
 26   ST_26                59   FT_WTTPR_DB   8C   FR_8C            BF   FR_BF         F2   FR_F2
 27   ST_27                5A   FT_WTIPR_ADR  8D   FR_8D            C0   FR_C0         F3   FR_F3
 28   ST_28                5B   FT_WTIPR_DB   8E   FR_8E            C1   FR_C1         F4   FR_F4
 29   ST_29                5C   FT_WTIE_ADR   8F   FR_8F            C2   FR_C2         F5   FR_F5
 2A   ST_2A                5D   FT_WTIE_DB    90   FR_90            C3   FR_C3         F6   FR_F6
 2B   ST_2B                5E   FT_WTIEA_ADR  91   FR_91            C4   FR_C4         F7   FR_F7
 2C   ST_2C                5F   FT_WTIEA_DB   92   FR_92            C5   FR_C5         F8   FR_F8
 2D   ST_2D                60   FT_INIT_ADR   93   FR_93            C6   FR_C6         F9   FR_F9
 2E   ST_2E                61   FT_INIT_DB    94   FR_94            C7   FR_C7         FA   FR_FA
 2F   ST_2F                62   FT_STOP_ADR   95   FR_95            C8   FR_C8         FB   FR_FB
 30   ST_30                63   FT_STOP_DB    96   FR_96            C9   FR_C9         FC   FR_FC
 31   ST_31                64   L_20_DESCR    97   FR_97            CA   FR_CA         FD   FR_FD
 32   ST_32                65   L_18_DESCR    98   FR_98            CB   FR_CB         FE   FR_FE
                                                                                       FF   FR_FF
```

Figure 15-2. Compiler Output Example (Sheet 3 of 5)

```
FORTRAN R1.5 CYCLE FL1      BUILT 06/19/80 11:44    STORAGE MAP          PASCAL    COMPILED 09/09/80 11:56 0=CAM              PAGE    6

                              PROGRAM NAME IS PASCAL      TOTAL LENGTH IS       33 HEX HALF WORDS


DATA AREA COPY OF ALL REGISTERS USED BY THIS FORTRAN PROGRAM
   START ADDRESS =           700                                    (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

      1180      4A      PI_DYNSP                                SIMPLE VARIABLE      INTGR
      11C0      4B      P_DYNBAS                                SIMPLE VARIABLE      INTGR
      1200      4C      L_TARVEC                                SIMPLE VARIABLE      INTGR
      1240      4D      LEN_TARG                                SIMPLE VARIABLE      INTGR
      1280      4E      V_TEMP1                                 SIMPLE VARIABLE      INTGR
      12C0      4F      V_TEMP2                                 SIMPLE VARIABLE      INTGR
      1300      50      V_TEMP3                                 SIMPLE VARIABLE      INTGR
      1340      51      V_TEMP4                                 SIMPLE VARIABLE      INTGR
      1380      52      C_#1                                    CONSTANT             INTGR
      13C0      53      C_#8                                    CONSTANT             INTGR
      1400      54      C_#A                                    CONSTANT             INTGR
      1440      55      K                                       SIMPLE VARIABLE      INTGR
      1480      56      J                                       SIMPLE VARIABLE      INTGR
      14C0      57      I                                       SIMPLE VARIABLE      INTGR
      1500      58,59   FT_WTTPR_ADR        ,FT_WTTPR_DB        REF.EXTERNAL SUBPR   UNKNW
      1580      5A,5B   FT_WTIPR_ADR        ,FT_WTIPR_DB        REF.EXTERNAL SUBPR   INTGR
      1600      5C,5D   FT_WTIE_ADR         ,FT_WTIE_DB         REF.EXTERNAL SUBPR   UNKNW
      1680      5E,5F   FT_WTIEA_ADR        ,FT_WTIEA_DB        REF.EXTERNAL SUBPR   UNKNW
      1700      60,61   FT_INIT_ADR         ,FT_INIT_DB         REF.EXTERNAL SUBPR   UNKNW
      1780      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

      1800      64      L_20_DESCR                              ARRAY NAME
      1840      65      L_18_DESCR                              ARRAY NAME
      1880      66      L_C00001_DESCR                          CHAR/BIT/FORMAT
      18C0      67      L_F4_DESCR                              CHAR/BIT/FORMAT
      1900      68      L_F3_DESCR                              CHAR/BIT/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 =       33 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 =           280       LENGTH =       0  HEX HALF WORDS      (START ADDRESS IS RELATIVE TO DATA AREA BASE ADDRESS)
```

Figure 15-2. Compiler Output Example (Sheet 4 of 5)

```
FORTRAN R1.5 CYCLE FL1      BUILT 06/19/80 11:44     STORAGE MAP          PASCAL     COMPILED 09/09/80 11:56 O=CAM              PAGE     7

 CONSTANTS,EXTERNALS,DESCRIPTORS AND NON-COMMON VARIABLES NOT ASSIGNED TO REGISTERS, NAMELISTS,CHARACTER SCALARS
    START ADDRESS =          280      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)

          280L                                ARRAY VARIABLE      INTGR


 TEMPORARY STORAGE
                                        LENGTH =       0  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)

           0    PASCAL

 LIST OF ALL EXTERNALS

                SYMBOLIC NAME

                  FT_WTTPR
                  FT_WTIPR
                  FT_WTIE
                  FT_WTIEA
                  FT_INIT
                  FT_STOP
      NO ERRORS
```

Figure 15-2. Compiler Output Example (Sheet 5 of 5)

The variable map provides information about each symbolic name used in a program except procedure names and symbolic constant names. See figure 15-3 for the format of the variable map. The variable map is always printed when the C compile option is selected. Some uses of the variable map include:

- Identifying symbolic names that are not associated with the proper data type
- Locating the place in the program where a value is assigned to a symbolic name
- Identifying functions that should be arrays
- Locating misspelled symbolic names
- Verifying that symbolic names are in the proper common blocks
- Locating all statements in a program in which a symbolic name is referenced, and identifying and locating symbolic names that are defined but never used.

See figure 15-2 for an example of a variable map.

```
VARIABLE MAP
---NAME----BLOCK----TYPE----CLASS----REFERENCES

     sym      blk      typ      cls      refs
     .        .        .        .        .
     .        .        .        .        .
     .        .        .        .        .
```

sym A symbolic name that appears in the program. Symbolic names are listed in alphabetical order.

blk The name of the common block in which sym appears. If sym appears in blank common, two consecutive slashes are printed for blk. If sym does not appear in any common block, the blk field is left blank.

typ The data type with which sym is associated; typ can be any of the following:

INTEGER
REAL
DOUBLE
COMPLEX
LOGICAL
CHAR*len (len is the character length)
BIT

cls The class of sym; cls can be any of the following:

SIMPLE
ARRAY
DESCRIPTOR
DESCRIPTOR ARRAY
UNKNOWN

refs The source line numbers of all source lines that contain references to sym. The source line numbers are listed in numerical order, and multiple references are listed. A source line number appearing in refs can be followed by a suffix. A suffix describes how sym is used in the source line. The suffixes and their meanings are:

/A The symbolic name sym is an actual argument in a subroutine call or function reference.

/C The symbolic name sym is the control variable of a DO loop.

/I The symbolic name sym is initialized in a DATA statement.

/R The symbolic name sym appears in the input/output list of an input statement.

/S The symbolic name sym appears on the left side of an assignment statement.

/W The symbolic name sym appears in the input/output list of an output statement.

Figure 15-3. Variable Map Format

The symbolic constant map provides information about each symbolic constant used in a program. See figure 15-4 for the format of the symbolic constant map. If no symbolic constants are used in a program, the symbolic constant map is not printed. Some uses of the symbolic constant map include:

- Verifying that the proper values are assigned to symbolic constant names
- Verifying that the symbolic constant names are associated with the proper data type
- Identifying and locating symbolic constant names that are defined but never used
- Locating the PARAMETER statement that defines each symbolic constant, and locating all occurrences of a symbolic constant in the program.

See figure 15-2 for an example of a symbolic constant map.

SYMBOLIC CONSTANT MAP
—NAME——TYPE——VALUE——REFERENCES

sym	typ	val	refs
.	.	.	.
.	.	.	.
.	.	.	.

sym The name of a symbolic constant that appears in the program. Symbolic constant names are listed in alphabetical order.

typ The data type with which sym is associated; typ can be any of the following:

INTEGER
REAL
DOUBLE
COMPLEX
LOGICAL
CHAR*len (len is the character length)
BIT

val The value assigned to the symbolic constant name sym. The format of var depends on the data type of sym:

Integer	The integer value is printed. A negative value is preceded by a minus sign.
Real and Double-precision	The value is printed as a hexadecimal string constant. The format is X'nnn'.
Complex	The complex value is printed as two hexadecimal string constants. The first constant represents the real part, and the second constant represents the imaginary part. The format is X'nnn',X'nnn'.
Logical	The logical value is printed as the logical constant .TRUE. or .FALSE..
Character	The character value is printed as a character string enclosed in apostrophes. If the string is too long to fit in the columns provided, the trailing apostrophe is replaced by an ellipsis.
Bit	The bit value is printed as a bit string constant. The format is B'nnn'. If the string is too long to fit in the columns provided, the trailing apostrophe is replaced by an ellipsis.

refs The source line numbers of all source lines that contain references to sym. The source line numbers are listed in numerical order, and multiple references are listed. A source line number appearing in refs can be followed by the suffix /S, which indicates that the symbolic constant is defined in that source line.

Figure 15-4. Symbolic Constant Map Format

The procedure map provides information about subroutines, functions, statement functions, and external symbolic names used in a program. If no procedures are used in a program, the procedure map is not printed. See figure 15-5 for the format of the procedure map. Some of the uses of the procedure map include:

- Identifying statement functions that should be arrays
- Verifying that procedure names are associated with the proper data types
- Identifying misspelled procedure names
- Locating statement function definitions
- Identifying and locating statement function names that are defined but never used
- Locating all references to a procedure name

See figure 15-2 for an example of a procedure map.

ASSEMBLY LISTING

When the A compile option is selected, a listing of the assembly representation of the FORTRAN program appears after any cross-reference maps. Given are the location counter (the offset from the code area base address), the machine instruction in hexadecimal (either halfword or fullword instruction), the source line number of the associated source program statement, the instruction mnemonic, instruction qualifiers, and operands. See figure 15-2 for an example of an assembly listing. See the CYBER 200 Assembler reference manual for more information about the assembly language.

```
PROCEDURE MAP
--NAME---TYPE---CLASS---REFERENCES

    sym      typ      cls      refs
    .        .        .        .
    .        .        .        .
    .        .        .        .
```

sym The symbolic name of a subroutine, function, statement function, or external symbol. Symbolic names are listed in alphabetical order.

typ The data type of the procedure result; typ can be any of the following:

```
INTEGER
REAL
DOUBLE
COMPLEX
LOGICAL
CHAR*len (len is the character length)
BIT
GENERIC (for generic functions)
```

If the symbolic name sym is a subroutine name or an external symbol, the typ field is left blank.

cls The class of sym; cls can be any of the following:

```
SUBROUTINE   (Subroutine)
DUMMY SUBR   (Subroutine name is a dummy argument)
INTRINSIC    (Intrinsic function)
STAT FUNC    (Statement function)
BASIC EXTRN  (Basic external function)
DUMMY FUNC   (Function name is a dummy argument)
EXTERNAL     (Procedure name appears in an EXTERNAL statement and is not one of the above)
```

refs The source line numbers of all source lines that contain references to sym. The source line numbers are listed in numerical order, and multiple references are listed. If sym is a statement function name, a source line number in refs can be followed by the suffix /D, which indicates that the statement function is defined in that source line.

Figure 15-5. Procedure Map Format

REGISTER MAP AND STORAGE MAP

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 CYBER 200 FORTRAN register usage conforms to standard CYBER 200 operating system register conventions, which are described in volume 2 of the CYBER 200 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

See figure 15-2 for an example of a register map and a storage map.

EXECUTION-TIME FILE REASSIGNMENT

The PROGRAM statement declarations for files can be entirely or partially overridden at program 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).

When the first form is used, the file declarations in the PROGRAM statement are ignored and the file declarations in the message are used. When the second form is used, 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 the form that is 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 by 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.

This section consists of examples of FORTRAN programs illustrating some of the features of the CYBER 200 FORTRAN programming language.

PROGRAM PASCAL

Program PASCAL produces a table of binomial coefficients (Pascal's triangle).

Features:

FORTRAN system control statement

Nested DO loops

DATA statement

Implied DO loop

The source listing for the program, along with the output generated by execution of the program, is shown in figure 16-1. The system control statements perform the following functions:

- FORTRAN.

  This requests that the program in the file INPUT be compiled by the FORTRAN compiler, with the object code generated during compilation to be placed in the

System control statements:

```
FORTRAN.
LOAD.
GO.
```

Source listing:

```
      PROGRAM PASCAL (OUTPUT)
      INTEGER L(11)
      DATA L(11) /1/
C
      PRINT 4, (I,I=1,11)
4     FORMAT(44H1COMBINATIONS OF M THINGS TAKEN N AT A TIME.//20X,3H-N-/
     111I5)
      DO 200  I=1,10
      K=11-I
      L(K)=1
      DO  100  J=K,10
100   L(J)=L(J)+L(J+1)
200   PRINT 3,(L(J),J=K,11)
3     FORMAT (11I5)
C
      STOP
      END
```

Program output:

```
 COMBINATIONS OF M THINGS TAKEN N AT A TIME.

                    -N-
    1    2    3    4    5    6    7    8    9   10   11
    2    1
    3    3    1
    4    6    4    1
    5   10   10    5    1
    6   15   20   15    6    1
    7   21   35   35   21    7    1
    8   28   56   70   56   28    8    1
    9   36   84  126  126   84   36    9    1
   10   45  120  210  252  210  120   45   10    1
   11   55  165  330  462  462  330  165   55   11    1
```

Figure 16-1. Program PASCAL

file BINARY and compiler-generated listings to be placed in the file OUTPUT. All defaults for the statement have been selected, including the names of the files INPUT, BINARY, and OUTPUT.

- LOAD.

  This requests that the loader build a controllee file GO consisting of the object code from the file BINARY, and resolve any unsatisfied external references from the library SYSLIB. All defaults for the statement have been selected, including the names of the files BINARY, SYSLIB, and GO.

- GO.

  This requests that the file GO be executed. Program output is produced as a result of this statement.

The functions of some of the FORTRAN statements in program PASCAL are:

- INTEGER L(11)

  L is defined as an 11-element integer array.

- DATA L(11)/1/

  The DATA statement stores the value 1 in the last element of the array L. When the program is executed L(11) has the initial value 1.

- PRINT 4,(I,I=1,11)

  This statement prints the headings. The implied DO loop generates the values 1 through 11 for the column headings.

- PRINT 3,(L(J),J=K,11)

  The index value J is used as a subscript and is not printed. The end of the array is printed from a variable starting position. The 1, which appears on the diagonal in the output is not moving in the array; it is always in L(11); but the starting point is moving.

- DO 200 I=1,10
  K=11-I

  These statements illustrate the technique of going backwards through an array. As I goes from 1 to 10, K goes from 10 to 1. The increment value in a DO statement must be positive; therefore, this DO loop provides a legal method of writing the illegal statement DO 200 K=10,1,-1.

- DO 100 J=K,10
  100 L(J)=L(J)+L(J+1)

  This inner DO loop generates the line of values output by statement number 200. When control reaches statement 200, the variable J can be used again because statement number 200 is outside the inner DO loop. However, if I were used in statement 200 instead of J, the statement 200 PRINT 3,(L(I),I=K,11) would be in error. Statement 200 is inside the outer DO loop and would change the value of the control variable from within the loop. Changing the value of a loop's control variable from inside the loop is bad practice and, in this case, would cause a fatal error or a never-ending loop.

```
Type:   INTEGER I/2/,J/2/
        REAL A/3.0/

Data:   DATA I,J,A/2*2,3.0/ or DATA I,J,A/2*X'2',3.0/

Asst:   I=2
        J=2
        A=3.0


Type:   CHARACTER CHA*5/'AEIOU'/,CHB*3/'123'/

Data:   CHARACTER CHA*5,CHB*3
        DATA CHA/'AEIOU'/,CHB/'123'/

Asst:   CHARACTER CHA*5,CHB*3
        CHA='AEIOU'
        CHB='123'


Type:   REAL A/3.0/
        CHARACTER*5 CHA/'AEIOU'/,CHB/'123'/
          (CHB contains 123ΔΔ after initialization)

Data:   IMPLICIT CHARACTER*5 (C),REAL(A)
        DATA CHA,A,CHB/'AEIOU',3.0,'123'/
          (CHB contains 123ΔΔ after initialization)

Asst:   IMPLICIT CHARACTER*5 (C),REAL(A)
        CHA='AEIOU'
        A=3.0
        CHB='123'
          (CHB contains 123ΔΔ after initialization)


Type:   REAL A/3.0/
        BIT C/B'1'/,D/B'1'/

Data:   IMPLICIT BIT (C,D)
        DATA A,C,D /3.0,2*B'1'/

Asst:   IMPLICIT BIT (C,D)
        A=3.0
        C=B'1'
        D=B'1'
```

Figure 16-2. Examples of Initializing Simple Variables and Array Elements

DATA INITIALIZATION

Each of figures 16-2 through 16-7 compares the statements required to define the value of a particular kind of data element, using explicit type statements, DATA statements, and assignment statements. Each type-data-assignment statement group performs the same definition of the data elements involved, in three different ways. Type and DATA statements, being nonexecutable, are processed once. The assignment statements could be executed any number of times (depending on the program's other executable statements).

PROGRAM ADD

Program ADD illustrates the use of the DECODE statement. The ENCODE and DECODE statements are easier to understand when related to the READ and WRITE statements.

```
Type:   BIT B(20)/B'10',2*X'0',B'1111110111'/
        REAL A/3.0/, . . .
             or
        BIT B(20)/X'803F7'/
        REAL A/3.0/, . . .
             or
        BIT B(20)/B'10000000001111110111'/
        REAL A/3.0/, . . .

Data:   BIT B(20)
        DATA A,B/3.0,B'10',2*X'0',B'1111110111'/
             or
        BIT B(20)
        DATA A,B/3.0, X'803F7'/
             or
        BIT B(20)
        DATA A,B/3.0,B'10000000001111110111'/

Asst:   BIT B(20)
        A=3.0
        B=B'0'
        B(1)=B'1'
        B(11:16)=B'1'
        B(18:20)=B'1'

Type:   INTEGER I(10)/0,1,2,3,4,5,6,7,8,9/
        REAL A/3.0/

Data:   DIMENSION I(10)
        DATA A,I/3.0,0,1,2,3,4,5,6,7,8,9/

Asst:   DIMENSION I(10)
        DO  100 N=1,10
    100 I(N)=N-1
        A=3.0

Type:   INTEGER I(10)/10*0/
        REAL A/3.0/

Data:   DIMENSION I(10)
        DATA A,I/3.0,10*0/

Asst:   DIMENSION I(10)
        I=0
        A=3.0

Type:   INTEGER I(10)/1,0,2,0,3,0,4,0,5,0/
        REAL A/3.0/

Data:   DIMENSION I(10)
        DATA A,(I(N),N=1,9,2),
       $     (I(N+1),N=1,9,2)/3.0,1,2,3,4,5,5*0/

Asst:   DIMENSION I(10)
        A=3.0
        I(2:10:2)=0
        I(1)=1
        I(3)=2
        I(5)=3
        I(7)=4
        I(9)=5

Type:   CHARACTER C*4(10)/10*'NULL'/
        REAL A/3.01

Data:   CHARACTER C*4(10)
        DATA A,C/3.0,10*'NULL'/

Asst:   CHARACTER C*4(10)
        C='NULL'
        A=3.0
```

Figure 16-3. Examples of Initializing Simple Arrays

```
Type:   INTEGER I(10)/0,1,2,3,4,5,6,7,8,9/
        REAL A/3.0/

Data:   DIMENSION I(10)
        DATA A,I(1;10)/3.0,0,1,2,3,4,5,6,7,8,9/

Asst:   DIMENSION I(10)
        DO 600 N=1,10
    600 I(N)=N-1
        A=3.0


Type:   INTEGER I(10)/10*0/,I2(10)/10*0/
        REAL A/3.0/

Data:   DIMENSION I(10),I2(10)
        DATA A,I(1;10),I2(1;10)/3.0,20*0/

Asst:   DIMENSION I(10),I2(10)
        I(1;10)=0
        I2(1;10)=0
        A=3.0


Type:   COMPLEX COMP(50)/50*(0.,0.)/
        REAL A/3.0/

Data:   COMPLEX COMP(50)
        DATA A,COMP/3.0,10*(0.,0.)/

Asst:   COMPLEX COMP(50)
        COMP (20;10)=(0.,0.)
        A=3.0
```

Figure 16-4. Examples of Vector Initialization

```
Type:   (no type statement equivalent)

Data:   DESCRIPTOR ID
        DIMENSION I(10)
        DATA A,ID/3.0,I(1;10)/

Asst:   DESCRIPTOR ID
        DIMENSION I(10)
        ASSIGN ID,I(1;10)
        A=3.0
```

Figure 16-5. Example of Descriptor Initialization

```
Type:   (no type statement equivalent)

Data:   DESCRIPTOR ID(8)
        DIMENSION I(10)
        DATA A,ID(4)/3.0,I(1;10)/

Asst:   DESCRIPTOR ID(8)
        DIMENSION I(10)
        ASSIGN ID(4),I(1;10)
        A=3.0
```

Figure 16-6. Example of Descriptor Array Element Initialization

Features:

DECODE statement

The source listing, input, and output for program ADD are shown in figure 16-8.

The functions of some of the FORTRAN statements in program ADD are:

- DECODE (READ)

  A READ statement places the image of each record read into an input buffer. Compiler routines convert the character string in the record into floating point, integer or logical values, as specified by the FORMAT statement, and store these values in the locations associated with the variables named in the list.

  With DECODE, the array specified in the DECODE statement is used as the input buffer.

```
Type:   (no type statement equivalent)

Data:   DESCRIPTOR ID(8)
        DIMENSION I(60),J(20)
        DATA A,ID/3.0,I(1;10),I(11;10),I(21;10),I(31;10),
       $     I(41;10),I(51;10),J(1;10),J(11;10)/

Asst:   DESCRIPTOR ID(8)
        DIMENSION I(60),J(20)
        A=3.0
        DO 100 K=1,6
    100 ASSIGN ID(K),I(10*K-9;10)
        ASSIGN ID(7),J(1;10)
        ASSIGN ID(8),J(11;10)
```

Figure 16-7. Example of Descriptor Array Initialization

Program:

```
      PROGRAM ADD(INPUT,OUTPUT,TAPE5=INPUT,TAPE6=OUTPUT)
      INTEGER CARD(10),REC(79),TOTAL
C
C  DECODE STATEMENT ILLUSTRATION:  TRANSFER DATA FROM
C  BUFFER (CARD) TO ARRAY (REC) FOR OUTPUT.  SUM ELEMENTS TOO.
C
600   READ (5,100,END=1000) KEY,CARD
C
C  SELECT WHICH DECODE IS TO BE USED
C
      KEY=MAX0 (1,MIN0 (KEY,3))
      GO TO (1,2,3),KEY
C
C  USE FORMAT STATEMENT 200
C
1     DECODE (79,200,CARD)  REC
      N=79
      GO TO 40
C
C  USE FORMAT STATEMENT 300
C
2     DECODE (78,300,CARD) (REC(I),I=1,39)
      N=39
      GO TO 40
C
C  USE FORMAT STATEMENT 400
C
3     DECODE (78,400,CARD) (REC(I),I=1,26)
      N=26
C
C  SUM ELEMENTS
C
40    TOTAL=0
      DO  41  I=1,N
41    TOTAL=TOTAL+REC(I)
C
      WRITE (6,500) TOTAL,N,KEY,CARD,(REC(I),I=1,N)
      GO TO 600
100   FORMAT (I1,9A8,A7)
200   FORMAT (79I1)
300   FORMAT (39I2)
400   FORMAT (26I3)
500   FORMAT (/I6,20H IS THE TOTAL OF THE ,I3,20H NUMBERS ON THE CARD/
     1I2,10A8   /16H THE NUMBERS ARE/(20I4))
1000  STOP
      END
```

Figure 16-8. Program ADD (Sheet 1 of 2)

Program input:

```
3654475633321455578963332020258520202851741714446655998879654100258741235896333

00223210233256555458777563222458999630222145632508745050512406303698521475283696

852475369888521475360369095280870741028063952836985274144455338722599066044025888
```

Program output:

```
13912 IS THE TOTAL OF THE 26 NUMBERS ON THE CARD
3654475633321455578963332020258520202851741714446655998879654100258741235896333
THE NUMBERS ARE
654 475 633 321 455 578 963 332  20 258 520 202 851 741 714 446 655 998 879 654
100 258 741 235 896 333


  323 IS THE TOTAL OF THE 79 NUMBERS ON THE CARD
1022321023325655545877563222458999630222145632508745050512406303698521475283696
THE NUMBERS ARE
  0   2   2   3   2   1   0   2   3   3   2   5   6   5   5   5   4   5   8   7
  7   7   5   6   3   2   2   2   4   5   8   9   9   9   6   3   0   2   2   2
  1   4   5   6   3   2   5   0   8   7   4   5   0   5   0   5   1   2   4   0
  6   3   0   3   6   9   8   5   2   1   4   7   5   2   8   3   6   9   6


15363 IS THE TOTAL OF THE 26 NUMBERS ON THE CARD
3524753698885214753603690952808707410280639528369852741444553387225990660440258
THE NUMBERS ARE
524 753 698 885 214 753 603 690 952 808 707 410 280 639 528 369 852 741 444 553
387 225 990 660 440 258
```

Figure 16-8. Program ADD (Sheet 2 of 2)

With the READ statement, when the FORMAT specification indicates a new record is to be processed (by a slash or the final right parenthesis of the FORMAT statement), a new record is read into the input buffer.

With the DECODE statement, when the FORMAT statement indicates a new record is to be processed (by a slash or final right parenthesis), the next part of the array is used as the input buffer. The record length indicates the number of characters used for each record.

- ENCODE (WRITE)

A WRITE statement causes the output buffer to be cleared. Data in the WRITE statement list is converted into a character string according to the format specified in the FORMAT statement, and placed in the output buffer. When the FORMAT statement indicates an end of a record with either a slash or the final right parenthesis, the character string is passed from the output buffer to the output system; the output buffer area is reset, and the next string of characters is placed in the buffer.

The ENCODE statement is processed by compiler routines in the same way as the WRITE statement, but with the array specified within the parentheses of the ENCODE statement used as the output buffer. The number of characters per record in the array is determined by the record length.

In program ADD, the format of data on input is specified in column 1. If column 1 is a one, each of the remaining columns is a data item. If column 1 is a two, each pair of the remaining columns is a data item. If column 1 is a three or greater, each triplet of the remaining columns is a data item. Based on the information in column 1, the correct DECODE statement (the proper format and item count) is selected. The program then totals and prints out the items in each input record.

- INTEGER CARD(10),REC(79),TOTAL

CARD is dimensioned 10 to receive the 79 characters in columns 2 through 80. REC is dimensioned 79 to receive the numeric values of the input items.

- 600 READ(5,100,END=1000)KEY,CARD
  100 FORMAT(I1,9A8,A7)

The first column of the record is read into KEY under I format, and the remaining 79 characters are read into the array CARD under A format, so they can be converted later to I format with a DECODE statement. The END parameter of the READ statement tests for the end of data in which case the program simply stops because statement 1000 is a STOP statement.

- KEY=MAX0(1,MIN0(KEY,3))

Guarantees that $1 \leq KEY \leq 3$.

```
40  TOTAL=0
    DO 41 I=1,N
41  TOTAL=TOTAL+REC(I)
```

Adds up the items and leaves the total in TOTAL.

```
    WRITE(6,500)TOTAL,N,KEY,CARD,(REC(I),I=1,N)
500 FORMAT(/I6,20H IS THE TOTAL OF THE ,I3,20H
    NUMBERS ON THE CARD/I2,10A8/16H THE
    NUMBERS ARE/(20I4))
```

Outputs the results.

```
GOTO 600
```

Goes back to process the next record.

PROGRAM CPVECT

Program CPVECT illustrates the use of complex vector elements.

Features:

Complex vectors

Vector function VCSQRT

A listing of program CPVECT is shown in figure 16-9.

The functions of some of the FORTRAN statements in program CPVECT are:

```
COMPLEX A(20),B(20),C(20),ROOT1(20),
ROOT2(20),D(20)
```

Arrays A, B, and C contain the coefficients for 20 quadratic equations; A(1), B(1), and C(1) for example, contain the coefficients for the first equation, whose principal root will be placed in ROOT1(1) and whose other root will be placed in ROOT2(1). D is a working array.

```
COMPLEX     DA,DB,DC,DD,DROOT1,DROOT2
DESCRIPTOR  DA,DB,DC,DD,DROOT1,DROOT2
DATA DA/A(1;20)/,DB/B(1;20)/,DC/C(1;20)/,
DD/D/(1;20)/
DATA DROOT1 /ROOT1(1;20)/,
DROOT2 /ROOT2(1;20)/
```

Descriptors are defined for elegance and conciseness in the rest of the program. For example, after descriptors have been defined, DA and A(1;20) can be used interchangeably in assignment statements.

```
    DO 100 I=1,20
100 A(I)=I
```

Array A is initialized.

```
DB = DA + 3
DC = DA + 1
```

These complex vector arithmetic assignment statements initialize arrays B and C. In the first statement, the constant 3 is added to each element of the array A. The result of adding 3 to A(1), for example, is used to define B(1).

```
DD = VCSQRT(DD;DD)
```

This complex vector arithmetic assignment statement computes the square root of a vector of intermediate values storing the result of the function back into array D.

```
PRINT 1,ROOT1,ROOT2
```

The arrays ROOT1 and ROOT2, which contain the results of computing the roots for 20 equations, are printed with this statement.

Source listing:

```
      PROGRAM CPVECT (OUTPUT)
C
C  CALCULATE N PAIRS OF ROOTS FOR N QUADRATIC EQUATIONS
C  THE I-TH EQUATION IS A(I)*X**2 + B(I)*X + C(I) = 0
C
      COMPLEX A(20),B(20),C(20),ROOT1(20),ROOT2(20),D(20)
      COMPLEX     DA,DB,DC,DD,DROOT1,DROOT2
      DESCRIPTOR DA,DB,DC,DD,DROOT1,DROOT2
      DATA DA /A(1:20)/, DB /B(1:20)/, DC /C(1:20)/, DD /D(1:20)/
      DATA DROOT1 /ROOT1(1:20)/, DROOT2 /ROOT2(1:20)/
C
C  INITIALIZE ARRAYS
C
      DO  100  I=1,20
100   A(I) = I
      DB = DA + 3
      DC = DA + 1
C
C  CALCULATE THE DISCRIMINANTS
C
      DD = DB * DB-4 * DA * DC
      DD = VCSQRT(DD:DD)
C
C  CALCULATE THE ROOTS
C
      DROOT1 = (-DB+DD) / (2*DA)
      DROOT2 = (-DB-DD) / (2*DA)
C
C  PRINT RESULTS
C
      PRINT 1,ROOT1,ROOT2
1     FORMAT (1H1//10X,'ROOT1 CONTAINS THE FOLLOWING',
     X 4(/5X, 5('(',F6.3,',',F6.3,')')))
     X          ///10X,'ROOT2 CONTAINS THE FOLLOWING',
     X 4(/5X, 5('(',F6.3,',',F6.3,')')))
C
      STOP
      END
```

Program output:

```
     ROOT1 CONTAINS THE FOLLOWING
( -.586,    .0)(-1.000,    .0)(-1.000,  .577)( -.875,  .696)( -.800,  .748)
( -.750,  .777)( -.714,  .795)( -.688,  .808)( -.667,  .816)( -.650,  .823)
( -.636,  .828)( -.625,  .832)( -.615,  .836)( -.607,  .838)( -.600,  .841)
( -.594,  .843)( -.588,  .844)( -.583,  .846)( -.579,  .847)( -.575,  .848)


     ROOT2 CONTAINS THE FOLLOWING
(-3.414,    .0)(-1.500,    .0)(-1.000, -.577)( -.875, -.696)( -.800, -.748)
( -.750, -.777)( -.714, -.795)( -.688, -.808)( -.667, -.816)( -.650, -.823)
( -.636, -.828)( -.625, -.832)( -.615, -.836)( -.607, -.838)( -.600, -.841)
( -.594, -.843)( -.588, -.844)( -.583, -.846)( -.579, -.847)( -.575, -.848)
```

Figure 16-9. Program CPVECT

The CYBER 200 FORTRAN compiler recognizes 52 characters; the FORTRAN character set is a subset of the CYBER 200 character set.

Table A-1 shows both the FORTRAN character set and the CYBER 200 character set. Some of the characters in the CYBER 200 character set do not have corresponding characters in the FORTRAN character set; therefore, those characters cannot be used in a program unless they appear in a comment or in a character string.

Table A-1 also shows the internal hexadecimal representation and the Hollerith punch code for each character. Each hexadecimal digit in the internal hexadecimal representation corresponds to 4 bits. The Hollerith punch code indicates the rows that are punched in a computer card for each character.

Some characters do not appear on all keypunches and terminals. If a particular character is not represented on a keypunch or terminal, a character that appears on the keypunch or terminal that has the same internal hexadecimal representation can be substituted.

TABLE A-1. CHARACTER SETS

FORTRAN Character	Hex	CYBER 200 Character	Hollerith Punch (029)
Δ space	20	Δ space	no punch
	21	! exclamation point	12-8-7
	22	" quote	8-7
	23	# pound sign	8-3
	24	$ dollar sign	11-8-3
	25	% percent sign	0-8-4
& ampersand	26	& ampersand	12
' apostrophe	27	' apostrophe	8-5
(left parenthesis	28	(left parenthesis	12-8-5
) right parenthesis	29	) right parenthesis	11-8-5
* asterisk	2A	* asterisk	11-8-4
+ plus	2B	+ plus	12-8-6
, comma	2C	, comma	0-8-3
- minus	2D	- minus	11
. period	2E	. period	12-8-3
/ slash	2F	/ slash	0-1
0	30	0	0
1	31	1	1
2	32	2	2
3	33	3	3
4	34	4	4
5	35	5	5
6	36	6	6
7	37	7	7
8	38	8	8
9	39	9	9
: colon	3A	: colon	8-2
; semicolon	3B	; semicolon	11-8-6
	3C	< less than	12-8-4
= equals sign	3D	= equals sign	8-6
	3E	> greater than	0-8-6
	3F	? question mark	0-8-7
	40	@ commercial at	8-4
A	41	A	12-1
B	42	B	12-2
C	43	C	12-3
D	44	D	12-4
E	45	E	12-5
F	46	F	12-6
G	47	G	12-7

TABLE A-1. CHARACTER SETS (Contd)

FORTRAN Character	Hex	CYBER 200 Character	Hollerith Punch (029)
H	48	H	12-8
I	49	I	12-9
J	4A	J	11-1
K	4B	K	11-2
L	4C	L	11-3
M	4D	M	11-4
N	4E	N	11-5
O	4F	O	11-6
P	50	P	11-7
Q	51	Q	11-8
R	52	R	11-9
S	53	S	0-2
T	54	T	0-3
U	55	U	0-4
V	56	V	0-5
W	57	W	0-6
X	58	X	0-7
Y	59	Y	0-8
Z	5A	Z	0-9
[left bracket	5B	[left bracket	12-8-2
	5C	\ reverse slash	0-8-2
] right bracket	5D	] right bracket	11-8-2
	5E	^ circumflex	11-8-7
	5F	_ underscore	0-8-5
	60	` reverse apostrophe	8-1
	61	a	12-0-1
	62	b	12-0-2
	63	c	12-0-3
	64	d	12-0-4
	65	e	12-0-5
	66	f	12-0-6
	67	g	12-0-7
	68	h	12-0-8
	69	i	12-0-9
	6A	j	12-11-1
	6B	k	12-11-2
	6C	l	12-11-3
	6D	m	12-11-4
	6E	n	12-11-5
	6F	o	12-11-6
	70	p	12-11-7
	71	q	12-11-8
	72	r	12-11-9
	73	s	11-0-2
	74	t	11-0-3
	75	u	11-0-4
	76	v	11-0-5
	77	w	11-0-6
	78	x	11-0-7
	79	y	11-0-8
	7A	z	11-0-9
	7B	{ left brace	12-0
	7D	} right brace	11-0

This appendix contains descriptions of three basic groups of diagnostics: compilation diagnostics, run-time diagnostics, and vector messages.

COMPILER FAILURE AND COMPILATION ERRORS

Compiler failure messages are messages generated because of compiler failure. Compilation errors are messages generated because of errors in the program. The seriousness of the error is indicated by the error type.

COMPILER FAILURE

Error messages produced when the compiler fails are listed in table B-1. The compiler failure error type is:

A (abort) Compilation was terminated because of compiler failure. The return code is 8 (RC=8)

COMPILATION ERRORS

Error messages produced when the compiler detects errors in the source program are listed in table B-2. Compilation error types are:

W (warning) The statement in error was compiled. Compilation continued, but part of the statement might not have been processed. The return code is 4 (RC=4).

F (fatal) The statement in error was not compiled. Object code generation is inhibited. The return code is 8 (RC=8).

RETURN CODES

The user has control over the execution of a batch job in that the user can determine whether to initiate error exit processing or to allow batch job processing to continue. The TV control statement allows a termination value to be

TABLE B-1. COMPILER FAILURE MESSAGES

Error Number	Type	Message	Significance	Action
93	A	COMPILER FAILURE - REFERENCE FOR NON-DIMENSIONED ARRAY	The subscript processor detected a bad symbol table entry.	Follow site-defined procedure.
94	A	COMPILER FAILURE - ALL FULL REG TABLE ENTRIES ARE CLASS 4	The doubleword register assignment table became invalid during the generation phase.	Follow site-defined procedure.
95	A	COMPILER FAILURE - HALF REG TABLE ENTRIES ARE CLASS 4	The fullword register assignment table became invalid during the generation phase.	Follow site-defined procedure.
96	A	COMPILER FAILURE - VARIABLE EQUIVALENCED TO COMMON BLOCK THAT HAS NO ELEMENT	The storage class table became invalid during the allocation phase.	Follow site-defined procedure.
97		(Currently unassigned)	--	--
98	A	COMPILER FAILURE - I/O STACK FORMED INCORRECTLY	The input/output list stack that was built by the IOLIST processor became invalid during the parse phase.	Follow site-defined procedure.
99	A	COMPILER FAILURE - ILLEGAL DESCRIPTOR ENCOUNTERED IN ALLOCATION PHASE(2)	The descriptor table became invalid.	Follow site-defined procedure.
100	A	COMPILER FAILURE - TABLE AREA OVERFLOW	One of the compiler table areas reached its maximum size. Possibly the program was too big to be compiled.	Follow site-defined procedure.
101	A	COMPILER FAILURE	Compiler detected an internal inconsistency.	Follow site-defined procedure.

TABLE B-2. COMPILATION ERROR MESSAGES

Error Number	Type	Message	Significance	Action
102	F	ILLEGAL SUBPROGRAM NAME	The subprogram is compiled as a main program.	Correct error; recompile.
103	F	FUNCTION CANNOT BE CALLED AS A SUBROUTINE	A function is called with a CALL statement.	Replace the CALL statement with a statement that contains a function reference; recompile.
104	W	CANNOT TYPE SUBROUTINE NAME	A type is specified for the subroutine name; the type was ignored by the compiler.	Verify that a subroutine, rather than a function, was intended.
105	F	ILLEGAL SUBROUTINE REFERENCE	A subroutine name is used improperly.	Correct error; recompile.
106	F	MISSING OPERATOR OR DELIMITER	An operator or delimiter is required.	Supply missing operator or delimiter; recompile.
107	F	ILLEGAL OPERAND	An expression contains an illegal operand.	Correct error; recompile.
108	F	ILLEGAL OR MISSING DELIMITER	A delimiter is required.	Supply missing delimiter or correct error in existing delimiter; recompile.
109	F	ILLEGAL USE OF ARRAY NAME	An array name appears without a subscript.	Supply subscript for array reference; recompile.
110	F	MISSING LEFT PARENTHESIS	A left parenthesis is required.	Supply missing left parenthesis; recompile.
111	F	ILLEGAL USE OF HEXADECIMAL CONSTANT	A hexadecimal constant is used improperly.	Correct error; recompile.
112	F	RECURSIVE SUBPROGRAM REFERENCE IS ILLEGAL	A subprogram calls itself.	Remove recursive subprogramre ferences from the program; recompile.
113	F	ILLEGAL ARGUMENT DELIMITER	Arguments must be delimited by commas.	Correct error; recompile.
114	F	ILLEGAL USE OF SUBPROGRAM NAME	A subroutine or function name is used improperly.	Correct error; recompile.
115	F	ILLEGAL ARGUMENT IN INTRINSIC OR BASIC FUNCTION REFERENCE	The arguments are not what the function requires.	Correct error; recompile.
116	W	FUNCTION NAME USED AS ARGUMENT NOT DECLARED EXTERNAL	The function name is not declared in an EXTERNAL statement.	Declare function name in an EXTERNAL statement; recompile.
117	F	INTRINSIC FUNCTION CANNOT BE ACTUAL ARGUMENT	An intrinsic function name appears in the argument list of a function or subroutine reference.	Remove intrinsic function name from the argument list; recompile.
118	F	ILLEGAL OPERATOR IN EXPRESSION	The operator cannot be used in the expression.	Correct error; recompile.
119	F	PARENTHESES DO NOT MATCH OR ILLEGAL ASSIGNMENT STATEMENT	A one-to-one correspondence does not exist between left and right parentheses.	Check all parentheses in the expression. Correct errors; recompile.

TABLE B-2. COMPILATION ERROR MESSAGES (Contd)

Error Number	Type	Message	Significance	Action
120	F	INCORRECT NUMBER OF ARGUMENTS FOR INTRINSIC OR BASIC FUNCTION	The argument list for an intrinsic function reference or a basic function reference contains a different number of arguments than the function requires.	Check the requirements of the intrinsic or basic function. Add missing arguments or delete extra arguments from the argument list of the function reference; recompile.
121	F	INCORRECT ARGUMENT TYPE FOR INTRINSIC OR BASIC FUNCTION	An argument that appears in the argument list of an intrinsic function reference or a basic function reference is of the wrong type.	Check the requirements of the intrinsic or basic function. Change the type of the erroneous argument; recompile.
122	F	ILLEGAL TYPE MIXING IN STATEMENT	The data types of two entities that appear in a statement are incompatible.	Correct error; recompile.
123	F	ILLEGAL ARRAY MODE IN VECTOR REFERENCE		Correct error; recompile.
124	F	ILLEGAL MODE USAGE IN RELATIONAL OR ARITHMETIC EXPRESSION		Correct error; recompile.
125	W	MORE THAN 19 CONTINUATION LINES	All continuation lines after line 19 are not compiled.	Restructure the statement so that no more than 19 continuation lines are used; recompile.
126	W	THIS STATEMENT CANNOT BE EXECUTED	The previous statement does not allow execution of this statement.	Check for an error in logic. Check for a missing label on the current statement.
127	W	INDEFINITE RESULT, PRODUCT TOO LARGE	The multiplication of two constants produces a result that is too large.	Verify that an indefinite result does not affect the logic of the program.
128	W	DIVIDE FAULT IN CONSTANT ARITHMETIC	The division of one constant by another produces a divide fault.	Verify that the divide fault does not affect the logic of the program.
129	W	EXPONENT OVERFLOW IN CONSTANT ARITHMETIC	Constant arithmetic produces exponent overflow.	Verify that exponent overflow does not affect the logic of the program.
130	F	ILLEGAL DELIMITER IN A VECTOR REFERENCE		Correct error; recompile.
131	F	SUBSCRIPT FOR NON-DIMENSIONED ARRAY, OR STMT FUNCTION DEF DOES NOT PRECEDE ALL EXECUTABLE STATEMENTS	The array that appears on the left side of an assignment is not dimensioned, or this is a statement function definition that does not precede all executable statements.	Correct error; recompile.
132	F	THIS SYMBOL MAY NOT BE DEFINED TO BE A STATEMENT FUNCTION	The symbol is already defined.	Correct error; recompile.

TABLE B-2. COMPILATION ERROR MESSAGES (Contd)

Error Number	Type	Message	Significance	Action
133	F	ILLEGAL STATEMENT FUNCTION ARGUMENT	An illegal argument appears in a statement function reference.	Correct error; recompile.
134	F	ILLEGAL STATEMENT FUNCTION DEFINITION	A statement function is defined improperly.	Correct error; recompile.
135	F	ILLEGAL LABEL	A label must be numeric and between 1 and 99999.	Supply numeric label; recompile.
136	F	DESCRIPTOR MODE IS NOT INTEGER, REAL, BIT, OR COMPLEX	A descriptor must be of one of these types.	Change the type of the descriptor; recompile.
137	F	ILLEGAL DELIMITER FOR HEX OR BIT CONSTANT	Hexadecimal and bit constants must be delimited by apostrophes.	Change delimiters to apostrophes; recompile.
138	F	DOUBLY DEFINED LABEL	The same label appears on more than one statement in a program.	Change one of the occurrences of the label. Also, check all references to the label that is changed in order to maintain correct logic; recompile.
139	F	(Currently unassigned)	--	--
140	F	ILLEGAL DELIMITER IN STATEMENT FUNCTION ARGUMENT LIST	Statement function arguments must be delimited by commas.	Correct error; recompile.
141	F	INCORRECT NO. OF ARGUMENTS FOR STATEMENT FUNCTION	The argument list for a statement function reference contains a different number of arguments than the function requires.	Check the statement function definition to find out how many arguments the function requires. Add missing arguments or delete extra arguments from the argument list of the function reference; recompile.
142	F	COMPLEX MAY NOT BE USED AS POWER	A complex number appears as an exponent.	Change the type of the exponent; recompile.
143	F	COMPLEX MAY ONLY BE RAISED TO INTEGER OR REAL POWER	Exponentiation of a complex number involves an exponent that is not real or integer.	Change the type of the exponent to real or integer; recompile.
144	F	SUBSCRIPT MUST BE INTEGER CONSTANT	The subscript is not an integer constant.	Change the subscript to integer constant; recompile.
145	F	SPECIFICATION STATEMENTS MUST PRECEDE ALL EXECUTABLE STATEMENTS	A specification statement appears after an executable statement.	Move all specification statements in front of all executable statements; recompile.
146	F	ILLEGAL VARIABLE IN DATA STATEMENT	A symbol that appears in a DATA statement cannot be initialized.	Remove the symbol from the DATA statement; recompile.
147	F	SYNTAX ERROR IN DATA LIST	An error appears in a DATA statement.	Correct error; recompile.

TABLE B-2. COMPILATION ERROR MESSAGES (Contd)

Error Number	Type	Message	Significance	Action
148	F	SUBSCRIPT MAY NOT BE AN EXPRESSION	An expression is used as a subscript.	Correct error; recompile.
149	F	TOO MANY SUBSCRIPTS	The array is declared to have fewer dimensions than there are subscripts.	Correct error; recompile.
150	F	SYNTAX ERROR IN HEXADECIMAL OR BIT CONSTANT	An error appears in a hexadecimal or bit constant.	Correct error; recompile.
151	F	ILLEGAL DATA ITEM		Correct error; recompile.
152	F	ILLEGAL VECTOR REFERENCE MODE IN DATA STATEMENT		Correct error; recompile.
153	F	CHARACTER, HEX OR BIT CONSTANT TOO LARGE	Constant is too large to be represented.	Reduce size of constant; recompile.
154	F	ILLEGAL USE OF VECTOR REFERENCE MODE IN DATA STATEMENT		Correct error; recompile.
155	W	TOO MANY DATA CONSTANTS	There are more values in a DATA statement than there are variables. The extra values are not used.	Verify that the proper number of variables and constants are specified.
156	F	SYNTAX ERROR	A language construct is written improperly.	Correct error; recompile.
157	F	SPECIFICATION STATEMENTS MUST PRECEDE STATEMENT FUNCTION DEFINITIONS	A specification statement appears after a statement function definition.	Move all specification statements in front of all statement function definitions; recompile.
158	F	ILLEGAL ELEMENT IN SPECIFICATION LIST		Correct error; recompile.
159	F	ILLEGAL OPERATOR IN SPECIFICATION		Correct error; recompile.
160	F	ILLEGAL LENGTH SPECIFICATION OF CHARACTER VARIABLE	The length specification that appears in a CHARACTER statement is illegal.	Correct error; recompile.
161	W	NAMELIST NAME IN TYPE STATEMENT	A type is given to a name listname; this action is ignored by the compiler.	Check user-defined names to find out if a name is used as both a namelist name and a variable or array name.
162	W	VARIABLE TYPED MORE THAN ONCE	The first type is used. The additional type specifications are ignored.	Verify that the first type is intended. Check user-defined names to find out if two different variables are intended.
163	F	LENGTH OF ADJUSTABLE CHARACTER MUST BE TYPE INTEGER	The length specification that appears in a CHARACTER statement is not an integer.	Correct error; recompile.
164	F	ZERO LENGTH FOR CHARACTER VARIABLE	The length specification for a character variable is zero.	Correct error; recompile.

TABLE B-2. COMPILATION ERROR MESSAGES (Contd)

Error Number	Type	Message	Significance	Action
165	F	ERROR IN DATA LIST OF TYPE STATEMENT		Correct error; recompile.
166	F	ILLEGAL STATEMENT ON LOGICAL IF	The consequent statement on a logical IF is not allowed.	Correct error; recompile.
167	W	NO LABELED COMMON IN BLOCK DATA SUBPROGRAM	No labeled common blocks are declared in the BLOCK DATA subprogram.	Verify that all state ments appear in the BLOCK DATA subprogram as intended.
168	F	ILLEGAL STATEMENT IN BLOCK DATA SUBPROGRAM	This statement cannot appear in a BLOCK DATA sub-program.	Correct error; recompile.
169	W	MAIN PROGRAM HAS NO EXECUTABLE STATEMENTS		Verify that all state-ments in the main program appear as intended.
170		(Currently unassigned)	--	--
171	W	END NOT PRECEDED BY BRANCH STATEMENT	A STOP statement was gen-erated by the compiler.	Verify that a STOP state-ment was intended.
172	W	FUNCTION NAME IS NOT DEFINED	A function returns a value through its name. The name must be assigned a value during execution of the function.	Check the function for a missing assignment statement.
173	W	NO RETURN STATEMENT	A RETURN statement was generated by the compiler.	Verify that a RETURN statement was intended.
174	F	ENTRY IN RANGE OF DO OR IN BLOCK IF	An ENTRY statement appears in the range of a DO loop or in a block IF.	Remove the ENTRY state-ment from the range of the DO loop or block IF; recompile.
175	F	NO ARGUMENTS FOR FUNCTION	The subprogram is compiled as a main program.	Supply the argument list for the FUNCTION state-ment; recompile.
176	F	ILLEGAL DUMMY ARGUMENT	An argument that appears in a FUNCTION or SUBROUTINE statement is illegal.	Correct error; recompile.
177	F	MISSING NAMELIST NAME	A NAMELIST statement does not contain a namelist name.	Supply the namelist name enclosed in slashes; recompile.
178	F	ILLEGAL NAMELIST NAME	A namelist name is illegal.	Correct error; recompile.
179	F	MISSING SLASH AFTER NAMELIST NAME	A namelist name must be enclosed in slashes.	Supply the missing slash after the namelist name; recompile.
180	F	LIST ITEM MUST BE A VARIABLE		Correct error; recompile.
181	F	ILLEGAL OPERATOR		Correct error; recompile.
182	F	ILLEGAL OR MISSING VARIABLE		Correct error; recompile.

TABLE B-2. COMPILATION ERROR MESSAGES (Contd)

Error Number	Type	Message	Significance	Action
183	F	SYNTAX ERROR IN LABEL STRING		Correct error; recompile.
184		(Currently unassigned)	--	--
185	F	INVALID LABEL REFERENCE		Correct error; recompile.
186	F	MORE THAN 253 COMMON BLOCK NAMES	Too many common blocks are used in the program.	Reduce the number of common blocks used; recompile.
187	F	ATTEMPTED TO RE-ORDER COMMON	COMMON and EQUIVALENCE statements conflict.	Correct error; recompile.
188	F	VARIABLE APPEARS IN COMMON MORE THAN ONCE	The same variable appears more than once in a common block.	Eliminate all but one occurrence of the variable from the COMMON statement; recompile.
189	F	ENTRY MUST BE IN A SUBROUTINE OR FUNCTION	An ENTRY statement appears in a main program or a BLOCK DATA subprogram.	Remove the ENTRY statement; recompile.
190	F	DUPLICATION OF DUMMY ARGUMENT NAMES	The same name appears more than once in the dummy argument list of a FUNCTION, SUBROUTINE, or ENTRY statement.	Eliminate all but one occurrence of the dummy argument from the argument list of the statement; recompile.
191	F	ILLEGAL DIMENSION SPECIFICATION		Correct error; recompile.
192	F	ILLEGAL FORMATION OF I/O STATEMENT		Correct error; recompile.
193	F	I/O UNIT MUST BE INTEGER CONSTANT OR INTEGER VARIABLE	A non-integer unit number appears in an input/output statement.	Change the unit number to an integer constant or an integer variable; recompile.
194	W	DUPLICATE OPTION IN I/O STATEMENT	The first option is used.	Verify that the first option is intended.
195	F	ILLEGAL OPTION IN I/O STATEMENT	The option specified cannot be used with the input/output statement.	Eliminate or change the option; recompile.
196	W	REFERENCED UNDEFINED FORMAT	The format specified in an input/output statement is not defined in the program.	Check for a missing FORMAT statement, or check for an error in the format number specified in the input/output statement..
197	F	RECORD LENGTH MUST BE INTEGER CONSTANT OR INTEGER VARIABLE	A non-integer record length is specified in an input/output statement.	Change the record length specification to an integer constant or an integer variable; recompile.
198	F	FORMAT REFERENCE MUST BE FORMAT STATEMENT NUMBER OR ARRAY NAME	The format reference in an input/output statement is not the label of a FORMAT statement or the name of an array.	Supply a format label or an array name to the input/output statement; recompile.

TABLE B-2. COMPILATION ERROR MESSAGES (Contd)

Error Number	Type	Message	Significance	Action
199	F	ILLEGAL ELEMENT IN I/O LIST		Correct error; recompile.
200	F	ILLEGAL OR MISSING DELIMITER IN I/O LIST	Elements in an input/output list must be delimited by commas.	Place commas between names in the input/output list; recompile.
201	F	ILLEGAL FORMATION OF REWIND, ENDFILE OR BACKSPACE		Correct error; recompile.
202	F	ILLEGAL FORMATION OF COMMON STATEMENT		Correct error; recompile.
203	F	COMMON BLOCK NAME IS NOT SYMBOLIC	An invalid symbol is specified as a common block name.	Supply a valid identifier as the name of the common block; recompile.
204	F	DUPLICATE SYMBOLIC NAME IN COMMON STATEMENT	The same symbol appears more than once in a COMMON statement.	Change the symbols so that all of the symbols in the COMMON statement are unique; recompile.
205	W	DATA SHOULD NOT BE PRESET IN BLANK COMMON	BLOCK DATA subprograms can be used to initialize data in named common blocks only.	Remove initialized variable from blank common or use executable statements to initialize it.
206	F	DUMMY ARGUMENT CANNOT APPEAR IN COMMON	A dummy argument appears in a COMMON statement.	Change the name of the dummy argument or change the name in the COMMON statement; recompile.
207	F	ILLEGAL USE OF VARIABLE OR VARIABLE DIMENSIONED MORE THAN ONCE		Correct error; recompile.
208	F	A VARIABLE IN A DIMENSION STATEMENT MUST BE DIMENSIONED DIMENSION	The dimension specification for a variable that appears in a DIMENSION statement is not specified.	Add the dimension specification to the variable name that appears in the statement; recompile.
209	F	MISSING COMMA	A comma is required.	Supply the comma; recompile.
210	F	DIMENSIONING FORMAT ERROR		Correct error; recompile.
211	F	ILLEGAL USE OF SUBSCRIPT		Correct error; recompile.
212	W	VARIABLE DIMENSION NEITHER A DUMMY ARGUMENT NOR IN COMMON YET	The variable used as the dimension specification is not in a preceding SUBROUTINE, FUNCTION, ENTRY, or COMMON statement.	Place the variable used as the dimension specification in the argument list of the FUNCTION or SUBROUTINE statement, or in a COMMON statement that appears before the statement in which the variable is used; recompile.
213	F	VARIABLE DIMENSION HAS TO BE A SIMPLE VARIABLE	The dimension specification in a DIMENSION statement is not a simple variable.	Change the dimension specification to a simple integer variable; recompile.
214	F	VARIABLE DIMENSION CANNOT BE DEFINED	An attempt is made to change a dimension for a variably dimensioned array.	Correct error; recompile.

TABLE B-2. COMPILATION ERROR MESSAGES (Contd)

Error Number	Type	Message	Significance	Action
215	F	MORE THAN 7 DIMENSIONS SPECIFIED	An array can have no more than 7 dimensions.	Reduce the number of dimensions; recompile.
216	F	CONSTANT GREATER THAN 2**18 IN SPECIFICATION STATEMENT	The constant is too large.	Reduce the value of the constant; recompile.
217	F	ILLEGAL OR MISSING REFERENCE IN DO STATEMENT		Correct error; recompile.
218	F	LABEL REFERENCE GREATER THAN 99999	A label can have no more than 5 digits.	Shorten the label to 5 digits. Correct all references to the label appropriately; recompile.
219	F	ILLEGAL PARAMETER IN DO STATEMENT		Correct error; recompile.
220	F	ILLEGAL OR MISSING DELIMITER	A delimiter is required.	Supply the delimiter; recompile.
221	F	DO LOOP NEVER TERMINATED	An END statement appears in the range of a DO loop.	Supply the last statement of the DO loop if it is missing, or move the END statement out of the DO loop; recompile.
222	F	A DO LOOP MAY NOT TERMINATE ON THIS STATEMENT	This statement cannot be the last statement in a DO loop.	Add a CONTINUE statement after this statement. Movethe label of this statement to the label field of the CONTINUE statement; recompile.
223	F	EQUIVALENCE FORMAT ERROR	The format of the EQUIVALENCE statement is incorrect.	Correct error; recompile.
224	F	ILLEGAL COMPONENT BEING EQUIVALENCED	The argument of the EQUIVALENCE statement is illegal.	Correct error; recompile.
225	F	ILLEGAL DELIMITER SEPARATING EQUIVALENCE GROUPS	Equivalence groups must be separated by commas.	Add commas between equivalence groups; recompile.
226	F	ARRAY ELEMENT MUST HAVE AT AT LEAST ONE SUBSCRIPT	An array name appears that does not have a subscript.	Supply the subscript; recompile.
227	F	ONLY SYMBOLIC NAMES CAN APPEAR IN EXTERNAL STATEMENTS	Something other than a symbolic name appears in an EXTERNAL statement.	Correct error; recompile.
228	F	EXTERNAL STATEMENT DID NOT PRECEDE REFERENCE OR VARIABLE IS WRONG TYPE		Correct error; recompile.
229	F	ILLEGAL USE OF NAME IN EXTERNAL STATEMENT		Correct error; recompile.
230	F	ILLEGAL EXPRESSION IN IF	An arithmetic IF statement must have an arithmetic or logical expression.	Correct error; recompile.
231	F	COMMA IS ONLY OPERATOR ALLOWED BETWEEN LABELS	An operator other than a comma was found between labels.	Change the operator to a comma; recompile.

TABLE B-2. COMPILATION ERROR MESSAGES (Contd)

Error Number	Type	Message	Significance	Action
232	F	SUBSCRIPT EXPRESSION NOT INTEGER, REAL, OR DOUBLE PRECISION	A subscript expression can be integer, real, or double-precision. The result of the expression is truncated to an integer.	Change the type of expression in the subscript to integer, real, or double-precision; recompile.
233	F	I/O SPECIAL EXIT PARAMETER MUST BE AN INTEGER VARIABLE		Correct error; recompile.
234	F	ITEMS IN COMMON MUST BE ARRAYS OR SIMPLE VARIABLES	Something other than an array or simple variable appears in a common block.	Remove the erroneous element from the COMMON statement; recompile.
235	F	DIRECT ACCESS I/O NOT IMPLEMENTED	Direct access input/output cannot be performed.	Replace the direct access input/output statement; recompile.
236	W	UNREFERENCED FORMAT	A FORMAT statement appears in a program, but is not referenced in an input/output statement.	Check the format references in all input/output statements to find out if the proper formats are specified.
237	F	NAMELIST IS USED ILLEGALLY		Correct error; recompile.
238	W	UNREFERENCED NAMELIST	A NAMELIST statement appears in the program, but it is not referenced in an input/output statement.	Check the namelist references in all input/output statements to find out if the proper namelists are specified.
239	F	ADJUSTABLE LENGTH IS NOT A DUMMY ARGUMENT OR IN COMMON	The variable used as the length of a character variable is not defined prior to its use.	Place the variable used as the length of a character variable in the argument list of the FUNCTION, SUBROUTINE, or ENTRY statement, or in a COMMON statement that appears before the statement in which the variable is used; recompile.
240	F	INCORRECT DO SPECIFICATION IN I/O LIST	The implied DO loop in an input/output statement is in error.	Correct error; recompile.
241	F	BUFFER MUST BE VARIABLE OR ARRAY OR SUBSCRIPTED VARIABLE	Bad buffer specification in buffer input/output statement.	Correct error; recompile.
242	F	EQUIVALENCE RELATION ERROR BETWEEN GROUPS	Equivalence declaration conflicts with other declarations.	Correct error; recompile.
243	F	NON-REDEFINABLE VARIABLE IN INPUT LIST	There is a variable in the input list whose value cannot be altered, such as a DO loop control variable.	Correct error; recompile.
244	F	ARRAY REFERENCED WITH WRONG NUMBER OF SUBSCRIPTS	The number of subscripts of an array reference is not the same as the number of subscripts declared in the specification statement for the array.	Check the specification statement for the array and correct the array reference appropriately; recompile.

TABLE B-2. COMPILATION ERROR MESSAGES (Contd)

Error Number	Type	Message	Significance	Action
245	W	CONSTANTS MAY BE TOO LARGE		
246	F	EQUIVALENCE HAS ATTEMPTED TO RE-ORIGIN COMMON	The EQUIVALENCE statement is incompatible with a COMMON statement. A common block cannot be extended at its beginning.	Correct the EQUIVALENCE statement so that it does not extend the common block at its beginning; recompile.
247	W	MISSING SUBSCRIPT - A ONE IS SUBSTITUTED	An array reference has missing subscript expressions or too few subscript expressions.	Correct error; recompile.
248	F	ILLEGAL COMPONENT IN I/O STATEMENT		Correct error; recompile.
249	F	ILLEGAL OR MISSING BUFFER SPECIFICATION	Bad buffer specification in buffer input/output statement.	Correct error; recompile.
250	W	RETURN STATEMENT IGNORED IN BLOCK DATA SUBPROGRAM	A BLOCK DATA subprogram does not permit a RETURN statement. The RETURN statement is ignored.	No action necessary.
251	W	RETURN STATEMENT REPLACED BY STOP STATEMENT IN MAIN PROGRAM	A main program requires a STOP statement rather than a RETURN statement. The RETURN statement is assumed to be a STOP statement.	No action necessary.
252	W	ILLEGAL PARAMETER IN RETURN STATEMENT	The parameter in the RETURN statement is ignored.	Verify that ignoring the parameter does not affect the logic of the program.
253	W	MODE OF RETURN PARAMETER MUST BE INTEGER	The parameter in a RETURN statement must be integer. The noninteger parameter is ignored.	Verify that ignoring the parameter does not affect the logic of the program.
254	W	ILLEGAL VALUE FOR RETURN STATEMENT	Value is greater than the number of alternate returns, or is not positive.	Correct error; recompile.
255	F	SYNTAX ERROR ON LEFT SIDE OF ASSIGNMENT STATEMENT	An illegal language construct appears to the left of the equals sign.	Correct error; recompile.
256	F	NON-REDEFINABLE VARIABLE ON LEFT SIDE OF ASSIGNMENT STATEMENT	The value of the variable that appears to the left of the equals sign cannot be changed.	Correct error; recompile.
257	F	ILLEGAL FIELD SPECIFICATION IN FORMAT		Correct error; recompile.
258	F	FORMAT STATEMENT IN BLOCK DATA SUBPROGRAM	A FORMAT statement cannot appear in a BLOCK DATA subprogram.	Remove the FORMAT statement from the BLOCK DATA subprogram; recompile.
259	F	LENGTH OF HOLLERITH FIELD OUT OF RANGE	The maximum length of a Hollerith field is 255 characters.	Reduce the length of the Hollerith field to no more than 255 characters; recompile.

TABLE B-2. COMPILATION ERROR MESSAGES (Contd)

Error Number	Type	Message	Significance	Action
260	F	END OF STATEMENT IN HOLLERITH FIELD	The Hollerith field extends beyond the end of the source statement.	Split the Hollerith field into two or more shorter fields and continue the statement on subsequent source lines; recompile.
261	F	MISSING CLOSING APOSTROPHE OR ASTERISK ON CHARACTER STRING	The character string must be delimited by apostrophes or asterisks.	Supply the missing apostrophe or asterisk; recompile.
262	F	ASSIGN MUST BE FOLLOWED EITHER BY A LABEL OR A DESCRIPTOR VARIABLE		Correct error; recompile.
263	F	ASSIGN VARIABLE MUST BE SIMPLE INTEGER VARIABLE		Correct error; recompile.
264	F	MISSING SUBSCRIPTS	An array name appears without subscripts.	Supply the subscripts; recompile.
265	F	MISSING LABEL(S) IN GO TO - POSSIBLE MIS-USE OF COMPUTED GO TO STATEMENT IN SOURCE	A computed GO TO statement must specify statement labels to which control can transfer depending on the condition.	Supply proper number of statement labels; recompile.
266	F	ILLEGAL LABEL VALUE IN ASSIGN STATEMENT	The specified label does not exist or is a FORMAT label.	Correct error; recompile.
267	F	ATTEMPT TO INITIALIZE CHARACTER VARIABLE WITH NON-CHARACTER DATA		Use character data to initialize character variables; recompile.
268	F	LOGICAL CONSTANT CAN NOT INITIALIZE OTHER TYPES	A logical constant can initialize a logical variable only.	Replace the logical constant with a constant of the appropriate type, or change the type specification of the variable being initialized to logical; recompile.
269	F	MISSING DATA	The list of variables in the DATA statement is longer than the list of constants.	Eliminate the excessive variables, or add more constants to the DATA statement; recompile.
270	F	FLOATING POINT NUMBER OUT OF ALLOWABLE RANGE	A real constant is too small or too large to be represented.	Correct error; recompile.
271	F	MODE MUST BE INTEGER CONSTANT OR INTEGER VARIABLE	A noninteger number is used where an integer or an integer variable is required.	Change the number to an integer; recompile.
272		(Currently unassigned)	--	--
273	W	MISSING END STATEMENT	The compiler supplied an END statement.	No action necessary.
274	F	ARRAY DECLARATOR NOT A VARIABLE		Correct error; recompile.

TABLE B-2. COMPILATION ERROR MESSAGES (Contd)

Error Number	Type	Message	Significance	Action
275	F	VARIABLE CANNOT BE DIMENSIONED		Correct error; recompile.
276	F	ATTEMPT TO RE-DIMENSION A VARIABLE	The variable is already dimensioned.	Eliminate one of the dimension specifications or change the variable name; recompile.
277	F	PROGRAM STARTS WITH A CONTINUATION CARD	The first statement of a program has a nonzero, nonblank character in column 6.	Supply the source statements that are missing from the beginning of the program; recompile.
278	F	SUBSCRIPT OR DIMENSION CANNOT BE ZERO OR NEGATIVE	A zero or a negative number is used as a subscript.	Replace the subscript with a positive number; recompile.
279	F	ARRAY HAS TO BE FORMAL ARGUMENT TO HAVE VARIABLE DIMENSION	The array name that has a variable dimension is not in the formal argument list of the FUNCTION or SUBROUTINE statement.	Place the array name in the argument list of the FUNCTION or SUBROUTINE statement. Correct all subprogram references appropriately; recompile.
280	F	VARIABLE DIMENSION SHOULD BE SIMPLE INTEGER VARIABLE	The variable dimension specified is not a simple integer variable.	Replace the variable dimension with a simple integer variable; recompile.
281	F	LOGICAL VARIABLE INITIALIZED INCORRECTLY		Correct error; recompile.
282	F	BOTH VARIABLE LENGTH SPECIFIER AND CHARACTER VARIABLE MUST BE DUMMY ARGUMENTS		Correct error; recompile.
283	W	EQUIVALENCE VARIABLE ATTEMPTED TO BE ASSIGNED TO IMPROPER BOUNDARY	An EQUIVALENCE statement attempted to assign a logical, integer, real, double-precision, or complex variable to a nonword boundary, or a character variable to a nonbyte boundary.	Correct error; recompile.
284	F	ILLEGAL ELEMENT IN ARGUMENT VECTOR	Illegal branch into DO loop.	Correct error; recompile.
285	F	DO LOOP IS BRANCHED INTO, BUT HAS NO EXTENDED RANGE		Correct error; recompile.
286	F	ILLEGAL TRANSFER INTO RANGE OF DO LOOP	A statement causes a transfer into a DO loop.	Restructure the program so that control does not transfer into the range of a DO loop. Control can transfer to the DO statement; recompile.
287	F	REFERENCE TO UNDEFINED LABEL	A label is referenced, but it does not appear in the label field of any statement in the program.	Change the label reference so that it references a label that exists in the program, or supply the missing label in the program; recompile.
288	W	ILLEGAL EXPONENTIATION	Illegal operands for exponentiation.	Correct error; recompile.

TABLE B-2. COMPILATION ERROR MESSAGES (Contd)

Error Number	Type	Message	Significance	Action
289	F	SUBROUTINE CANNOT BE CALLED AS FUNCTION	A subroutine must be called with the CALL statement.	Change the statement that contains the function reference to a CALL statement; recompile.
290	W	FORMAT NOT LABELED	A FORMAT statement requires a label in the label field. The unlabeled FORMAT statement is not used.	Verify that the FORMAT statement is not referenced in the program.
291	F	ILLEGAL MODE FOR A LENGTH EXPRESSION IN A VECTOR REFERENCE	The length must be integer.	Correct error; recompile.
292	F	VECTOR LENGTH CANNOT BE A NEGATIVE CONSTANT		Correct error; recompile.
293	F	MODE ERROR IN A VECTOR ARITHMETIC OR BIT ASSIGNMENT STATEMENT		Correct error; recompile.
294	F	ILLEGAL MODE IN A VECTOR EXPRESSION		Correct error; recompile.
295	F	VECTOR EXPRESSION ASSIGNED TO A NON-VECTOR VARIABLE	A vector must be on the left side of a vector assignment statement.	Replace the variable on the left of the vector assignment statement with a vector; recompile.
296	F	SUBSCRIPT REFERENCE FOR NON-DIMENSIONED ARRAY	A subscript is specified for a variable that is not dimensioned.	Use a DIMENSION statement to dimension the variable, or remove the subscript from the variable reference; recompile.
297	F	DESCRIPTOR NOT INITIALIZED BY VECTOR REFERENCE		Correct error; recompile.
298	W	COMMON BLOCK HAS BEEN PADDED IN ORDER TO ENSURE ALIGNMENT	Alignment of the common block is performed by the compiler to place a character variable on a byte boundary or other variables (except bit) on a word boundary.	No action necessary.
299	F	FIRST AND LAST MUST BE VARIABLES OR ARRAY ELEMENTS	Illegal specification for first or last location in BUFFER IN or BUFFER OUT statement.	Correct error; recompile.
300	W	EXTRANEOUS INFORMATION AT END OF STATEMENT	The compiler ignored the extra information at the end of the statement.	Verify that the compiler interpreted the statement correctly.
301	F	STATEMENT CANNOT BE IDENTIFIED	Syntax error in statement.	Correct error; recompile.
302	F	A LABEL MUST BE AN INTEGER CONSTANT	A label is specified that is something other than an integer constant.	Change the label to an integer constant. Correct all references to the label appropriately; recompile.
303	F	DIGIT STRING EXCEEDS MAXIMUM OF FIVE	No more than 5 digits can appear in the digit string.	Reduce the string to 5 digits; recompile.

TABLE B-2. COMPILATION ERROR MESSAGES (Contd)

Error Number	Type	Message	Significance	Action
304	F	ILLEGAL CHARACTER	A character is used that is not in the CYBER 200 FORTRAN character set.	Replace the character with the appropriate character from the CYBER 200 FORTRAN character set; recompile.
305	W	ILLEGAL CONSTANT ON A PAUSE OR STOP	The constant is ignored.	Verify that the constant is not intended.
306	F	ILLEGAL CONSTANT TYPE		Correct error; recompile.
307	F	CHARACTER STRING EXCEEDS 255	No more than 255 characters can appear in a character string.	Reduce the character string to no more than 255 characters; recompile.
308	F	HOLLERITH FIELD COUNT IS TOO LARGE	Too many characters are in a Hollerith field. No more than 255 characters can appear in a Hollerith field.	Reduce the Hollerith field to no more than 255 characters; recompile.
309	F	SYMBOLIC NAME HAS MORE THAN 8 CHARACTERS	A symbolic name can consist of no more than 8 characters.	Reduce the symbolic name to no more than 8 characters; recompile.
310	F	COMPONENT HAS MORE THAN 255 CHARACTERS	No more than 255 characters are allowed.	Reduce the component to no more than 255 characters; recompile.
311	F	REAL NUMBER HAS MORE THAN 255 DIGITS	A real number can contain no more than 255 digits.	Reduce the real number to no more than 255 digits; recompile.
312	F	LOGICAL CONSTANT OR LOGICAL/ RELATIONAL OPERATOR IS INCORRECT		Correct error; recompile.
313	F	ERROR IN HOLLERITH COUNT		Correct error; recompile.
314	F	REAL NUMBER CANNOT BE FOLLOWED BY A LETTER		Correct error; recompile.
315	F	COMPLEX NUMBER COMPONENTS CANNOT BE DOUBLE PRECISION	A complex number can consist of real components only.	Change the double-precision components of the complex number to real; recompile.
316	F	MISSING RIGHT PARENTHESIS	A right parenthesis is required.	Supply the right parenthesis; recompile.
317	F	SYNTAX ERROR IN A COMPLEX CONSTANT		Correct error; recompile.
318	F	ZERO LENGTH CHARACTER STRING	The length of a character string is specified to be zero.	Change the zero to a positive integer; recompile.
319	F	ILLEGAL ARGUMENT FIELD SYNTAX		Correct error; recompile.
320	W	IMPLICIT STATEMENT MUST BE FIRST SPECIFICATION STATEMENT	Other statements appear before an IMPLICIT statement. The IMPLICIT statement is ignored.	Verify that ignoring the IMPLICIT statement does not affect the logic of the program.

TABLE B-2. COMPILATION ERROR MESSAGES (Contd)

Error Number	Type	Message	Significance	Action
321	F	ILLEGAL TYPE IN IMPLICIT STATEMENT	The valid types are INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, BIT, and CHARACTER.	Correct error; recompile.
322	F	ILLEGAL USE OF *		Correct error; recompile.
323	F	IMPLICIT RANGE IS INCORRECT	The characters specified in the range of an IMPLICIT statement must be in alphabetical order. A character cannot be associated with more than one type.	Arrange the characters in alphabetical order and eliminate duplicate specifications for characters; recompile.
324	F	NON-FORTRAN CHARACTER FOUND AND IS NOT IN HOLLERITH CHARACTER STRING	A character is used that is not in the CYBER 200 FORTRAN character set. These characters can be used only in Hollerith strings.	Replace the character with the appropriate character from the CYBER 200 FORTRAN character set; recompile.
325	F	SYNTAX ERROR AFTER A SYMBOLIC NAME		Correct error; recompile.
326	F	ILLEGAL CHARACTER AFTER A ZERO		Correct error; recompile.
327	F	SYNTAX ERROR AFTER AN INTEGER CONSTANT		Correct error; recompile.
328	F	SYNTAX ERROR FOLLOWING A PERIOD		Correct error; recompile.
329	F	ILLEGAL CHARACTER IN A LOGICAL CONSTANT OR LOGICAL/RELATIONAL OPERATOR		Correct error; recompile.
330	F	SYNTAX ERROR AFTER A REAL NUMBER		Correct error; recompile.
331	F	ILLEGAL CHARACTER APPEARS IN THE NUMBER PART OF THE EXPONENT FIELD		Correct error; recompile.
332	W	TOO MANY DIGITS IN THE EXPONENT FIELD	The exponent field is truncated.	Verify that the truncation does not affect the logic of the program.
333	F	SYNTAX ERROR FOLLOWING A SYMBOLIC STRING THAT WAS FOLLOWED BY A PERIOD		Correct error; recompile.
334	F	SYNTAX ERROR FOLLOWING A LOGICAL CONSTANT		Correct error; recompile.
335	F	SYNTAX ERROR FOLLOWING A REAL CONSTANT		Correct error; recompile.
336	F	SYNTAX ERROR FOLLOWING AN *		Correct error; recompile.
337	F	SYNTAX ERROR FOLLOWING A CHARACTER STRING		Correct error; recompile.
338	F	SYNTAX ERROR FOLLOWING A COMPLEX CONSTANT		Correct error; recompile.

TABLE B-2. COMPILATION ERROR MESSAGES (Contd)

Error Number	Type	Message	Significance	Action
339	F	SYNTAX ERROR IN A LABEL REFERENCE FIELD		Correct error; recompile.
340	W	SUBSCRIPT REFERENCE OUT OF RANGE	The subscript is less than 1 or greater than the upper bound of the array.	Verify that the reference is intended.
341	F	DO LOOPS OR IF BLOCKS NESTED ILLEGALLY	Nested DO loops and IF blocks must appear entirely within outer DO loops and IF blocks.	Restructure the DO loops and IF blocks so that the nested DO loops are entirely within the outer DO loops and IF blocks; recompile.
342	F	INDUCTION VARIABLE USED ILLEGALLY	The variable used as the loop index cannot be altered within the range of the DO loop.	Remove all statements that alter the value of the loop index from the DO loop; recompile.
343	F	ILLEGAL DO STRUCTURE		Correct error; recompile.
344	F	IMPLIED DO STRUCTURES DO NOT MATCH		Correct error; recompile.
345	F	ILLEGAL ARGUMENT		Correct error; recompile.
346	F	MISSING OUTPUT ARGUMENT IN A VECTOR FUNCTION REFERENCE	A vector function reference must have an output argument, which is preceded by a semicolon.	Correct error; recompile.
347	F	OUTPUT ARGUMENT NOT ALLOWED IN SCALAR FUNCTION REFERENCE		Correct error; recompile.
348	F	FUNCTION CANNOT BE REFERENCED AS BOTH A SCALAR AND A VECTOR FUNCTION		Correct error; recompile.
349	F	MODE OF OUTPUT ARGUMENT AND MODE OF FUNCTION NAME DO NOT MATCH		Correct error; recompile.
350	F	ILLEGAL OUTPUT ARGUMENT IN A FUNCTION REFERENCE		Correct error; recompile.
351	F	VECTOR EXPRESSION REQUIRES MORE TEMPORARIES, CODE CANNOT BE GENERATED	Compiler limitation exceeded.	Simplify statement; recompile.
352	F	VECTOR REFERENCE DATA ITEM USED FOR NON-DESCRIPTOR VARIABLE ITEM		Correct error; recompile.
353	F	CHARACTER CONSTANT CANNOT INITIALIZE A BIT VARIABLE	DATA statement cannot have character string as a value for a bit variable.	Correct error; recompile.
354	F	ILLEGAL INITIALIZATION OF A CHARACTER OR BIT VARIABLE		Correct error; recompile.
355	W	CHARACTER CONSTANT TOO LONG - TRUNCATED ON THE RHS	The character constant is truncated on the right side. A character constant can contain no more than 255 characters.	Verify that the truncation does not affect the logic of the program.

TABLE B-2. COMPILATION ERROR MESSAGES (Contd)

Error Number	Type	Message	Significance	Action
356	W	HEX OR BIT CONSTANT TOO LONG - TRUNCATED ON THE LHS	The hexadecimal or bit constant is truncated on the left side. A hexadecimal or bit constant can contain no more than 255 characters.	Verify that the truncation does not affect the logic of the program.
357	F	BIT VARIABLES ARE NOT ALLOWED IN BUFFER IN/OUT		Correct error; recompile. Equivalence the bit variable to nonbit variables, and perform the input/output on the nonbit variables.
358	F	DESCRIPTOR INITIALIZATION ILLEGAL		Correct error; recompile.
359	W	VECTOR MODE CHANGED TO BE SAME AS DESCRIPTOR IT INITIALIZES		Verify that this change does not affect the logic of the program.
360	W	OPTIMIZATION TURNED OFF BECAUSE SOURCE PROGRAM IS NOT ADVANTAGEOUS FOR ITS IMPLEMENTATION	The compiler stopped optimizing the program.	No action necessary.
361		(Currently unassigned)	--	--
362	F	ILLEGAL RIGHT-HAND SIDE FOR DESCRIPTOR ASSIGN		Correct error; recompile.
363		(Currently unassigned)	--	--
364	F	+-*/ ARE THE ONLY LEGAL OPERATORS FOR COMPLEX VECTORS		Correct error; recompile.
365		(Currently unassigned)	--	--
366	F	ILLEGAL SUBSCRIPT IN IMPLIED DO		Correct error; recompile.
367	F	TOO MANY LEFT PARENTHESES IN EXPRESSION	There are more left parentheses in the expression than there are right parentheses.	Match the parentheses properly; recompile.
368	F	TOO MANY RIGHT PARENTHESES IN EXPRESSION	There are more right parentheses in the expression than there are left parentheses.	Match the parentheses properly; recompile.
369	F	VARIABLE APPEARS IN DESCRIPTOR STATEMENT MORE THAN ONCE		Correct error; recompile.
370	F	MISSING LABEL IN ARITHMETIC IF	An arithmetic IF must have three labels to which control can transfer depending on the condition. Labels can be duplicated.	Supply the missing label in the arithmetic IF statement; recompile.
371	A	JAM TEMP TABLE OVERFLOW	Compilation aborted.	Recompile without instruction scheduling.
372	F	INDEX PARAMETER FOR IMPLIED DO SUBARRAY REFERENCE CANNOT BE EXPRESSION		Correct error; recompile.

TABLE B-2. COMPILATION ERROR MESSAGES (Contd)

Error Number	Type	Message	Significance	Action
373	F	ILLEGAL SYMBOL IN EQUIVALENCE		Correct error; recompile.
374	F	TOO LITTLE DATA IN HEX OR BIT CONSTANT	The length of the bit constant must equal the length of the portion of the bit array being initialized.	Increase the length of the bit constant to the appropriate size; recompile.
375	F	TOO MUCH DATA IN HEX OR BIT CONSTANT	The length of the bit constant must equal the length of the portion of the bit array being initialized.	Decrease the length of the bit constant to the appropriate size; recompile.
376	F	EVEN/ODD REGISTER PAIR REQUIRED FOR C AND C+1 FIELDS	Incorrect use of registers in special call.	Correct error; recompile.
377	W	INSTRUCTION SCHEDULING ABANDONED - REGISTER JAM	Compiler was unable to optimize instruction scheduling.	No action necessary; object code is generated.
378	F	THE COMMON BLOCK NAME AND AN ENTRY NAME ARE THE SAME	The name of a common block and the name of an entry point are the same.	Change one of the names; recompile.
379	F	SCALAR ARGUMENTS NOT ALLOWED IN Q8SDOT		Correct error; recompile.
380	W	RELATIVE BRANCH OUT OF RANGE	Branch too far in special call.	Correct error; recompile.
381	F	A SPECIAL CALL RELATIVE BRANCH MAY ONLY BRANCH TO A STATEMENT LABEL	Branching a constant number of halfwords is not permitted.	Correct error; recompile.
382	F	NON-ZERO OPERAND IN SPECIAL CALL FIELD THAT MUST BE NULL OR ZERO	Arguments are missing or in the wrong order.	Correct error; recompile.
383	W	HOLLERITH CONSTANT TOO LONG - TRUNCATED ON RHS	The Hollerith constant is truncated on the right side. A Hollerith constant can have no more than 255 characters.	Verify that the truncation does not affect the logic of the program.
384	F	REPEAT COUNT CANNOT EXCEED 255	The repeat count is more than 255.	Reduce the repeat count to no more than 255; recompile.
385	F	SUBROUTINE CONTAINS NON-STANDARD RETURN BUT NO * IN ARGUMENT LIST	Asterisks must appear in the argument list of the SUBROUTINE statement. Each asterisk must correspond to a statement label that appears in the argument list of the subroutine CALL statement.	Place asterisks in the appropriate positions in the argument list of the SUBROUTINE statement. Place statement labels in the appropriate positions in the CALL statements; recompile.
386	A	COMPILER FAILURE - IRRESOLVABLE REGISTER JAM	Compilation aborted.	Recompile without optimization.
387	W	R CONSTANT TOO LONG - TRUNCATED ON RHS	The R constant is truncated on the right side. An R constant can have no more than 255 characters.	Verify that the truncation does not affect the logic of the program.

TABLE B-2. COMPILATION ERROR MESSAGES (Contd)

Error Number	Type	Message	Significance	Action
388	F	HOLLERITH CONSTANT NOT PERMITTED IN SPECIAL CALL		Remove the Hollerith constant from the special call; recompile.
389	W	ABOVE ERROR MAY BE IN STATEMENT FUNCTION DEFINITION	Error might be in statement function definition.	Correct error; recompile.
390	F	LIST-DIRECTED I/O NOT IMPLEMENTED	List-directed input/output statements cannot be used.	Remove list-directed input/output statements from the program; recompile.
391	F	DUMMY ARGUMENT MAY NOT APPEAR IN EQUIVALENCE		Correct error; recompile.
392	F	MISSING SYMBOLIC NAME		Supply the symbolic name; recompile.
393	F	MISSING =	Equals sign missing from PARAMETER statement.	Supply the equals sign; recompile.
394	W	SYMBOLIC CONSTANT NAME PREVIOUSLY DECLARED	A symbolic constant is declared more than once. The first declaration was used.	Verify that the first declaration is intended.
395	F	SYMBOLIC CONSTANT PREVIOUSLY USED FOR SOMETHING ELSE	A symbolic constant must not be the same as another symbol in the program.	Change the symbolic constant so that it is unique in the program; recompile.
396	F	VALUE MUST BE CONSTANT OR CONSTANT EXPRESSION	The value specified for a symbolic constant is not a constant.	Change the value to a constant or a constant expression; recompile.
397	F	INCOMPATIBLE MODES FOR SYMBOLIC NAME AND ITS VALUE		Correct error; recompile.
398	W	PARAMETER STATEMENTS MUST PRECEDE DATA STATEMENTS		Move the PARAMETER statementin front of the DATA statement; recompile.
399	W	PARAMETER STATEMENTS MUST PRECEDE STATEMENT FUNCTION DEFINITIONS		Move the PARAMETER statement in front of the statement function definitions; recompile.
400	W	PARAMETER STATEMENTS MUST PRECEDE EXECUTABLE STATEMENTS		Move the PARAMETER statement in front of all executable statements; recompile.
401	F	MISUSE OF SYMBOLIC CONSTANT NAME	A symbolic constant can be used like any other constant, except it cannot appear in a complex constant, in a FORMAT statement, orin a PROGRAM statement. Also, it cannot appear as input data.	Correct error; recompile.
402	F	STATEMENT NOT YET IMPLEMENTED	The statement cannot be used.	Eliminate the statement; recompile.

TABLE B-2. COMPILATION ERROR MESSAGES (Contd)

Error Number	Type	Message	Significance	Action
403	F	A SYMBOLIC CONSTANT MAY NOT BE TYPED AFTER ITS DECLARATION	A type statement for a symbolic constant must appear before its declaration in the PARAMETER statement.	Move the type statement in front of the PARAMETER statement that defines the symbolic constant; recompile.
404	F	DUPLICATE OR CONFLICTING IMPLICIT TYPE	A letter must not be assigned more than one implicit type.	Correct error; recompile.
405	W	ILLEGAL INSTRUCTION FOR TARGET MACHINE	The program cannot be correctly executed on the machine for which it is compiled.	Verify that the correct target machine is specified in the FORTRAN control statement.
406	F	ILLEGAL BLOCK IF NESTING	A nested block IF must be entirely contained in an outer block IF.	Correct error; recompile.
407	W	FUNCTION NOT AVAILABLE ON TARGET MACHINE	The program cannot be correctly executed on the machine for which it is compiled.	Verify that the correct target machine is specified in the FORTRAN control statement.
408	F	BRANCH INTO BLOCK IF	Control cannot transfer into an if-block from outside that if-block.	Rewrite the statement so that it does not transfer control into an if-block; recompile.
409	F	MISSING ENDIF	Each block IF statement must have a corresponding END IF statement.	Supply the missing END IF statement; recompile.
410		(Currently unassigned)	--	--
411	W	MISSING THEN IN ELSE IF STATEMENT	The keyword THEN must follow the keyword ELSE IF.	Supply the missing THEN.

entered with the program to be executed. The termination value is used to determine when error exit processing is to occur. All return codes having a value less than or equal to the termination value are ignored and job processing continues. All return codes having a value greater than the termination value cause error processing specified by the EXIT control statement to take place.

For example, a termination value of 8 would allow all warning and fatal errors to be ignored, and cause error exit processing to occur for abort errors. A termination value of 0 would trap all errors, including warning codes. The termination value control statement is discussed in the Operating System reference manual.

RUN-TIME ERRORS

Error messages listed in table B-3 are produced when error conditions are detected during the execution of a previously compiled program. The system error processor (SEP) can be called upon to change the attributes of certain run-time errors. Run-time error types are:

W (warning)	Nonfatal error. A warning is issued and execution continues. The return code is 4 (RC=4).
F (fatal)	Execution is terminated abnormally when this error condition exists. The return code is 8 (RC=8).
C (catastrophic)	Condition is nonalterable by SEP and not subject to user control, other than replacement of the standard message. The return code is 8 (RC=8).

All errors having a warning classification can be made fatal. Those errors which are designated as fatal can be altered to warning level. Catastrophic errors cannot be altered to fatal or warning level; however, the standard message can be replaced.

Error messages for mathematical routines have the CYBER 200 FORTRAN library function name appended to the message. In like manner, input/output error messages have the file name appended to the message.

The form of a run-time error message is:

ERROR xxx IN subr AT LINE nn

TABLE B-3. RUN-TIME ERRORS

Error Number	Type	Message	Significance	Action
1	C	SYNTAX ERROR IN PROGRAM STATEMENT FILE DECLARATION	A compilation error exists in the PROGRAM statement.	Correct compilation error. Rerun.
2	C	UNIT NUMBER IS MULTIPLY DEFINED IN PROGRAM STATEMENT	The same unit number is assigned to more than one file.	Change the PROGRAM statement so that each unit number is assigned to only one file. Correct all references to unit numbers accordingly. Rerun.
3	C	RUNTIME TABLE ERROR OVERFLOW		
4	C	ERROR IN CREATE FILE		
5	C	ERROR IN OPEN FILE		
6	C	MAXIMUM NUMBER OF FILES (70) EXCEEDED	No more than 70 files can be used in a program.	Reduce the number of files to no more than 70. Rerun.
7	C	SYSTEM ERROR IN CLOSE FILE		
8		(Currently unassigned)	--	--
9		(Currently unassigned)	--	--
10		(Currently unassigned)	--	--
11	C	FILE NOT LARGE ENOUGH FOR OUTPUT	The amount of output to a file exceeds the capacity of the file.	Increase the size of the file or reduce the amount of output to the file. Rerun.
12		(Currently unassigned)	--	--
13	C	END OF FILE IN INPUT STREAM -- file name	An input statement attempted to read data from the file indicated, but that file is positioned at the end of the file.	Use a REWIND or BACKSPACE statement to reposition the file before the input statement is executed, or supply missing data on the input file. Rerun.
14	F	A CALL TO Q8WIDTH MUST PRECEDE THE ACCESS TO A FILE		Call Q8WIDTH before first file access. Rerun.
15	F	TRANSMISSION ERROR DURING READ		
16	C	ILLEGAL I/O UNIT NUMBER	Unit numbers can be integers from 1 through 99.	Change the unit number to an integer from 1 through 99. Rerun.
17	C	ATTEMPT TO PERFORM SEQUENTIAL FORMATTED I/O ON A FILE OPENED FOR ANOTHER FORM OF I/O		Use the proper type of input/output statements, or open the file for sequential formatted input/output. Rerun.

TABLE B-3. RUN-TIME ERRORS (Contd)

Error Number	Type	Message	Significance	Action
18	C	ATTEMPT TO PERFORM SEQUENTIAL BINARY I/O ON A FILE OPENED FOR ANOTHER FORM OF I/O		Use the proper type of input/output statements, or open the file for sequential binary input/output. Rerun.
19	C	DIRECT ACCESS I/O NOT IMPLEMENTED	Direct access input/output cannot be used.	Eliminate direct access input/output statements. Rerun.
20	C	END OF FILE DURING BINARY INPUT	The end of a binary file was encountered during execution of an input statement.	Supply any data that is missing from the file, or use the END option input in the statement in order to continue execution after encountering an end of file condition.
21	C	ERROR DURING BINARY READ	An error occurred during execution of a binary input statement.	Use the ERR option in the input statement in order to continue execution after an input/output error occurs.
22	C	FILE PREVIOUSLY USED FOR BUFFER I/O	Mode of variable and format specification are incompatible.	
23	F	CHARACTER MODE, CONVERSION CODE F	Mode of variable and format specification are incompatible.	
24	F	CHARACTER MODE, CONVERSION CODE E	Mode of variable and format specification are incompatible.	
25	F	LOGICAL MODE, CONVERSION CODE D	Mode of variable and format specification are incompatible.	
26	F	INTEGER MODE, CONVERSION CODE D	Mode of variable and format specification are incompatible.	
27	F	REAL MODE, CONVERSION CODE D	Mode of variable and format specification are incompatible.	
28	F	COMPLEX MODE, CONVERSION CODE D	Mode of variable and format specification are incompatible.	
29	F	CHARACTER MODE, CONVERSION CODE D	Mode of variable and format specification are incompatible.	
30	F	G FORMAT SHOULD NOT BE SEEN BY LIST HANDLER	Mode of variable and format specification are incompatible.	
31	F	CHARACTER MODE, CONVERSION CODE I	Mode of variable and format specification are incompatible.	

TABLE B-3. RUN-TIME ERRORS (Contd)

Error Number	Type	Message	Significance	Action
32	F	CHARACTER MODE, CONVERSION CODE L	Mode of variable and format specification are incompatible.	
33	C	COMPILER FAILURE - ILLEGAL DATA IN TRANSLATED FORMAT STRING		
34		(Currently unassigned)	--	--
35	F	FORMAT ERROR		
36	F	ILLEGAL SYNTAX IN VARIABLE ARRAY FORMAT		
37	F	ILLEGAL HOLLERITH FIELD LENGTH	A Hollerith field can contain no more than 255 characters.	Reduce the length of the Hollerith field to no more than 255 characters. Rerun.
38	F	END OF STATEMENT ENCOUNTERED IN HOLLERITH FIELD	The length of a Hollerith field is longer than the source statement.	Reduce the length of the Hollerith field so that it is not longer than the length of the source statement.
39	F	FIELD COUNT OUT OF RANGE		
40	F	MISSING CLOSING APOSTROPHE OR ASTERISK AT END OF LITERAL STRING	A literal string must be delimited by apostrophes or asterisks.	Supply missing apostrophe or asterisk. Rerun.
41	F	IN F, E, D, GW.D; D IS GREATER THAN W	In these format specifications, the w field specifies the total length of the field; the d field specifies the number of spaces to the right of the decimal point. The d field must be less than the w field.	Rewrite the format specification so that the d field is less than the w field. Rerun.
42	F	UNDECLARED VARIABLE NAME ENCOUNTERED IN NAMELIST INPUT	The namelist input data contains a variable that is not declared in the NAMELIST statement of the program. All variables that are in the input data must be declared in the NAMELIST statement; all variables in the NAMELIST statement do not have to appear in the input data, however.	Place the variable in the NAMELIST statement or remove the variable from the input data. Rerun.
43	F	SUBSCRIPT ERROR IN NAMELIST INPUT		
44	F	FIRST COLUMN MUST BE BLANK FOR NAMELIST INPUT	The first column of each record of namelist input data must be blank.	Add a blank to the first column of each record of namelist input data. Rerun.
45	F	FORMAT ERROR IN NAMELIST INPUT DATA		Correct error. Rerun.

TABLE B-3. RUN-TIME ERRORS (Contd)

Error Number	Type	Message	Significance	Action
46	F	NAMELIST INPUT RECORD NOT PROPERLY TERMINATED	If a data item extends over more than one record, then variables, constants, array names, constants with repeat specifications, and the &END cannot extend over more than one record.	Correct error. Rerun.
47	F	NAMELIST - SCALAR VARIABLE CANNOT HAVE SUBSCRIPTS	A subscript is specified for a scalar variable in the namelist input data.	Remove the subscript or dimension the variable in the program. Rerun.
48	F	NUMBER OF ELEMENTS EXCEEDS DEFINED ARRAY LENGTH IN NAMELIST	An array in the namelist input data contains more elements than specified in the program.	Increase the size of the array in the program, or eliminate the extra array elements in the namelist input data. Rerun.
49	F	NAMELIST CHARACTER IS TOO LONG		
50	W	INDEFINITE VALUE IN NAMELIST OUTPUT	A variable that has an indefinite value is output using a namelist output statement.	Check the program for the source of the indefinite value.
51	W	LOGICAL DATA SYNTAX ERROR	Misspelled .TRUE. or .FALSE.	
52	W	COMPUTATIONAL ERROR - LOGICAL VALUE MUST BE EQUAL TO 0 OR 1		
53	W	DATA EXCEEDS (2**47)-1	A value in the input data is too large to be represented. The value was truncated.	Verify that the truncation does not affect the logic of the program.
54	W	ILLEGAL DATA IN FIELD		
55	W	INVALID DATA IN FIELD		
56	W	DATA OVERFLOW		
57	W	INDEFINITE ARGUMENT	The argument has an indefinite value. This can cause subsequent errors.	Check program for the source of the indefinite value.
58	W	ZERO TO THE ZERO POWER	Zero can be raised only to a positive power. This expression yields an indefinite result. The indefinite value can cause subsequent errors.	Check to find out if this indefinite value causes subsequent errors.
59	W	ZERO TO THE NEGATIVE POWER	Zero can be raised only to a positive power. This expression yields an indefinite result. The indefinite value can cause subsequent errors.	Check to find out if this indefinite value causes subsequent errors.
60	W	FLOATING POINT OVERFLOW	A real number is too large to be represented. The number was truncated.	Verify that the truncation does not affect the logic of the program.

TABLE B-3. RUN-TIME ERRORS (Contd)

Error Number	Type	Message	Significance	Action
61	W	ARGUMENT TOO LARGE - ACCURACY LOST	The value of the argument is too large to be represented precisely.	Verify that the loss of accuracy does not affect the logic of the program.
62	W	INTEGER OVERFLOW	An integer is too large to be represented. The value was truncated.	Verify that the truncation does not affect the logic of the program.
63	W	NEGATIVE TO THE REAL POWER	A negative value can have an integer exponent only. The exponent was truncated to an integer before the exponentiation was performed.	Verify that the truncation does not affect the logic of the program.
64	W	ZERO ARGUMENT		
65	W	NEGATIVE ARGUMENT		
66	W	X = Y = 0.0		
67	W	ABS (REAL PART) TOO LARGE	The real part of a complex number is too large or too small to be represented. The real part was truncated.	Verify that the truncation does not affect the logic of the program.
68	W	IMAG. PART TOO LARGE	The imaginary part of a complex number is too large to be represented. The imaginary part was truncated.	Verify that the truncation does not affect the logic of the program.
69	W	REAL PART TOO LARGE	The real part of a complex number is too large to be represented. The real part was truncated.	Verify that the truncation does not affect the logic of the program.
70	W	ABS (IMAG PART) TOO LARGE	The imaginary part of a complex number is too large or too small to be represented. The imaginary part was truncated.	Verify that the truncation does not affect the logic of the program.
71	W	ARGUMENT TOO LARGE, FLOATING POINT OVERFLOW	The value of the argument is too large and caused a floating-point overflow. The value was truncated.	Verify that the truncation does not affect the logic of the program.
72	W	INDEFINITE RESULT	An expression resulted in an indefinite result. The indefinite value can cause subsequent errors.	Check to find out if the indefinite value causes subsequent errors.
73	W	NEGATIVE TO A POWER	A negative number is raised to an exponent. The exponent must be an integer.	Verify that the exponent is an integer.
74	W	ARGUMENT TOO LARGE	The value of an argument is too large to be represented. The value was truncated.	Verify that the truncation does not affect the logic of the program.

TABLE B-3. RUN-TIME ERRORS (Contd)

Error Number	Type	Message	Significance	Action
75	W	ARGUMENT .GT. ONE	The value of an argument is greater than one.	Verify that the value 1 can be used for the argument without affecting the logic of the program.
76	W	EXPONENT OF INTEGER IS NON-ZERO		
77	C	ATTEMPTED READ FROM STANDARD OUTPUT OR PUNCH	Data cannot be read from an output device.	Check the input statement for an improper unit number specification. Correct error. Rerun.
78	C	ATTEMPTED WRITE TO STANDARD INPUT UNIT	Data cannot be written to an input device.	Check the output statement for an improper unit number specification. Correct error. Rerun.
79	F	RECORD LENGTH EXCEEDED ON FORMATTED READ	A formatted READ statement attempts to read a record that is longer than the maximum record length specified for the file from which it is reading.	Shorten the record read by the input statement, or increase the record length specification in the PROGRAM statement. Rerun.
80	F	RECORD LENGTH EXCEEDED ON FORMATTED WRITE	A formatted WRITE statement attempts to write a record that is longer than the maximum record length specified for the file to which it is writing.	Shorten the record written by the output statement, or increase the record length specification in the PROGRAM statement. Rerun.
81	C	NULL ELEMENTS IS NOT IMPLEMENTED	This feature cannot be used.	Correct error. Rerun.
82	C	SLASH ON INPUT IS NOT IMPLEMENTED	A slash cannot be used in an input statement to cause the next record to be input.	Correct error. Rerun.
83	C	ILLEGAL FILE NAME -- file name		Correct error. Rerun.
84	C	EXPLICIT FILE PARAMETERS ILLEGAL OR INCOMPATIBLE		Correct error. Rerun.
85		(Currently unassigned)	--	--
86	F	A CALL TO Q8WIDTH IS NOT ALLOWED FOR FILES WITH UNDEFINED RECORD TYPES		Correct error. Rerun.
87	C	BAD MESSAGE FORMAT ON CALL TO FORTRAN EXECUTION	An error exists in the FORTRAN control statement.	Correct error. Rerun.
88	C	ERROR IN OPEN BUFFER OR TAPE FUNCTION CALL		
89	C	ERROR IN CLOSE BUFFER		
90	F	RECORD EXCEEDS RECORD LENGTH ON FILE	The record is too long for this file.	Correct error or use a file with a longer record length. Rerun.

TABLE B-3. RUN-TIME ERRORS (Contd)

Error Number	Type	Message	Significance	Action
91	C	FILE PREVIOUSLY USED FOR NON-BUFFER I/O	A file cannot be used for buffer input/output if it is also used for nonbuffer input/output.	Use either buffer input/output statements with the file, or nonbuffer input/output statements, but not both. Rerun.
92	C	BUFFER DESIGNATION BAD - FIRST WORD ADDRESS .GT. LAST ADDRESS	The first word address of the buffer is greater than the last word address of the buffer.	Change the buffer designation so that the first word address is less than last word address. Rerun.
93	C	WRITE FOLLOWED BY READ	A WRITE statement that writes data to a file is followed by a READ statement that reads data from the same file; no file positioning statements intercede.	Place a REWIND or BACKSPACE statement between the WRITE and READ statements. Check to find out if the proper unit numbers are specified in the WRITE and READ statements, and to find out if the WRITE and READ statements appear in the correct place in the program. Rerun.
94	C	END OF FILE ENCOUNTERED IN BUFFER IN	A BUFFER IN statement attempts to read data from a file, but that file is positioned at the end of the file.	Use a REWIND or BACKSPACE statement to reposition the file before the BUFFER IN statement is executed, or supply missing data on the input file. Rerun.
95	C	FILE NOT READY		
96	C	BUFFER LENGTH GREATER THAN 24 SMALL PAGES	The length of the buffer is too large.	Reduce the length of the buffer. Rerun.
97	F	ILLEGAL RECORD TYPE FOR FORMATTED I/O		Use a file of the correct record type. Rerun.
98	F	ILLEGAL RECORD TYPE FOR UNFORMATTED I/O		Use a file of the correct record type. Rerun.
99	C	DIFFERENT RECORDING MODES ON PROGRAM CARD AND BUFFER I/O		
100		(Currently unassigned)	--	--
101		(Currently unassigned)	--	--
102		(Currently unassigned)	--	--
103		(Currently unassigned)	--	--
104		(Currently unassigned)	--	--
105	F	UNRECOGNIZABLE PARAMETER ENCOUNTERED IN Q7DFSET		

TABLE B-3. RUN-TIME ERRORS (Contd)

Error Number	Type	Message	Significance	Action
106	F	UNRECOGNIZABLE PARAMETER ENCOUNTERED IN Q7DFCL1		
107	F	END OF RECORD ENCOUNTERED DURING BINARY INPUT	A binary input statement attempted to read binary data from a file, but the file is positioned at the end of the file.	Use a REWIND or BACKSPACE statement to reposition the file before the input statement is executed or supply missing data on the input file. Rerun.
108	F	UNDOCUMENTED ERROR DURING BINARY INPUT		
109	F	BIT DATA PRINTED WITH NON B FORMAT--file name	The B format specification must be used for bit data.	Use the B format specification in the FORMAT statement. Rerun.
110	F	B FORMAT USED FOR OTHER THAN BIT DATA--file name	The B format specification is used for data that is of a type other than bit.	Change the B format specification to the appropriate format specification. Rerun.
111	F	DESCRIPTOR PRINTED WITH NON Z FORMAT--file name	The Z format specification must be used for descriptors.	Use the Z format specification in the FORMAT statement. Rerun.
112	F	ILLEGAL RECORD TYPE FOR BUFFER I/O		Use a file of the correct record type. Rerun.
113	C	Q7BUFIN OR Q7BUFOUT WAS CALLED WITH ILLEGAL PARAMETER--file name		
114	C	Q7SEEK WAS CALLED WITH ILLEGAL PARAMETER--file name		
115	C	ARRAY SPECIFIED AS BUFFER IS NOT ON PAGE BOUNDARY (Q7BUFIN/Q7BUFOUT)--file name		
116	C	UNEXPECTED ERROR IN Q7BUFIN OR Q7BUFOUT--file name		
117	C	TOO MANY OUTSTANDING REQUESTS FOR Q7BUFIN/Q7BUFOUT (MUST CALL Q7WAIT)--file name		
118	F	GARBAGE IN FILE OR FILE NOT STRUCTURED--file name		Correct the content or format of the indicated file. Rerun.
119	F	UNRECOGNIZABLE PARAMETER ENCOUNTERED IN Q7DFOFF		
120	C	ROUTINES CALLING Q7DFSET NESTED TOO DEEP		
121	W	DATA FLAG BRANCH - ORX - REGISTER 1 ADDRESS address		

TABLE B-3. RUN-TIME ERRORS (Contd)

Error Number	Type	Message	Significance	Action
122	W	DATA FLAG BRANCH - ORD - REGISTER 1 ADDRESS address		
123	F	DATA FLAG BRANCH - IMAGINARY SQUARE ROOT - REGISTER 1 ADDRESS address		
124	F	DATA FLAG BRANCH - INDEFINITE RESULT - REGISTER 1 ADDRESS address		
125	F	DATA FLAG BRANCH - ZERO DIVISOR - REGISTER 1 ADDRESS address		
126	W	DATA FLAG BRANCH - EXO - REGISTER 1 ADDRESS address		
127	W	DATA FLAG BRANCH - RMZ - REGISTER 1 ADDRESS address		
128	W	DATA FLAG BRANCH - SSC - REGISTER 1 ADDRESS address		
129	W	DATA FLAG BRANCH - DDF - REGISTER 1 ADDRESS address		
130	W	DATA FLAG BRANCH - TBZ - REGISTER 1 ADDRESS address		
131	C	CLASS I DATA FLAG BRANCH - NO INTERRUPT ROUTINE PROVIDED - REGISTER 1 ADDRESS address		
132	C	CLASS III INTERRUPT IN CLASS III INTERRUPT HANDLING ROUTINE - REGISTER 1 ADDRESS address		
133		(Currently unassigned)	--	--
134		(Currently unassigned)	--	--
135	C	DATA FLAG BRANCH, NO PRODUCT BITS ON - REGISTER 1 ADDRESS xxxxxxxx		
136	C	RLP VALUE MISSING OR INVALID IN PROGRAM STATEMENT		
137		(Currently unassigned)	--	--
138	F	Q8WIDTH CALLED WITH WIDTH NEGATIVE OR TOO LARGE		
139	C	SIO ERROR This is preceded by the text of the SIO error message.		
140	F	FORTRAN SECOND USE OF Q7DFCL1 CONFLICTS WITH USER		
141	F	USER USE OF Q7DFCL1 CONFLICTS WITH FORTRAN SECOND		

This indicates the location in the user program where an error occurred. Since the error is actually detected in a run-time routine, the statement identified should be one that generated a call to FORTRAN run-time; that is, an I/O statement or a reference to a FORTRAN supplied function such as SIN or COS.

For data flag branch errors, the form of the error message is:

```
ERROR xxx: EXECUTION INTERRUPTED IN subr
           AT LINE nn - REGISTER 1
           ADDRESS yyyyyy
```

If the register 1 address is in a user routine, the subroutine and line number should correspond to the register 1 address. However, if the register 1 address is in a run-time routine, the subroutine and line number will identify the location in the user's program that generated the call to FORTRAN run-time.

VECTORIZER MESSAGES

The messages listed in table B-4 are issued by the vectorizer phase of the compiler when the V option is specified on the FORTRAN control statement. The messages are informative only, and are not associated with any return code.

The message issued indicates the first impediment to vectorization detected by the compiler. The format of a vectorizer message is:

```
LINE xxxxx LINE yyyyy msg
```

The xxxxx represents the source line number at which the DO loop begins, and the yyyyy represents the source line number at which the impediment was detected. The msg represents the message.

TABLE B-4. VECTORIZER MESSAGES

Message	Significance	Action
BRANCH INTO LOOP	A DO loop must be entered from the top.	No action necessary.
BRANCH OUT OF LOOP	A DO loop must be exited at the bottom of the loop.	No action necessary.
CONTROL VARIABLE APPEARS OTHER THAN AS A SUBSCRIPT REFERENCE	A control variable can appear within the loop only as part of a subscript expression.	No action necessary.
ITERATION COUNT GREATER THAN 65K AT THIS LOOP LEVEL	The length of a vector cannot exceed 65535 elements.	No action necessary.
LHS ARRAY HAS A NON-UNIFORM INCREMENT VALUE	Although the stride of the innermost loop does not have to be 1, the subscript expression must reference memory in a positive, linear order.	No action necessary.
LHS HAS POSSIBLY RECURSIVE PROPERTIES	The compiler cannot determine that a feedback condition does not exist.	No action necessary.
LHS DUMMY ARRAY WITH VARIABLE TEST VALUE - U OPTIMIZATION NOT SPECIFIED	The loop might be vectorizable if the U compile option is specified on the FORTRAN control statement.	No action necessary.
LHS SUBSCRIPT CONTAINS AN EXTERNAL REFERENCE	A subscript expression must be one of the following forms: c, c+n, c-n, or c*n; where c is a control variable and n is an integer constant.	No action necessary.
LHS SUBSCRIPT CONTAINS INTRINSIC FUNCTION REFERENCE	A subscript expression must be one of the following forms: c, c+n, c-n, or c*n; where c is a control variable and n is an integer constant.	No action necessary.
LHS SUBSCRIPT IS DEFINED WITHIN LOOP	A subscript expression must be one of the following forms: c, c+n, c-n, or c*n; where c is a control variable and n is an integer constant.	No action necessary.
LHS VARIABLE APPEARS IN EQUIVALENCE STATEMENT	The loop is rejected because of the possibility of feedback.	No action necessary.
LHS VARIABLE MUST BE REAL, INTEGER, OR LOGICAL	Double-precision, complex, character, and bit data elements will not be vectorized.	No action necessary.

TABLE B-4. VECTORIZER MESSAGES (Contd)

Message	Significance	Action
LOOP IS AN OUTER LOOP WITH A NON-UNIT INCREMENT VALUE	Only innermost loops can have an increment value other than 1.	No action necessary.
LOOP IS AN OUTER LOOP WITH A VARIABLE INCREMENT VALUE	Only innermost loops can have a variable increment value.	No action necessary.
LOOP WITH VARIABLE/TERMINAL VALUE NESTED WITHIN LOOP	Uniform memory reference could not be guaranteed if the outer loop were vectorized.	No action necessary.
NONVECTORIZABLE LOOP NESTED WITHIN LOOP	Any nonvectorizable inner loop prohibits vectorization of all outer loops.	No action necessary.
PROPERTY OF EMBEDDED LOOP PROHIBITS VECTORIZATION OF LOOP		No action necessary.
RHS ARRAY HAS A NON-UNIFORM INCREMENT VALUE	Although the stride of the innermost loop does not have to be 1, the subscript expression must reference memory in a positive, linear order.	No action necessary.
RHS ARRAY HAS POSSIBLY RECURSIVE PROPERTIES	The compiler cannot determine that a feedback condition does not exist.	No action necessary.
RHS ARRAY MUST BE REAL, INTEGER, OR LOGICAL	Double-precision, complex, character, and bit data elements will not be vectorized.	No action necessary.
RHS DUMMY ARRAY WITH VARIABLE TEST VALUE - U OPTIMIZATION NOT SPECIFIED	Loop might be vectorizable if the U compile option is specified on the FORTRAN control statement.	No action necessary.
RHS SUBSCRIPT CONTAINS AN EXTERNAL REFERENCE	A subscript expression must be one of the following forms: c, c+n, c-n, c*n; where c is a control variable and n is an integer constant.	No action necessary.
RHS SUBSCRIPT CONTAINS INTRINSIC FUNCTION REFERENCE	A subscript expression must be one of the following forms: c, c+n, c-n, or c*n; where c is a control variable and n is an integer constant.	No action necessary.
RHS SUBSCRIPT IS DEFINED WITHIN LOOP	A subscript expression must be one of the following forms: c, c+n, c-n, or c*n; where c is a control variable and n is an integer constant.	No action necessary.
SCALAR DEFINED WITHIN LOOP AND APPEARS IN EMBEDDED LOOP	Scalar has recursive properties, which prohibits vectorization.	No action necessary.
SCALAR REFERENCED BEFORE FIRST DEFINITION WITHIN LOOP	Scalar has recursive properties, which prohibits vectorization.	No action necessary.
STATEMENT CONTAINS NON-VECTORIZABLE FUNCTION	The vectorizable functions are: ABS, ACOS, ALOG, ALOG10, ASIN, ATAN, COS, EXP, FLOAT, IABS, IFIX, SIN, SQRT, and TAN.	No action necessary.
STATEMENT CONTAINS NON-VECTORIZABLE OPERATOR	Only the arithmetic operators +, -, *, /, and **, and the logical operators are vectorizable.	No action necessary.

TABLE B-4. VECTORIZER MESSAGES (Contd)

Message	Significance	Action
STATEMENT IS A VECTOR STATEMENT	The loop cannot contain vector or sparse vector assignment statements	No action necessary.
STATEMENT IS NOT AN ASSIGNMENT STATEMENT	The loop can contain only scalar assignment statements.	No action necessary.
TWO OR MORE POTENTIALLY RECURSIVE REFERENCES TO LOOP INDEPENDENT ARRAY	Because a loop-independent array reference is considered to be a scalar reference, every reference to that array in the loop can have the same subscript.	No action necessary.

GLOSSARY C

Terms used in the main text of this manual are described in this section. The definitions give the general meanings of the terms. Precise definitions are given in the main text. Also, most general terms regarding computers and terms defined in the American National Standards documents regarding the FORTRAN language have been excluded.

Array -
An ordered set of variables identified by a single symbolic name. Referencing a single element of an array requires the array name plus a subscript that specifies the element's position in the array.

Array Declarator -
Specifies the dimensions of an array. It consists of an array name followed by a parenthesized list of integer constants or simple integer variables that specify the largest value of each dimension.

ASCII Data -
Characters, each of which has a standard internal representation. One byte (8 bits) is required for each character.

ASCII File -
A type of file that can be manipulated with formatted READ statements, formatted WRITE statements, PRINT statements, and PUNCH statements.

Binary File -
A type of file that can be manipulated by unformatted input/output routines.

Bit Data -
A binary value represented in a FORTRAN program as a binary number in the format B'bb...b' where each b is a 0 or a 1. Each 0 or 1 becomes a 0 bit or a 1 bit in the internal representation for the binary value.

Buffer Input/Output -
Input and output statements that cause data to be transferred between binary files and a buffer area in main memory.

Character Data -
An ASCII value represented in a FORTRAN program by a character string in the format 'cc...c' where each c is in ASCII. Each character becomes a byte of ASCII data in the internal representation for the ASCII value.

Colon Notation -
The notation used to express implied DO subscript expressions in a subarray. The colons separate the initial, terminal, and incrementation values for the implied DO.

Columnwise -
The ordering of the elements in an array declared in a DIMENSION, COMMON, or explicit type statement (the other ordering is rowwise). The succession of subscripts corresponding to the elements of a columnwise array is with the value of the leftmost subscript expression varying the fastest.

Compile Time -
The period of time during which the FORTRAN compiler is reading with the user's program and producing the relocatable module for the program. Compilation is initiated by the FORTRAN system control statement.

Conformability -
Determines whether two subarrays can occur in the same expression. Two subarrays are conformable if they contain the same number of implied DO subscripts and if corresponding implied DO subscript expressions are identical.

Control Vector -
A bit vector that controls operations regarding an associated vector. The control vector elements are set to a configuration of 0s and 1s. The control vector elements are set to a configuration of 0s and 1s. Some of the FORTRAN-supplied functions use control vectors.

Controllee File -
A file that consists of object code generated by the loader. The loader builds a controllee file from relocatable object code produced by a compiler, plus relocatable object code of any externally-defined routines.

Data Element -
A constant, variable, array, or array element.

Data Flag Branch Manager (DFBM) -
A FORTRAN run-time and CYBER 200 library routine that processes data flag branches when they occur in an executing program. A data flag branch is a hardware function of the CYBER 200 computers.

Data Flag Branch (DFB) Register -
Part of the data flag branch hardware. It is a 64-bit register located in the CYBER 200 central processor.

Declaration -
A specification statement that declares attributes of variables, arrays, or function names.

Defining -
Process whereby a variable or array element acquires a predictable or meaningful value. Definition can take place through data initialization, parameter association, DO statement execution, input statement execution, or assignment statement execution. Defining contrasts with naming and referencing.

Descriptor -
A pointer to a vector. In several FORTRAN forms, the descriptor can be used instead of the vector.

Dominance -
A conventional data type hierarchy determining the data type of the result of expression evaluation. Dominated operands are converted during evaluation to the dominant type. The type complex dominates all other types, with types double-precision, real, and integer following in order of decreasing dominance.

Drop File -
A file that is created and maintained for each executing program. Contains any modified pages of the program file, any free space attached, and any read-only data space defined to have temporary write access.

Dynamic Space -
Virtual memory space available for allocation and deallocation at execution time. In particular, space for vectors can be assigned in the dynamic space area by using the descriptor ASSIGN statement.

Explicit Typing -
Specification of the data type of a variable or array by means of one of the explicit type statements (the INTEGER, REAL, COMPLEX, DOUBLE PRECISION, BIT, CHARACTER, and LOGICAL statements). Explicit typing overrides any implicit typing.

External Function -
A function that is defined outside of the program unit that references it. A reference to an external function generates code in the user's object program that causes control to transfer to the external function during program execution. External functions contrast with in-line functions.

File -
A collection of information that can be defined by output statements, or referenced by input statements. Depending on the type of output used to create it, a file can be either implicit or explicit.

First-Letter Rule -
Default type association for data names according to the first letter of the name. Type assignment made is type integer to any name beginning with the letter I, J, K, L, M, or N, and type real to all others. The IMPLICIT statement is used to alter these defaults.

Floating-Point -
Refers to the internal representation for real, double-precision, and complex data.

Generic Function -
A function whose result mode depends on the mode of the argument.

Hexadecimal Data -
A value represented in a FORTRAN program as a hexadecimal number in the format X'hh...h' where each h is a hexadecimal digit (one of the digits 0 through 9 or one of the letters A through F). Each digit becomes the 4-bit binary equivalent in the internal representation for the value.

Implicit Typing -
Specification of the data type of a variable or array by means of the first-letter rule for data names.

Index Vector -
An integer vector whose elements are indexes into another vector. An index is an ordinal number indicating element position in a vector. Some of the FORTRAN-supplied functions use index vectors.

In-Line Function -
A type of predefined function. Referencing an in-line function causes the function's object code to be inserted directly into the relocatable object code of the user's program during compilation. In-line functions contrast with external functions.

Input -
The name of the file read with FORTRAN READ statements that do not specify a unit number. To be used, INPUT must be declared in the PROGRAM statement or in the execution line.

Large Page -
A block of 65536 words in memory starting on a large page boundary. A loader call parameter can be used to tell the operating system that the specified modules are to be placed within a large page loaded on a large page boundary.

Loader -
A utility that links relocatable object modules, together with modules from user libraries or the system library as needed to satisfy external references. It then converts external references and relocatable addresses into the virtual address constants. Thus, relocatable modules are transformed into a virtual code controllee file with the (default) name of GO.

Logical Unit Number -
Integer between 1 and 99 associated with a file by means of the PROGRAM statement declarations or execution line declarations, and used to refer to the file when performing FORTRAN input/output.

Loop-Dependent -
Describes a variable whose value changes as the value of the control variable of a DO loop passes through the range specified in the DO statement. A loop-dependent variable is defined within the range of the loop, while a loop-independent variable is defined (or could be defined with the same effect) outside the range of the loop.

Loop-Independent -
Describes a variable whose value remains constant within the range of a DO loop.

Naming -
Identifying data (or a procedure) without necessarily implying that its current value is to be made available (or, for procedures, that the procedure actions are to be made available) during the execution of the statement in which it is identified. Naming contrasts with referencing and defining.

Object Module -
The relocatable representation of a program unit created by compilation of the program unit. Consists of object code.

Output -
The name of the file to which all run-time error messages and records output with PRINT statements are written. WRITE statements can also be used to write on OUTPUT if OUTPUT is given a logical unit number in the PROGRAM statement.

Precedence -
A conventional arithmetic, relational, and logical operator hierarchy determining the order in which operations are performed during expression evaluation. Operator precedence in FORTRAN corresponds to the mathematical notion of the precedence of mathematical operations.

Predefined Function -
FORTRAN-supplied code that performs common manipulations. Predefined functions can be in-line functions, external functions, or both in-line and external functions.

Program -
A procedure described in the FORTRAN programming language, consisting of at least a main program along with any user-written functions and subroutines that are referenced directly or indirectly by that main program.

Punch -
The name of the file to which records written by the PUNCH statement are written.

Record -
The amount of information read or written by a single FORTRAN READ or WRITE statement. In formatted input/output, a new record is started each time a slash edit descriptor or a format repetition is processed.

Referencing -
Identifying data for the purpose of making its current value available during the execution of the statement containing the reference. Also, identifying a procedure for the purpose of making the actions specified available for execution. Referencing contrasts with naming and defining.

Rowwise -
The ordering of the elements in an array declared in a ROWWISE statement (the other ordering is columnwise). The succession of subscripts corresponding to the elements of a rowwise array is with the value of the rightmost subscript expression varying the fastest.

Run Time -
The period of time during which the compiled program is executing. Execution is initiated by a system control statement.

Scalar -
A single value; contrasted to vectors, which are typically groups of values.

Semicolon Notation -
A notation used to express a vector. The semicolon separates the two items specifying the vector, namely, its first element and its length.

Side Effect -
The alteration of an argument or an element in a common area as a result of a function reference.

Small Page -
A block of 512 words in memory starting on a small page boundary. A small page is the smallest unit that can be moved in or out of main memory by the operating system.

Special Call -
A CYBER 200 FORTRAN language feature that can be used to cause specific machine instructions to be generated in the object code at compile time.

STACKLIB Routine -
A routine that optimizes certain loops that cannot be vectorized.

Subarray -
A cross section of an array. Identified either by the array name or by the array name qualified by a subscript containing (among other kinds of subscript expressions) one or more subscript expressions in colon notation.

Subscripted Array Name -
An array name followed by a parenthesized list of integer constants or simple integer expressions that specify a particular element in an array. A subscripted array name is either an array element reference or an array element definition.

Symbolic Constant -
A name that has a constant value. The value is specified by the PARAMETER statement.

Unit -
A disk or tape on which a file can be created and kept by the operating system.

Vector -
A data representation that typically consists of more than one value; contrasted to scalar data, which represent single values. A subset of an array of scalar elements or of the dynamic space area, delimited by a length and a subscript which designates the position in the array of the vector's first element.

Vectorize -
Cause machine vector instructions to be generated as part of the object code for a CYBER 200 FORTRAN program, either by using vector data and referencing vector functions, or by including vectorizable DO loops in a program compiled when the V compile option has been selected.

Virtual Memory -
A conceptual extension of main memory achieved by a hardware/software technique which permits memory references beyond the physical limitation of main memory. Virtual memory addresses are associated with real addresses in physical memory during program execution.

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 13. Each special call name is a mnemonic preceded by Q8. The mnemonics are identical to the CYBER 200 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.

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 fullword or halfword register, the length and type of which is determined by G field 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, or must be set to zero, depending on G bit 2.
C	1, 2, and 3	Specifies a register that contains the field length and base address for storing the result vector or string field.
	C	Specifies a fullword or halfword register that contains the sum of (A) + (X) for indexed branch instructions, but must be set to zero for compare floating-point instructions.
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.
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. For some format 3 instructions, used as an immediate byte I8.
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.

TABLE D-1. OPERAND DESIGNATORS (Contd)

Designator	Format Type	Definition
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 indexed branch or compare floating-point instructions (B0 - B5), specifies a fullword or halfword register that contains an operand, the length and type of which is determined by G field bits.
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 indexed branch or compare floating-point instructions (B0 - B5), Y specifies one of the following: a register that contains an index used to form the branch address; part of the halfword item count in a relative branch; or a destination register for storing a one if the condition is met, and zero otherwise.
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 indexed branch or compare floating-point instructions (B0 - B5), contains a two's complement or unsigned integer that determines whether the condition is met.

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.

RJ Is an abbreviation for right-justified.

SE Is an abbreviation for sign extended.

YB Indicates a combined Y and B field.

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	xxxx 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 (18)→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	xxxx 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	xxxx 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}	
CALL Q8ADDXV(G,X,A,Y,B,Z,C)	83	1	Add index, vector: A_{16-63}+B_{16-63}→ C_{16-63},A_{0-15}→C_{0-15}	oxxx xooo
CALL Q8ADJE(R_f,S_f,T_f)	75	4	Adjust exponent, fullword FP: (R_f) per S→T_f	

TABLE D-2. SPECIAL CALL FORMATS (Contd)

Special Call	Op Code (Hex)	Instruction Format	Description	G Bits
CALL Q8ADJEH(R_h,S_h,T_h)	55	4	Adjust exponent, halfword FP: (R_h) per S→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→T_f	
CALL Q8ADJSH(R_h,S_h,T_h)	54	4	Adjust significance, halfword FP: (R_h) per S→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: $A\cdot\overline{B}\rightarrow C$	
CALL Q8ANDNV(G,X,A,Y,B,Z,C)†	9D	1	Logical AND NOT: $A\cdot\overline{B}\rightarrow C$, vector	xxxx x11o
CALL Q8ANDV(G,X,A,Y,B,Z,C)†	9D	1	Logical AND: A·B→C, vector	xxxx xool
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	
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	

TABLE D-2. SPECIAL CALL FORMATS (Contd)

Special Call	Op Code (Hex)	Instruction Format	Description	G Bits
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 Q8CFPEQ(G,X,A,YB)††	B0	C	Compare FP and branch if (A).OP. (X) then branch to (Y) + (B) or relative from current location	xlox xxxx
CALL Q8CFPGE(G,X,A,YB)††	B2	C		xlox xxxx
CALL Q8CFPGT(G,X,A,YB)††	B5	C		xlox xxxx
CALL Q8CFPLE(G,X,A,YB)††	B4	C		xlox xxxx
CALL Q8CFPLT(G,X,A,YB)††	B3	C		xlox xxxx
CALL Q8CFPNE(G,X,A,YB)††	B1	C		xlox xxxx
CALL Q8CFPEQ(G,X,A,Y)††	B0	C	Compare FP and set condition if (A).OP.(X) then 1→Y else 0→Y	xllx xxxx
CALL Q8CFPGE(G,X,A,Y)††	B2	C		xllx xxxx
CALL Q8CFPGT(G,X,A,Y)††	B5	C		xllx xxxx
CALL Q8CFPLE(G,X,A,Y)††	B4	C		xllx xxxx
CALL Q8CFPLT(G,X,A,Y)††	B3	C		xllx xxxx
CALL Q8CFPNE(G,X,A,Y)††	B1	C		xllx xxxx
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	
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] → 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] → T_{48-63}	
CALL Q8CON(R_f,,T_h)	76	A	Contract, fullword FP: R_{64} → T_{32}	

TABLE D-2. SPECIAL CALL FORMATS (Contd)

Special Call	Op Code (Hex)	Instruction Format	Description	G Bits
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: $(A/B)_S \rightarrow C$	xxxx 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: $((R_h)/(S_h))_U \rightarrow T_h$	
CALL Q8DIVUS(G,X,A,Y,B,Z,C)	AC	2	Divide upper, sparse vector: $(A/B)_U \rightarrow C$	xxxx 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$\rightarrow$C,C+1	xooo oooo
CALL Q8DOTV(G,X,A,Y,B,Z,C)	DC	1	Dot product vector: A·B$\rightarrow$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$\rightarrow$C	xxoo oooo
CALL Q8ELEN(R_e,I16)	2A	6	Enter length: I16$\rightarrow R_{0-15}$,R_{16-63}unchanged	
CALL Q8EMARK(G,X,A,Y,B,Z,C)†††	EB	3	Edit and mark: A per pattern B$\rightarrow$C, G=1st significant result address	
CALL Q8ES(R_fI16)	3E	6	Enter short, fullword: I16$\rightarrow R_{16-63}$,RJ,SE,0$\rightarrow R_{0-15}$	
CALL Q8ESH(R_h,I16)	4D	6	Enter short, halfword: I16$\rightarrow R_{8-31}$,RJ,SE,0$\rightarrow R_{0-7}$	

TABLE D-2. SPECIAL CALL FORMATS (Contd)

Special Call	Op Code (Hex)	Instruction Format	Description	G Bits
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 Q8EXPHR$_{eh}$,,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
CALL Q8EXTXH(R_h,,T_f)	5D	A	Extend index, halfword FP: R_{8-31}→T_{16-63}SE,R_{0-7}→T_{0-15},SE	
CALL Q8FAULT(G)	06	7	Simulate fault	oooo 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)→T_f	
CALL Q8FLRH(R_h,,T_h)	51	A	Floor, halfword FP: nearest integer .LE.(R_h)→T_h	
CALL Q8FLRV(G,X,A,,,Z,C)	91	1	Floor, vector: nearest integer .LE.A→C	xxxx oooo
CALL Q8IBNZ(R_f,S_i,T_a)	31	7	(R_f)+1→(R_f), if (R_f) ≠ 0 branch to [T_a,S_i]	
CALL Q8IBXEQ(G,X,A,YB,Z,C)	B0	C	Increment and branch index:	xoox xxxx
CALL Q8IBXGE(G,X,A,YB,Z,C)	B2	C	(A)+(X)→C,A_{len}→C_{len};	xoox xxxx
CALL Q8IBXGT(G,X,A,YB,Z,C)	B5	C	if (A)+(X).OP.(Z) then branch to	xoox xxxx
CALL Q8IBXLE(G,X,A,YB,Z,C)	B4	C	(Y)+(B) or YB halfwords from	xoox xxxx
CALL Q8IBXLT(G,X,A,YB,Z,C)	B3	C	current location	xoox xxxx
CALL Q8IBXNE(G,X,A,YB,Z,C)	B1	C		xoox xxxx
CALL Q8IBXEQ(G,X,A,Y,,Z,C)	B0	C	Increment index and set condition:	xolx xxxx
CALL Q8IBXGE(G,X,A,Y,,Z,C)	B2	C	(A)+(X)→C,A_{len}→C_{len};	xolx xxxx
CALL Q8IBXGT(G,X,A,Y,,Z,C)	B5	C	if (A)+(X).OP.(Z) then 1→Y else	xolx xxxx
CALL Q8IBXLE(G,X,A,Y,,Z,C)	B4	C	0→Y	xolx xxxx
CALL Q8IBXLT(G,X,A,Y,,Z,C)	B3	C		xolx xxxx
CALL Q8IBXNE(G,X,A,Y,,Z,C)	B1	C		xolx xxxx
CALL Q8IDLE	00	4	Idle: enable external interrupts and idle	

TABLE D-2. SPECIAL CALL FORMATS (Contd)

Special Call	Op Code (Hex)	Instruction Format	Description	G Bits
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)→C	xxxo oooo
CALL Q8IOR(,X,A,Y,B,Z,C)	F2	3	Logical inclusive OR: A+B→C	
CALL Q8IS(R_f,I16)	3F	6	Increase short, fullword: R_{16-63}+I16→R_{16-63},R_{0-15} unchanged	
CALL Q8ISH(R_h,I16)	4E	6	Increase short, halfword: R_{8-31}+I16→R_{8-31},R_{0-7} unchanged	
CALL Q8IX(R_f,I48)	BF	5	Increase index, fullword: I48+R→R	
CALL Q8IXH(R_h,I24)	CE	5	Increase index, halfword: I24+R→R	
CALL Q8LINKV(G)†	56	7	Link next two vector instructions	ooox xooo
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_{6-63} 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 Q8MCMPW(G,X,A,,B,,C)†	CC	3	Find A=B per maskword C, A index incremented by number of words	oooo ooox

TABLE D-2. SPECIAL CALL FORMATS (Contd)

Special Call	Op Code (Hex)	Instruction Format	Description	G Bits
CALL Q8MIN(G,X,A,,B,Z,C)	D9	1	Vector minimum: $A_{min} \rightarrow C$, item count $\rightarrow B$	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 $\rightarrow C$	xxxx xxxx
CALL Q8MOVL(G,X,A,,B,Z,C)	F8	3	Move bytes left: A $\rightarrow$ C (left to right)	xxxx oxox
CALL Q8MOVLC(G,X,A,,B,Z,C)†††	F9	3	Move bytes left, ones complement: A $\rightarrow$ C (left to right)	xxxx oxox
CALL Q8MOVR(R_i,S_i,T_d)†††	18	7	Move bytes right: $(T_d)+(R_i) \rightarrow (T_d)+(R_i)+(S_i)$	
CALL Q8MOVS(,X,A,,B,Z,C)†††	FA	3	Move and scale: A $\rightarrow$ C, scale (B) decimal places	
CALL Q8MPYB(,X,A,Y,B,Z,C)†††	E2	3	Multiply binary: A*B $\rightarrow$ C	
CALL Q8MPYD(,X,A,Y,B,Z,C)†††	E6	3	Multiply decimal: A*B $\rightarrow$ C	
CALL Q8MPYL(R_f,S_f,T_f)	69	4	Multiply lower, fullword FP: $((R_f)*(S_f))_L \rightarrow T_f$	
CALL Q8MPYLH(R_h,S_h,T_h)	49	4	Multiply lower, halfword FP: $((R_h)*(S_h))_L \rightarrow T_h$	
CALL Q8MPYLS(G,X,A,Y,B,Z,C)	A9	2	Multiply lower, sparse vector: $(A*B)_L \rightarrow C$	xxxx xxxx
CALL Q8MPYLV(G.X.A.Y,B,Z,C)	89	1	Multiply lower, vector: $(A*B)_L \rightarrow C$	xxxx xxxx
CALL Q8MPYS(R_f,S_f,T_f)	6B	4	Multiply significant, fullword FP: $((R_f)*(S_f))_S \rightarrow T_f$	
CALL Q8MPYSH(R_h,S_h,T_h)	4B	4	Multiply significant, halfword FP: $((R_h)*(S_h))_S \rightarrow T_h$	
CALL Q8MPYSS(G,X,A,Y,B,Z,C)	AB	2	Multiply significant, sparse vector: $(A*B)_S \rightarrow C$	xxxx xxxx
CALL Q8MPYSV(G.X,A,Y,B,Z,C)	8B	1	Multiply significant, vector: $(A*B)_S \rightarrow C$	xxxx xxxx
CALL Q8MPYU(R_f,S_f,T_f)	68	4	Multiply upper, fullword FP: $((R_f)*(S_f))_U \rightarrow T_f$	
CALL Q8MPYUH(R_h,S_h,T_h)	48	4	Multiply upper, halfword FP: $((R_h)*(S_h))_U \rightarrow T_h$	
CALL Q8MPYUS(G,X,A,Y,B,Z,C)	A8	2	Multiply upper, sparse vector: $(A*B)_U \rightarrow C$	xxxx xxxx
CALL Q8MPYUV(G,X,A,Y,B,Z,C)	88	1	Multiply upper, vector: $(A*B)_U \rightarrow C$	xxxx xxxx
CALL Q8MPYX(R_f,S_f,T_f)	3D	4	Multiply index, fullword: $R_{16-63}*S_{16-63} \rightarrow T_{16-63}, 0 \rightarrow T_{0-15}$	
CALL Q8MPYXH(R_h,S_h,T_h)	3C	4	Multiply index, halfword: $R_{8-31}*S_{8-31} \rightarrow T_{8-31}, 0 \rightarrow T_{0-7}$	

TABLE D-2. SPECIAL CALL FORMATS (Contd)

Special Call	Op Code (Hex)	Instruction Format	Description	G Bits
CALL Q8MRGB(R_d,S_d,T_d)	15	7	Merge bit strings: interleave (R_d) string with (S_d) string→ T_d string	
CALL Q8MRGC(R_d,S_d,T_d)	17	7	Merge byte strings: (R_d):(S_d), lesser→T_d	
CALL Q8MRGV(G,,A,,B,Z,C)	BD	2	Merge vector: if Z_n=1, A_n→C_n; if Z_n=0, B_n→C_n; result length→C_{0-15}	xoox xoox
CALL Q8MTIME(R_f)	0A	4	Transmit (R_f)→monitor interval timer	
CALL Q8NAND(,X,A,Y,B,Z,C)	F3	3	Logical NAND: $\overline{A \cdot B}$ →C	
CALL Q8NANDV(G,X,A,Y,B,Z,C)†	9D	1	Logical NAND: $\overline{A \cdot B}$ →C, vector	xxxx xoll
CALL Q8NOR(,X,A,Y,B,Z,C)	F4	3	Logical NOR: $\overline{A+B}$ →C	
CALL Q8NORV(G,X,A,Y,B,Z,C)†	9D	1	Logical NOR: $\overline{A+B}$ →C, vector	xxxx xloo
CALL Q8ORN(,X,A,Y,B,Z,C)	F5	3	Logical OR NOT: $A+\overline{B}$→C	
CALL Q8ORNV(G,X,A,Y,B,Z,C)†	9D	1	Logical OR NOT: $A+\overline{B}$→C, vector	xxxx xlol
CALL Q8ORV(G,X,A,Y,B,Z,C)†	9D	1	Logical inclusive OR: A+B→C, vector	xxxx xolo
CALL Q8PACK(R_f,S_f,T_f)	7B	4	Pack, fullword FP: R_{48-63} and S_{16-63}→T_f	
CALL Q8PACKH(R_h,S_h,T_h)	5B	4	Pack, halfword FP: R_{24-31} and S_{8-31}→T_h	
CALL Q8PACKV(G,X,A,Y,B,Z,C)	9B	1	Pack, vector: A_{48-63} and B_{16-63}→C	xxxx xooo
CALL Q8POLYEV(G,X,A,Y,B,Z,C)	DE	1	Polynomial evaluation: A_n per B→C_n	xxxx oooo
CALL Q8PRODCT(G,X,A,,,Z,C)	DB	1	Vector product: Product (A_0,A_1,...A_n)→C	xxoo oooo
CALL Q8RAND(R_f,S_f,T_f)	2D	4	Logical AND: R,S→T	
CALL Q8RCON(R_f,,T_h)	77	A	Rounded contract, fullword FP: R_{64}→T_{32}	
CALL Q8RCONV(G,X,A,,,Z,C)	97	1	Rounded contract, vector: A_{64} rounded→32	oxxx oooo
CALL Q8RIOR(R_f,S_f,T_f)	2E	4	Logical inclusive OR: R,S→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)→T_f	
CALL Q8RTORH(R_h,,T_h)	58	A	Register to register halfword transmit: (R_h)→T_h	
CALL Q8RXOR(R_f,S_f,T_f)	2C	4	Logical exclusive OR: R,S→T	

TABLE D-2. SPECIAL CALL FORMATS (Contd)

Special Call	Op Code (Hex)	Instruction Format	Description	G Bits
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 Q8SHIFTV(G,X,A,Y,B,Z,C)†	8A	1	Shift A by B→C, vector	xxxx xxxx
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: $(SQRT(R_f))_S \rightarrow T_f$	
CALL Q8SQRTH(R_h,,T_h)	53	A	Significant square root, halfword FP: $(SQRT(R_h))_S \rightarrow T_h$	
CALL Q8SQRTV(G,X,A,,,Z,C)	93	1	Significant square root, vector: $SQRT(A)_S \rightarrow C$	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]	
CALL Q8STOAR	0C	4	Store associative registers: AR→$400xx_8$ and higher addresses	
CALL Q8STOC(R_a,S_i,T_f)	13	7	Store byte (character): T_{56-63}→address [R_a+S_i]	
CALL Q8STOH(R_a,S_i,T_h)	5F	7	Store, halfword: (T_h)→ address [R_a+S_i]	
CALL Q8SUBB(,X,A,Y,B,Z,C)†††	E1	3	Subtract binary: A-B→C	
CALL Q8SUBD(,X,A,Y,B,Z,C)†††	E5	3	Subtract decimal: A-B→C	

TABLE D-2. SPECIAL CALL FORMATS (Contd)

Special Call	Op Code (Hex)	Instruction Format	Description	G Bits
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: $(A-B)_L \rightarrow C$	xxxx 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: $(A_n-B_n) \bmod (I8) \rightarrow C_n$	
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: $(A-B)_N \rightarrow C$	xxxx 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: $(A-B)_U \rightarrow C$	xxxx 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: $R_{16-63}-S_{16-63} \rightarrow T_{16-63}, R_{0-15} \rightarrow T_{0-15}$	
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: $Sum(A_0,A_1,\ldots A_n) \rightarrow C,C+1$	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.0	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

TABLE D-2. SPECIAL CALL FORMATS (Contd)

Special Call	Op Code (Hex)	Instruction Format	Description	G Bits
CALL Q8TRU(R_f,,T_f)	70	A	Truncate, fullword FP: nearest integer .LE.(R_f) → T_f	
CALL Q8TRUH(R_h,,T_h)	50	A	Truncate, halfword FP: nearest integer .LE.(R_h) → 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} → 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 xxxx
CALL Q8VXTOV(G,,A,,B,,C)	BA	1	Vector to vector indexed transmit: B indexed by A → C	xooo oxxx
CALL Q8WJTIME(R_f)	3A	A	Transmit (R_f) → 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-$\overline{B}$ → C	
CALL Q8XORNV(G,X,A,Y,B,Z,C)†	9D	1	Logical exclusive OR NOT: (equivalence) A-$\overline{B}$ → C, vector	xxxx x111
CALL Q8XORV(G,X,A,Y,B,Z,C)†	9D	1	Logical exclusive OR: A-B → C, vector	xxxx xooo
CALL Q8VSB(,,T_a)	05	4	Void instruction stack and branch to (T_a)	
CALL Q8ZTOD(G,X,A,,,Z,C)	FB	3	Pack zoned to BCD: A → C	xxoo oooo

†Available on the CYBER 205, but not on the STAR 100 or the CYBER 203.

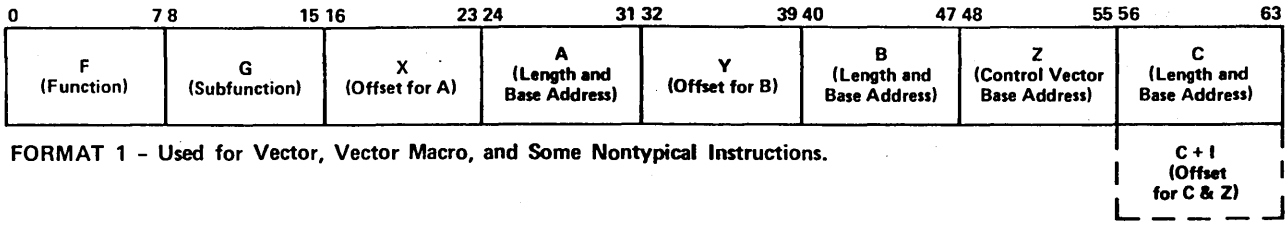
††Available on the CYBER 203 and CYBER 205, but not on the STAR 100.

†††Available on the STAR 100 and CYBER 203, but not on the CYBER 205.

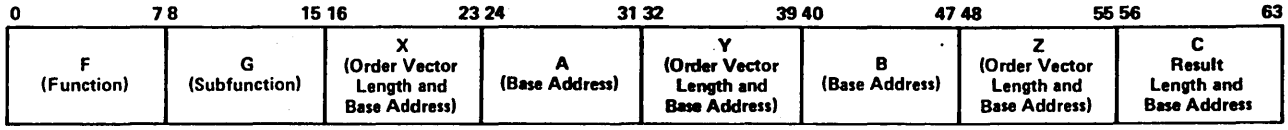
.OP.	Indicates one of the logical operators .EQ., .NE., .GE. or .LT.
U	Indicates upper result.
L	Indicates lower result.
N	Indicates normalized upper result.
S	Indicates significant result.

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 CYBER 200 Computer Hardware Reference Manual.

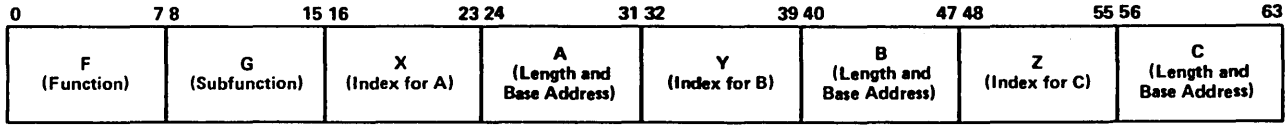
As a convenience for the user of special calls, the special calls are listed by op code in table D-3.



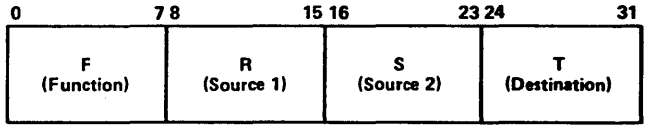
FORMAT 1 - Used for Vector, Vector Macro, and Some Nontypical Instructions.



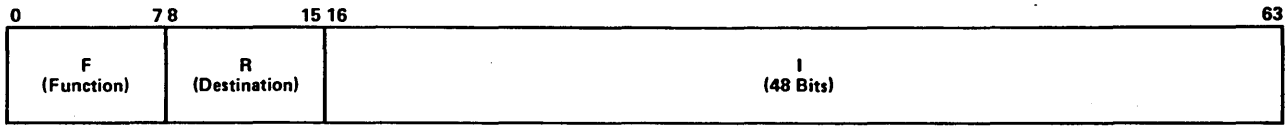
FORMAT 2 - Used for Sparse Vector and Some Nontypical Instructions.



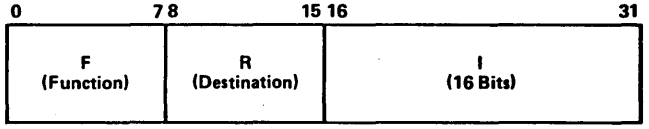
FORMAT 3 - Used for Logical String and String Instructions.



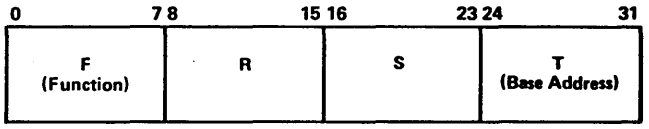
FORMAT 4 - Used for Some Register, for all Monitor instructions, and for the #3D, and #04 Nontypical Instructions.



FORMAT 5 - Used for the #BE, #BF, #CD, and #CE Index Instructions and for the #B6 Branch Instruction.

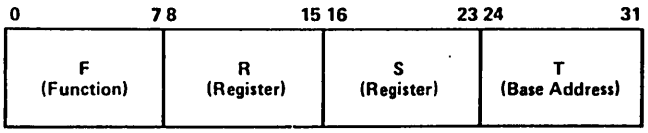


FORMAT 6 - Used for the #3E, #3F, #4D, and #4E Index Instructions and the #2A Register Instruction.

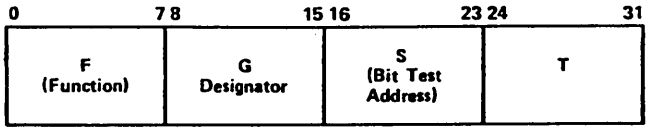


FORMAT 7 - Used for Some Branch and Nontypical Instructions.

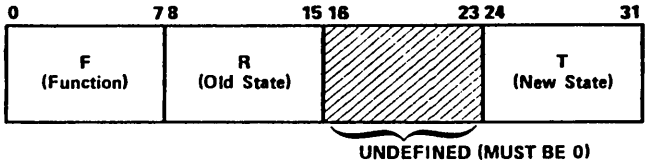
Figure D-1. Instruction Formats (Sheet 1 of 2)



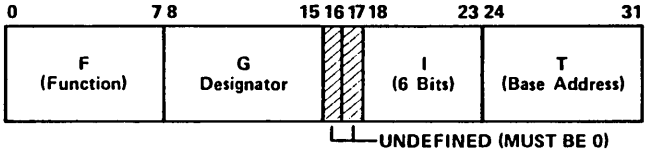
FORMAT 8 - Used for Some Branch Instructions.



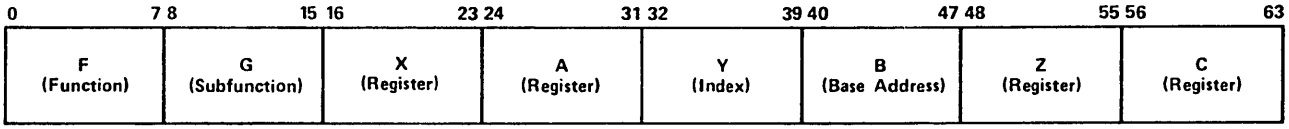
FORMAT 9 - Used for the #32 Branch Instruction



FORMAT A - Used for Some Index, Branch, and Register Instructions.



FORMAT B - Used for the #33 Branch Instruction.



FORMAT C - Used for the #B0 through #B5 Branch Instructions.

Figure D-1. Instruction Formats (Sheet 2 of 2)

TABLE D-3. SPECIAL CALLS LISTED BY OP CODE

Op code (hex)	Special Call	Op code (hex)	Special Call	Op code (hex)	Special Call	Op code (hex)	Special Call	Op code (hex)	Special Call
00	Q8IDLE	37	Q8RJTIME	6E	**Q8EXTB**		Q8ANDNV	CF	Q8ACPS
04	Q8BKPT	38	Q8LTOL	6F	**Q8DIVS**		Q8XORNV	D0	Q8AVG
05	Q8VSB	39	Q8CLOCK	70	**Q8TRU**	A0	Q8ADDUS	D1	Q8ADJM
06	Q8FAULT	3A	Q8WJTIME	71	**Q8FLR**	A1	Q8ADDLS	D4	Q8AVGD
08	Q8SETCF	3B	Q8LSDFR	72	**Q8CLG**	A2	Q8ADDNS	D5	Q8DELTA
09	Q8EXIT	3C	Q8MPYXH	73	**Q8SQRT**	A4	Q8SUBUS	D6	Q8SKEYB
0A	Q8MTIME	3D	Q8MPYX	74	**Q8ADJS**	A5	Q8SUBLS	D7	Q8TLMARK
0C	Q8STOAR	3E	Q8ES	75	**Q8ADJE**	A6	Q8SUBNS	D8	Q8MAX
0D	Q8LODAR	3F	Q8IS	76	**Q8CON**	A8	Q8MPYUS	D9	Q8MIN
0E	Q8TLXI	40	Q8ADDUH	77	**Q8RCON**	A9	Q8MPYLS	DA	Q8SUM
0F	Q8LODKEY	41	Q8ADDLH	78	**Q8RTOR**	AB	Q8MPYSS	DB	Q8PRODCT
10	Q8DTOB	42	Q8ADDNH	79	**Q8ABS**	AC	Q8DIVUS	DC	Q8DOTV
11	Q8BTOD	44	Q8SUBUH	7A	**Q8EXP**	AF	Q8DIVSS	DD	Q8DOTS
12	Q8LODC	45	Q8SUBLH	7B	**Q8PACK**	B0	Q8IBXEQ	DE	Q8POLYEV
13	Q8STOC	46	Q8SUBNH	7C	**Q8LTOR**		Q8CFPEQ	DF	Q8INTVAL
14	Q8CPSB	48	Q8MPYUH	7D	**Q8SWAP**	B1	Q8IBXNE	E0	Q8ADDB
15	Q8MRGB	49	Q8MPYLH	7E	**Q8LOD**		Q8CFPNE	E1	Q8SUBB
16	Q8MASKB	4B	Q8MPYSH	7F	**Q8STO**	B2	Q8IBXGE	E2	Q8MPYB
17	Q8MRGC	4C	Q8DIVUH	80	**Q8ADDUV**		Q8CFPGE	E3	Q8DIVB
18	Q8MOVR	4D	Q8ESH	81	**Q8ADDLV**	B3	Q8IBXLT	E4	Q8ADDD
19	Q8SCNRNE	4E	Q8ISH	82	**Q8ADDNV**		Q8CFPLT	E5	Q8SUBD
1A	Q8FILLC	4F	Q8DIVSH	83	**Q8ADDXV**	B4	Q8IBXLE	E6	Q8MPYD
1B	Q8FILLR	50	Q8TRUH	84	**Q8SUBUV**		Q8CFPLE	E7	Q8DIVD
1C	Q8MASKZ	51	Q8FLRH	85	**Q8SUBLV**	B5	Q8IBXGT	E8	Q8CMPB
1D	Q8MASKO	52	Q8CLGH	86	**Q8SUBNV**		Q8CFPGT	E9	Q8CMPD
1E	Q8CNTEQ	53	Q8SQRTH	87	**Q8SUBXV**	B6	Q8BIM	EA	Q8MMRGC
1F	Q8CNTO	54	Q8ADJSH	88	**Q8MPYUV**	B7	Q8VTOVX	EB	Q8EMARK
20	Q8BHEQ	55	Q8ADJEH	89	**Q8MPYLV**	B8	Q8VREVV	EC	Q8ADDMOD
21	Q8BHNE	56	Q8LINKV	8B	**Q8MPYSV**	B9	Q8TPMOV	ED	Q8SUBMOD
22	Q8BHGE	58	Q8RTORH	8C	**Q8DIVUV**	BA	Q8VXTOV	EE	Q8TL
23	Q8BHLT	59	Q8ABSH	8F	**Q8DIVSV**	BB	Q8MASKV	EF	Q8TLTEST
24	Q8BEQ	5A	Q8EXPH	90	**Q8TRUV**	BC	Q8CPSV	F0	Q8XOR
25	Q8BNE	5B	Q8PACKH	91	**Q8FLRV**	BD	Q8MRGV	F1	Q8AND
26	Q8BGE	5C	Q8EXTH	92	**Q8CLGV**	BE	Q8EX	F2	Q8IOR
27	Q8BLT	5D	Q8EXTXH	93	**Q8SQRTV**	BF	Q8IX	F3	Q8NAND
28	Q8SCNLEQ	5E	Q8LODH	94	**Q8ADJSV**	C0	Q8SELEQ	F4	Q8NOR
29	Q8SCNLNE	5F	Q8STOH	95	**Q8ADJEV**	C1	Q8SELNE	F5	Q8ORN
2A	Q8ELEN	60	Q8ADDU	96	**Q8CONV**	C2	Q8SELGE	F6	Q8ANDN
2B	Q8ADDLEN	61	Q8ADDL	97	**Q8RCONV**	C3	Q8SELLT	F7	Q8XORN
2C	Q8RXOR	62	Q8ADDN	98	**Q8VTOV**	C4	Q8CMPEQ	F8	Q8MOVL
2D	Q8RAND	63	Q8ADDX	99	**Q8ABSV**	C5	Q8CMPNE	F9	Q8MOVLC
2E	Q8RIOR	64	Q8SUBU	9A	**Q8EXPV**	C6	Q8CMPGE	FA	Q8MOVS
2F	Q8BARB	65	Q8SUBL	9B	**Q8PACKV**	C7	Q8CMPLT	FB	Q8ZTOD
30	Q8SHIFTI	66	Q8SUBN	9C	**Q8EXTV**	C8	Q8SRCHEQ	FC	Q8DTOZ
31	Q8IBNZ	67	Q8SUBX	9D	**Q8XORV**	C9	Q8SRCHNE	FD	Q8MCMPC
32	Q8BAB	68	Q8MPYU		**Q8ANDV**	CA	Q8SRCHGE	FE	Q8SKEYC
33	Q8BADF	69	Q8MPYL		**Q8ORV**	CB	Q8SRCHLT	FF	Q8SKEYW
34	Q8SHIFT	6B	Q8MPYS		**Q8NANDV**	CC	Q8MCMPW		
35	Q8DBNZ	6C	Q8DIVU		**Q8NORV**	CD	Q8EXH		
36	Q8BSAVE	6D	Q8INSB		**Q8ORNV**	CE	Q8IXH		

This appendix contains a list of the functions that are available for reference for any CYBER 200 FORTRAN program, as discussed in section 14. For each function, table E-1 indicates what type of code (in-line, external, or both) is generated during compilation as a result of referencing the function.

TABLE E-1. SUPPLIED FUNCTIONS

Function	Category	Fast Call Name	Function	Category	Fast Call Name
ABS	NX	-	DASIN	X	FT_XDASN
ACOS	X	FT_XACOS	DATAN	X	FT_XDATN
AIMAG	NX	-	DATAN2	X	FT_XDTN2
AINT	NX	-	DATE	X	-
ALOG	X	FT_XALOG	DBLE	NX	-
ALOG10	X	FT_XLOGT	DCOS	X	FT_XDCOS
AMAX0	NX	-	DCOSH	X	FT_XDCSH
AMAX1	NX	-	DDIM	NX	FT_XDDIM
AMIN0	NX	-	DEXP	X	FT_XDEXP
AMIN1	NX	-	DFLOAT	NX	-
AMOD	NX	-	DIM	NX	-
ASIN	X	FT_XASIN	DINT	NX	-
ATAN	X	FT_XATAN	DLOG	X	FT_XDLOG
ATAN2	X	FT_XATN2	DLOG10	X	FT_XDLGT
CABS	NX	FT_XCABS	DMAX1	X	-
CCOS	X	FT_XCCOS	DMIN1	X	-
CEXP	X	FT_XCEXP	DMOD	X	FT_XDMOD
CLOG	X	FT_XCLOG	DPROD	NX	FT_XDPRD
CMPLX	NX	-	DSIGN	NX	-
CONJG	NX	-	DSIN	X	FT_XDSIN
COS	X	FT_XCOS	DSINH	X	FT_XDSNH
COSH	X	FT_XCOSH	DSQRT	X	FT_XDSQT
COTAN	X	FT_XCOTN	DTAN	X	FT_XDTAN
CSIN	X	FT_SCSIN	DTANH	X	FT_XDTNH
CSQRT	X	FT_XCSQT	EXP	X	-
DABS	NX	-	FLOAT	NX	-
DACOS	X	FT_XDACS	IABS	NX	-

TABLE E-1. SUPPLIED FUNCTIONS (Contd)

Function	Category	Fast Call Name
IDIM	NX	-
IDINT	NX	-
IFIX	NX	-
INT	NX	-
ISIGN	NX	-
MAXO	NX	-
MAX1	NX	-
MINO	NX	-
MIN1	NX	-
MOD	NX	-
Q8SCNT	N	-
Q8SDFB	N	-
Q8SDOT	N	-
Q8SEQ	N	-
Q8SEXTB	N	-
Q8SGE	N	-
Q8SINSB	N	-
Q8SLEN	N	-
Q8SLT	N	-
Q8SMAX	N	-
Q8SMAXI	N	-
Q8SMIN	N	-
Q8SMINI	N	-
Q8SNE	N	-
Q8SPROD	N	-
Q8SSUM	N	-
Q8VADJM	N	-
Q8VARCMP	N	-
Q8VAVG	N	-
Q8VAVGD	N	-
Q8VCMPRS	N	-
Q8VCTRL	N	-
Q8VDELT	N	-
Q8VEQI	N	-
Q8VGATHP	NX	-
Q8VGATHR	NX	-
Q8VGEI	N	-
Q8VINTL	N	-
Q8VLTI	N	-
Q8VMASK	N	-
Q8VMERG	N	-
Q8VMKO	N	-
Q8VMKZ	N	-
Q8VNEI	N	-
Q8VPOLY	N	-
Q8VREV	N	-
Q8VSCATP	NX	-
Q8VSCATR	NX	-
Q8VXPND	N	-
RANF	NX	-
REAL	NX	-
SECOND	X	-
SIGN	NX	-
SIN	X	FT_XSIN
SINH	X	FT_XSINH
SNGL	NX	-
SQRT	NX	FT_XSQRT
TAN	X	FT_XTAN
TANH	X	FT_XTANH
TIME	X	-
VABS	NX	FT_XVABS
VACOS	X	FT_XVACS
VAIMAG	X	FT_XVAIM
VAINT	NX	FT_XVAIN
VALOG	X	FT_XVLOG
VALOG10	X	FT_XVLGT
VAMOD	X	FT_XVAMD
VASIN	X	FT_XVASN

TABLE E-1. SUPPLIED FUNCTIONS (Contd)

Function	Category	Fast Call Name	Function	Category	Fast Ca'l Name
VATAN	X	FT_XVATN	VFLOAT	NX	FT_XVFLT
VATAN2	X	FT_XVAT2	VIABS	NX	FT_XVIAB
VCABS	X	FT_XVCAB	VIDIM	X	FT_XVDIM
VCCOS	X	FT_XVCCS	VIFIX	NX	FT_XVFIX
VCEXP	X	FT_XVCXP	VINT	NX	FT_XVINT
VCLOG	X	FT_XVCLN	VISIGN	X	FT_XVISN
VCMPLX	X	FT_XVCPX	VMOD	X	FT_XVMOD
VCONJG	X	FT_XVCJG	VREAL	X	FT_XVREL
VCOS	X	FT_XVCOS	VSIGN	X	FT_XVSGN
VCSIN	X	FT_XVCSN	VSIN	X	FT_XVSIN
VCSQRT	X	FT_XVCSR	VSNGL	X	FT_XVSGL
VDBLE	X	FT_XVDBL	VSQRT	NX	FT_XVSQT
VDIM	X	FT_XVDIM	VTAN	X	FT_XVTAN
VEXP	X	FT_XVEXP			

N = In-Line

X = External

NX = In-line and external

CYBER 200 FORTRAN STATEMENT SUMMARY

This appendix contains a summary of the statement forms described in the main text. Given are the entities that compose each statement; refer to the main text for the detailed specifications for these entities. Abbreviations used in this appendix are the following:

v	=	variable or array element
va=	=	variable, array element, or array
s	=	statement label
iv=	=	integer variable
n	=	integer constant, integer symbolic constant, or integer variable
type	=	INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, BIT, or CHARACTER
p	=	variable, array, or array declarator
pf	=	variable, array, function name, or array declarator
k	=	length of a type character pf
K	=	length of all type character pf in statement
a	=	array declarator
arg	=	argument (dummy or actual)
d	=	descriptor or descriptor array element
vr	=	vector (expressed in semicolon notation), descriptor, or descriptor array element
u	=	logical unit number
fmt	=	format designator
iolist	=	input/output list

Brackets around an item indicates that the item is optional.

ASSIGNMENT STATEMENTS

integer	v = arithmetic expression
real	v = arithmetic expression
complex	v = arithmetic expression
double-precision	v = arithmetic expression
character	v = character expression
logical	v = logical expression
bit	v = bit expression

FLOW CONTROL STATEMENTS

GO TO s

GO TO iv[,](s[, . . . ,s])
ASSIGN s TO iv

GO TO (s[, . . . ,s])[,] iv

IF (arithmetic expression) s,s,s

IF (logical expression) statement

IF (logical expression) THEN
ELSE IF (logical expression) THEN
ELSE
END IF

DO s iv = n,n[,n]

CONTINUE

PAUSE [five-digit integer constant]
PAUSE [character constant]

STOP [five-digit integer constant]
STOP [character constant]

SPECIFICATION STATEMENTS

IMPLICIT type (list of letters and ranges of letters)
[, . . . ,type (list of letters and ranges of letters)]

type pf [/initial value/] [, . . . ,pf [/initial value/]]

CHARACTER [*K] pf [*k] [/initial value/] [, . . . ,pf [*k] [/initial value/]]

DIMENSION a[, . . . ,a]

ROWWISE a[, . . . ,a]

COMMON p[, . . . ,p] [. . ./common block name/p[, . . . ,p]]
COMMON /common block name/ p [, . . . ,p] [. . ./common block name/ p[, . . . ,p]]
COMMON / / p[, . . . ,p] [. . ./common block name/ p [, . . . ,p]]

EQUIVALENCE (va,va[, . . . ,va]) [, . . . ,(va,va[, . . . ,va])]

EXTERNAL procedure name [, . . . , procedure name]

DATA variable list/data list/ [, . . . , variable list/data list/]
where a variable list element is a variable, array element, array name, or implied DO; and a data list element is a constant or a repeat count times a constant

PARAMETER ($name_1$=$value_1$[, . . . ,$name_n$=$value_n$])
where $name_i$ is the name of a symbolic constant, and $value_i$ is a constant expression.

PROCEDURE DEFINITION

PROGRAM [procedure name] [(fip[,...,fip])]
where fip is a file information parameter

statement function name (arg[,...,arg]) = expression

[type] FUNCTION procedure name (arg[,...,arg]),
except for type = CHARACTER

CHARACTER FUNCTION procedure name *K
(arg[,...,arg])

SUBROUTINE procedure name [(arg[,...,arg])]

BLOCK DATA [subprogram name]

ENTRY procedure name [(arg[,...,arg])]
for subroutines
ENTRY procedure name (arg[,...,arg])
for functions

RETURN [n]
for subroutines
RETURN
for functions

CALL procedure name [(arg[,...,arg])]

INPUT/OUTPUT STATEMENTS

READ (u[,fmt] [,END=s] [,ERR=s]) [iolist]
READ fmt [, iolist]

WRITE (u) iolist
WRITE (u,fmt)[iolist]

PRINT fmt[, iolist]

PUNCH fmt[, iolist]

BUFFER IN (u,data transfer mode)(first location in buffer, last location in buffer)

BUFFER OUT (u, data transfer mode)(first location in buffer, last location in buffer)

ENCODE (record length,fmt,name of buffer of records) [iolist]

DECODE (record length,fmt,name of buffer of records) [iolist]

NAMELIST /group name/list of variables and arrays [. . ./group name/list of variables and arrays]

Namelist input: &group name Δ pn=constant list, pn=constant list, ... &END

READ (u,group name)
READ group name

WRITE (u,group name)

PRINT group name

PUNCH group name

REWIND u

BACKSPACE u

ENDFILE u

s FORMAT ([/. . .]field spec sep field spec sep ...[/. .])
where sep is a separator (a comma or one or more slashes), and field spec is a field specification for data conversion

UNIT (u)

LENGTH (u)

ARRAY ASSIGNMENT

subarray = array expression

subarray = scalar expression

VECTOR STATEMENTS

ASSIGN d, vr
ASSIGN d, .DYN. integer scalar expression

FREE

vr = vector arithmetic expression or scalar arithmetic expression

bit vr = vector bit expression

DESCRIPTOR p [,..., p]

type FUNCTION procedure name (arg[,..., arg];*)

ENTRY procedure name (arg[,..., arg];*)

DATA variable list/data list/[,..., variable list/data list/]
where variable list can include vr and data list can include vectors (expressed in semicolon notation)

Certain features of CYBER 200 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 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 fullword. An H constant that is too long is truncated on the right side, and a warning diagnostic is issued. An R constant is right-justified with binary zero fill in a fullword. An R constant that is too long is truncated on the right 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.

The BUFFER IN and BUFFER OUT statements are used to transmit binary data between binary 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.

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 by 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.

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

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

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)

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 14. 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

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 $0 \leq n \leq 64$. 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.

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 $-64 \leq n \leq 64$. The result is undefined if n is outside the range. The n value is integer.

XOR (a_1,a_2, . . .)

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

INDEX

COMMENT SHEET

MANUAL TITLE: CYBER 200 FORTRAN Version 3 Reference Manual

PUBLICATION NO.: 60457040 **REVISION:** C

NAME: _____

COMPANY: _____

STREET ADDRESS: _____

CITY: _____ **STATE:** _____ **ZIP CODE:** _____

This form is not intended to be used as an order blank. Control Data Corporation welcomes your evaluation of this manual. Please indicate any errors, suggested additions or deletions, or general comments below (please include page number references).

☐ Please reply ☐ No reply necessary

CUT ALONG LINE

AA3419 REV. 4/79 PRINTED IN U.S.A.

NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

FOLD ON DOTTED LINES AND TAPE

TAPE

TAPE

FOLD

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 8241 MINNEAPOLIS, MINN.

POSTAGE WILL BE PAID BY

CONTROL DATA CORPORATION

Publications and Graphics Division

215 Moffett Park Drive
Sunnyvale, California 94086

CUT ALONG LINE

FOLD

FOLD

)

)

)

)

)

)

)

)

)

CORPORATE HEADQUARTERS P.O. BOX 0 MINNEAPOLIS, MINNESOTA 55440