CDC® CYBER 200 FORTRAN VERSION 3

FOR USE WITH
CDC® CYBER 200
OPERATING SYSTEM
VERSION 1

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)	
D	This revision documents the CDC CYBER 200 FORTRAN language at release 1.5.2.
(2-16-81)	
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	D
iii/iv	D
v/vi	C
vii thru x	D
xi/xii	D
xiii	D
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	B
5-2	B
5-3 thru 5-5	D
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	D
11-3	B
11-4	C
11-5 thru 11-7	B
11-8 thru 11-14	D
12-1	B
12-2	B
13-1 thru 13-17	C
14-1 thru 14-4	B
14-5	D
14-6	B
14-7 thru 14-25	B
15-1	D
15-2 thru 15-12	C
16-1 thru 16-7	B
A-1	C
A-2	C
B-1	D
B-2	B
B-3	D
B-4	B
B-5	B
B-6	D
B-7 thru B-20	B
B-21	D
B-22	D
B-22.1	D
B-22.2	D
B-23 thru B-28	B
B-29	D
B-30	B
C-1	D
C-2	B
C-3	B
D-1 thru D-16	B
E-1	B
E-2	D
E-3	D
F-1 thru F-4	D
G-1 thru G-3	B
Index-1 thru -5	D
Comment Sheet	D
Mailer	-
Back Cover	-

CONTENTS

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.

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, else-block, and elseif-block respectively.

An if-block, else-block, and elseif-block can contain any number of executable statements or no statements. Control can transfer out of an if-block, else-block, or elseif-block. Control cannot transfer into an if-block, else-block, or elseif-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 the if-block. If a statement in the if-block does not transfer control elsewhere, control transfers to the statement following the END IF statement after execution of the if-block.

```
IF (expr) THEN

     if-block

ELSE

     else-block

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 the else-block. If a statement in the else-block does not transfer control elsewhere, control transfers to the statement following the END IF statement after execution of the else-block.

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

```
IF (expr) THEN

     if-block

ELSE IF (expr) THEN

     elseif-block-1

ELSE IF (expr) THEN

     elseif-block-2

END IF
```

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

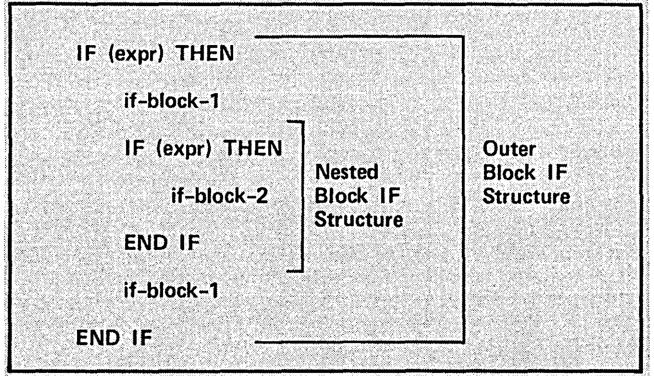
If the expression in the block IF statement is false, control transfers to the first ELSE IF statement that is associated with the block IF statement. The expression in this ELSE IF statement is evaluated. If the expression is true, execution continues with the first executable statement in elseif-block-1. If a statement in elseif-block-1 does not transfer control elsewhere, control transfers to the statement following the END IF statement after execution of elseif-block-1.

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



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

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, else-block, or elseif-block but the entire range of the DO loop must appear in the if-block, else-block, or elseif-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 block WHERE or OTHERWISE 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, else-block, or elseif-block, but the entire range of the DO loop must appear in the if-block, else-block, or elseif-block.

The range of a DO loop can contain a block WHERE structure, but the entire block WHERE structure must appear in the DO loop range. An END WHERE statement can be the terminal statement of a DO loop.

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

s	The symbolic name of a subroutine, or an entry point name in a subroutine.
a_i	Optional. An actual argument which can be an expression, vector, descriptor, 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

VECTOR PROGRAMMING

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

```
A.
             DO 222 KEM=1,300      ◄——①
                  •
                  •
               DO 100 I=1,10       ◄——②
                 DO 200 J=1,N      ◄——③
      200          A(J,I)=X(J,I)
      100      CONTINUE
                  •
                  •
      222    CONTINUE

B.
             DO 300 MM=1,11        ◄——⑤
               DO 200 JJ=1,30      ◄——⑥
                 DO 400 IN=1,200   ◄——⑦
                    •
                    •
      400        CONTINUE
      200      CONTINUE
      300    CONTINUE
```

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 that appear in an EQUIVALENCE statement if the U compile option is selected.	Any data elements appearing on the left side of an assignment statement which have appeared in EQUIVALENCE statements if the U compile option is not selected.
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

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 ith bit is 1 if the ith 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 ith 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 ith bit is 1 if the ith element of the real vector VR(1;89) equals the ith 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 ith bit is 1 if the ith 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 ith bit is 1 if the ith element of the real vector VR(1;25), multiplied by 3, is equal to the ith 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, B1, B2
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 ith element is 1 unless the ith elements of the vector values of BV and BV2 are 0, in which case the ith element is 0.

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

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

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

  Bit vector in which the ith element is 1 if the ith 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
      DESCRIPTOR 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 ith 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 ith element of RES(1;30) with the ith 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 ith 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 ith element of BV(1;60) with the value of the ith 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.

WHERE STATEMENT

The WHERE statement has the following form:

WHERE (e) st

e	A vector bit expression.
st	A vector assignment statement.

All vector operands in the vector bit expression and in the vector expression that appears in the vector assignment statement must have the same length. All vectors and vector expressions that appear in the vector assignment statement must be of type integer or real. The vector expression that appears in the vector assignment statement can contain only addition, subtraction, multiplication, and division operations, and references to the vector functions VFLOAT, VIFIX, VINT, VAINT, VSQRT, VABS, and VIABS.

When the WHERE statement is executed, the vector bit expression is evaluated. The evaluation produces a control vector. A control vector is a bit vector that controls the storing of values into a vector. This control vector is used for the vector assignment statement that appears in the WHERE statement.

The vector expression that appears in the vector assignment statement is evaluated. Each value of the result vector is assigned to the corresponding vector element on the left side of the vector assignment statement only if the corresponding element in the control vector contains a 1 bit.

A value is not assigned to the corresponding vector element on the left side of the vector assignment statement if the corresponding element in the control vector contains a 0 bit. If a value is not assigned to a vector element, data flag branches are disabled for the operations that compute that value.

Given the declarations:

```
REAL A(5), B(5), C(5)
BIT CA(5), CB(5)
DATA A /3.0, 9.0, 12.0, 2.0, 16.0/
DATA B /6.0, 6.0, 10.0, 5.0, 4.0/
DATA C /9.0, 3.0, 0.0, 7.0, 7.0/
DATA CA /B'11101'/, CB /B'11000'/
```

the following are examples of the WHERE statement:

- WHERE (A(1;5).LT.B(1;5)) C(1;5)=B(1;5)-A(1;5)

  Causes the value the value 3.0 to be assigned to the first and fourth elements of vector C(1;5). All other elements of vector C(1;5) are unchanged.

- WHERE (CA(1;5).AND.CB(1;5)) C(1;5)=A(1;5)+B(1;5)

  Causes the values 9.0 and 15.0 to be assigned to the first and second elements of vector C(1;5) respectively. All other elements of vector C(1;5) are unchanged.

- WHERE (C(1;5).NE.0.0) A(1;5)=B(1;5)/C(1;5)

  Causes the values .66667, 2.0, .71429, and .57143 to be assigned to the first, second, fourth, and fifth elements of vector A(1;5) respectively. The division of the third element of vector B(1;5) by the third element of vector C(1;5) is a division by zero; however, because the third element of the control vector contains a 0 bit, no data flag branch occurs.

BLOCK WHERE STATEMENT

The block WHERE statement has the following form:

WHERE (e)

e A vector bit expression.

When the block WHERE statement is executed, the vector bit expression is evaluated. The evaluation produces a control vector. A control vector is a bit vector that controls the storing of values into a vector. This control vector is used for all of the vector assignment statements that appear between the block WHERE statement and the next END WHERE statement.

OTHERWISE STATEMENT

The OTHERWISE statement has the following form:

OTHERWISE

The OTHERWISE statement can be used with a block WHERE statement to reverse the effect of the control vector established in the block WHERE statement. Reversing the effect of the control vector causes a value to be assigned to a vector element only if the corresponding element in the control vector contains a 0 bit, rather than a 1 bit. An OTHERWISE statement affects all vector assignment statements that appear between the OTHERWISE statement and the next END WHERE statement.

END WHERE STATEMENT

The END WHERE statement has the following form:

END WHERE

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

BLOCK WHERE STRUCTURES

Block WHERE structures provide for the execution of any number of vector assignment statements using a single control vector. A control vector is a bit vector that controls the storing of values into a vector.

A block WHERE structure begins with a block WHERE statement and ends with an END WHERE statement; it can contain one OTHERWISE statement. The block WHERE statement can be followed by a block of vector assignment statements called a where-block. An OTHERWISE statement can be followed by a block of vector assignment statements called an otherwise-block.

A where-block or an otherwise-block can contain any number of vector assignment statements or it can contain no statements. No other types of statements can appear in a where-block or otherwise-block. All vector operands that appear in a where-block or otherwise-block must have the same length as the control vector that is established in the block WHERE statement. All vectors and vector expressions that appear in a where-block or otherwise-block must be of type integer or real. All vector expressions that appear in a where-block or otherwise-block can contain only addition, subtraction, multiplication, and division operations, and references to the vector functions VFLOAT, VIFIX, VINT, VAINT, VSQRT, VABS, and VIABS.

Control must not transfer into a where-block or otherwise-block.

A simple block WHERE structure is shown in figure 11-9. When the block WHERE statement is executed, the vector bit expression in the block WHERE statement is evaluated. The evaluation produces a control vector. A control vector is a bit vector that controls the storing of values into a vector. This control vector is used for all of the vector assignment statements that appear between the block WHERE statement and the next END WHERE statement.

```
WHERE (e)

    where-block

END WHERE
```

Figure 11-9. Simple Block WHERE Structure

The vector assignment statements in the where-block are then executed in order as follows:

1. The vector expression that appears in a vector assignment statement in the where-block is evaluated.

2. Each value of the result vector is assigned to the corresponding vector element on the left side of the vector assignment statement only if the corresponding element in the control vector contains a 1 bit. A value is not assigned to the corresponding vector element on the left side of the vector assignment statement if the corresponding element in the control vector contains a 0 bit. If a value is not assigned to a vector element, data flag branches are disabled for the operations that compute that value.

3. Steps 1 and 2 are repeated for each vector assignment statement in the where-block.

A block WHERE structure that contains an OTHERWISE statement is shown in figure 11-10. When the block WHERE statement is executed, the vector bit expression in the block WHERE statement is evaluated. The evaluation produces a control vector. A control vector is a bit vector that controls the storing of values into a vector. This control vector is used for all of the vector assignment statements that appear between the block WHERE statement and the next END WHERE statement.

```
WHERE (e)

    where-block

OTHERWISE

    otherwise-block

END WHERE
```

Figure 11-10. Block WHERE Structure With OTHERWISE Statement

The vector assignment statements in the where-block are then executed in order as follows:

1. The vector expression that appears in a vector assignment statement in the where-block is evaluated.

2. Each value of the result vector is assigned to the corresponding vector element on the left side of the vector assignment statement only if the corresponding element in the control vector contains a 1 bit. A value is not assigned to the corresponding vector element on the left side of the vector assignment statement if the corresponding element in the control vector contains a 0 bit. If a value is not assigned to a vector element, data flag branches are disabled for the operations that compute that value.

3. Steps 1 and 2 are repeated for each vector assignment statement in the where-block.

The vector assignment statements in the otherwise-block are then executed in order as follows:

1. The vector expression that appears in a vector assignment statement in the otherwise-block is evaluated.

2. Each value of the result vector is assigned to the corresponding vector element on the left side of the vector assignment statement only if the corresponding element in the control vector contains a 0 bit. A value is not assigned to the corresponding vector element on the left side of the vector assignment statement if the corresponding element in the control vector contains a 1 bit. If a value is not assigned to a vector element, data flag branches are disabled for the operations that compute that value.

3. Steps 1 and 2 are repeated for each vector assignment statement in the otherwise-block.

Given the declarations:

```
REAL A(5), B(5), C(5), D(5), E(5)
BIT CA(5), CB(5)
DATA A /3.0, 1.0, 6.0, 12.0, 12.0/
DATA B /4.0, 1.0, 8.0, 12.0, 16.0/
DATA C /5*0.0/, D /5*0.0/, E /5*0.0/
DATA CA /B'10101'/, CB /B'11111'/
```

the following are examples of block WHERE structures:

-
  ```
  WHERE (A(1;5).NE.B(1;5))
      E(1;5)=A(1;5)*A(1;5)+B(1;5)*B(1;5)
      C(1;5)=VSQRT(E(1;5);C(1;5))
      D(1;5)=A(1;5)*B(1;5)/2.0
  END WHERE
  ```

  The statements in the where-block cause the values 25.0, 100.0, and 400.0 to be assigned to the first, third, and fifth elements of vector E(1;5) respectively. They also cause the values 5.0, 10.0, and 20.0 to be assigned to the first, third, and fifth elements of vector C(1;5) respectively, and cause the values 6.0, 24.0, and 96.0 to be assigned to the first, third, and fifth elements of vector D(1;5) respectively. All other elements of vectors C(1;5) and D(1;5) are unchanged.

-
  ```
  WHERE (CA(1;5).AND.CB(1;5))
      E(1;5)=A(1;5)*A(1;5)+B(1;5)*B(1;5)
      C(1;5)=VSQRT(E(1;5);C(1;5))
      D(1;5)=A(1;5)*B(1;5)/2.0
  END WHERE
  ```

  Has the same effect as the previous example.

-
  ```
  WHERE (A(1;5).NE.B(1;5))
      E(1;5)=A(1;5)*A(1;5)+B(1;5)*B(1;5)
      C(1;5)=VSQRT(E(1;5);C(1;5))
      D(1;5)=A(1;5)*B(1;5)/2.0
  OTHERWISE
      E(1;5)=16.0
      C(1;5)=16.0
      D(1;5)=16.0
  END WHERE
  ```

The statements in the where-block cause the values 25.0, 100.0, and 400.0 to be assigned to the first, third, and fifth elements of vector E(1;5) respectively. They also cause the values 5.0, 10.0, and 20.0 to be assigned to the first, third, and fifth elements of vector C(1;5) respectively, and cause the values 6.0, 24.0, and 96.0 to be assigned to the first, third, and fifth elements of vector D(1;5) respectively.

The statements in the otherwise-block cause the value 16.0 to be assigned to the second and fourth elements of vectors C(1;5) and D(1;5).

NESTING BLOCK WHERE STRUCTURES

A block WHERE structure can appear in an if-block, else-block, or elseif-block of a block IF structure, but the entire block WHERE structure must appear in the if-block, else-block, or elseif-block.

A block WHERE structure can appear in the range of a DO loop, but the entire block WHERE structure must appear in the range of the DO loop. An END WHERE statement can be the terminal statement of a DO loop.

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.

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

Function	Function Reference	Type of	
		Arguments (other than c and i)[†††]	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 indefinite.

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:

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

let:

$$x = 2^n * w$$

where:

$$1/2 \le w < 1$$

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:

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

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:

$$c_0 = 1.00000000000000000172016224 * 10^0$$

$$c_1 = 3.33333333333327618176885283 * 10^{-1}$$

$$c_2 = 2.00000000030980778908993 07 * 10^{-1}$$

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

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

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

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

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|.

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=f1,BINARY=f2/l2,
    LIST=f3/l3/d3,OPTIONS=olist)
```

INPUT=f_1	Optional; f_1 is the name of the file containing the FORTRAN source program to be compiled. When the parameter is omitted, the default file name INPUT is used.
BINARY=f_2/l_2	Optional; f_2 is the name of the file that is to receive the compiler-generated object modules. l_2 is a specification of the length of f_2, and can be either an integer constant or a hexadecimal number prefixed with a #. l_2 can be omitted along with the slash. When the entire parameter is omitted, the default file name BINARY is used. When l_2 or the entire parameter is omitted, the default file length of 16 small pages is used.
LIST=$f_3/l_3/d_3$	Optional; f_3 is the name of the file that is to receive the compiler-generated listings and program output. l_3 is a specification of the length of f_3. Like l_2, l_3 can be either an integer constant or a hexadecimal number prefixed with a #. d_3 is the routing disposition of f_3 and must be PR (the line printer) or can be omitted (in which case no routing is performed). l_3 and d_3 can occur in either order. When l_3 is omitted, the default file size of 336 small pages is used. When the entire parameter is omitted, the default is OUTPUT.
OPTIONS=olist	Optional; olist is some logical combination of the compile option letters ABCEIKLMOR SUVYZ12, with the restriction that Y must not occur with any other option except L. Default olist is B. 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.

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.

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.

DIAGNOSTICS B

This appendix describes the four groups of diagnostic messages: compiler failure messages, compilation error messages, run-time error messages, and vectorizer messages.

COMPILER FAILURE AND COMPILATION ERRORS

Compiler failure messages are messages generated because of compiler failure. Compilation error messages 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. Some of the error numbers have no messages currently assigned to them. These error numbers are reserved for future use by CDC. 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).

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	F	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 refer ences 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 state ments; 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 statements in the main program appear as intended.
170		(Currently unassigned)	--	--
171		(Currently unassigned)	--	--
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 statement 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 statement; 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
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, else-block, or elseif-block.	Rewrite the statement so that it does not transfer control into an if-block, else-block, or elseif-block.
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.
412		(Currently unassigned)	—	—
413		(Currently unassigned)	—	—
414		(Currently unassigned)	—	—
415		(Currently unassigned)	—	—
416		(Currently unassigned)	—	—
417		(Currently unassigned)	—	—
418		(Currently unassigned)	—	—
419		(Currently unassigned)	—	—
420		(Currently unassigned)	—	—
421		(Currently unassigned)	—	—
422		(Currently unassigned)	—	—
423		(Currently unassigned)	—	—
424		(Currently unassigned)	—	—
425		(Currently unassigned)	—	—

TABLE B-2. COMPILATION ERROR MESSAGES (Contd)

Error Number	Type	Message	Significance	Action
426		(Currently unassigned)	—	--
427		(Currently unassigned)	—	--
428	F	BRANCH INTO THE RANGE OF A WHERE	Control must not transfer into a where-block or otherwise-block.	Rewrite the program so that it does not transfer control into a where-block or otherwise-block; recompile.
429	F	ILLEGAL VECTOR OPERATION IN THE RANGE OF A WHERE	A vector assignment statement that appears in a WHERE statement, where-block, or otherwise-block contains an invalid operator or function reference.	Remove or rewrite the statement; recompile.
430	F	MISSING ENDWHERE	Each block WHERE statement must have a corresponding END WHERE statement.	Supply the missing END WHERE statement; recompile.
431	F	WHERE EXPRESSION MUST BE OF TYPE BIT	The expression in the WHERE statement or block WHERE statement is not of type bit.	Supply an expression of type bit; recompile.
432	F	MISSING BLOCK WHERE	An OTHERWISE statement or an END WHERE statement appears without a corresponding block WHERE statement.	Check for mismatched or missing block WHERE statement; recompile.
433	F	EXTRA OTHERWISE	Only one OTHERWISE statement can appear in a block WHERE structure.	Rewrite block WHERE structure using no more than one OTHERWISE statement; recompile.
434	F	ILLEGAL STATEMENT IN THE RANGE OF A WHERE	Only vector assignment statements of type integer or real can appear in a where-block an otherwise-block or the vector assignment statement portion of a WHERE statement.	Remove or rewrite the illegal statements; recompile.
435	F	TERMINAL STATEMENT OF DO WITHIN RANGE OF A WHERE	If a block WHERE structure appears in the range of a DO statement, the entire block WHERE structure must appear in the range of the DO statement.	Move the terminal statement of the DO loop so that it is on or after the END WHERE statement of the block WHERE structure; recompile.

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 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, andcause 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. Some of the error numbers have no messages currently assigned to them.

These error numbers are reserved for future use by CDC. 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
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		(Currently unassigned)	--	--
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		

GLOSSARY

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 the storing of values into a vector. The control vector elements are set to a configuration of 0s and 1s. Control vectors are used in WHERE statements, block WHERE structures, and some FORTRAN-supplied functions.

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.

CYBER 200 FORTRAN-SUPPLIED FUNCTIONS LIST E

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
ABS	NX	-
ACOS	X	FT_XACOS
AIMAG	NX	-
AINT	NX	-
ALOG	X	FT_XALOG
ALOG10	X	FT_XLOGT
AMAX0	NX	-
AMAX1	NX	-
AMIN0	NX	-
AMIN1	NX	-
AMOD	NX	-
ASIN	X	FT_XASIN
ATAN	X	FT_XATAN
ATAN2	X	FT_XATN2
CABS	NX	FT_XCABS
CCOS	X	FT_XCCOS
CEXP	X	FT_XCEXP
CLOG	X	FT_XCLOG
CMPLX	NX	-
CONJG	NX	-
COS	X	FT_XCOS
COSH	X	FT_XCOSH
COTAN	X	FT_XCOTN
CSIN	X	FT_SCSIN
CSQRT	X	FT_XCSQT
DABS	NX	-
DACOS	X	FT_XDACS
DASIN	X	FT_XDASN
DATAN	X	FT_XDATN
DATAN2	X	FT_XDTN2
DATE	X	-
DBLE	NX	-
DCOS	X	FT_XDCOS
DCOSH	X	FT_XDCSH
DDIM	NX	FT_XDDIM
DEXP	X	FT_XDEXP
DFLOAT	NX	-
DIM	NX	-
DINT	NX	-
DLOG	X	FT_XDLOG
DLOG10	X	FT_XDLGT
DMAX1	X	-
DMIN1	X	-
DMOD	X	FT_XDMOD
DPROD	NX	FT_XDPRD
DSIGN	NX	-
DSIN	X	FT_XDSIN
DSINH	X	FT_XDSNH
DSQRT	X	FT_XDSQT
DTAN	X	FT_XDTAN
DTANH	X	FT_XDTNH
EXP	X	-
FLOAT	NX	-
IABS	NX	-

TABLE E-1. SUPPLIED FUNCTIONS (Contd)

Function	Category	Fast Call Name	Function	Category	Fast Call Name
IDIM	NX	-	Q8VGATHR	NX	-
IDINT	NX	-	Q8VGEI	N	-
IFIX	NX	-	Q8VINTL	N	-
INT	NX	-	Q8VLTI	N	-
ISIGN	NX	-	Q8VMASK	N	-
MAX0	NX	-	Q8VMERG	N	-
MAX1	NX	-	Q8VMKO	N	-
MIN0	NX	-	Q8VMKZ	N	-
MIN1	NX	-	Q8VNEI	N	-
MOD	NX	-	Q8VPOLY	N	-
Q8SCNT	N	-	Q8VREV	N	-
Q8SDFB	N	-	Q8VSCATP	NX	-
Q8SDOT	N	-	Q8VSCATR	NX	-
Q8SEQ	N	-	Q8VXPND	N	-
Q8SEXTB	N	-	RANF	NX	-
Q8SGE	N	-	REAL	NX	-
Q8SINSB	N	-	SECOND	X	-
Q8SLEN	N	-	SIGN	NX	-
Q8SLT	N	-	SIN	X	FT_XSIN
Q8SMAX	N	-	SINH	X	FT_XSINH
Q8SMAXI	N	-	SNGL	NX	-
Q8SMIN	N	-	SQRT	NX	FT_XSQRT
Q8SMINI	N	-	TAN	X	FT_XTAN
Q8SNE	N	-	TANH	X	FT_XTANH
Q8SPROD	N	-	TIME	X	-
Q8SSUM	N	-	VABS	NX	FT_XVABS
Q8VADJM	N	-	VACOS	X	FT_XVACS
Q8VAVG	N	-	VAIMAG	X	FT_XVAIM
Q8VAVGD	N	-	VAINT	NX	FT_XVAIN
Q8VCMPRS	N	-	VALOG	X	FT_XVLOG
Q8VCTRL	N	-	VALOG10	X	FT_XVLGT
Q8VDELT	N	-	VAMOD	X	FT_XVAMD
Q8VEQI	N	-	VASIN	X	FT_XVASN
Q8VGATHP	NX	-	VATAN	X	FT_XVATN

TABLE E-1. SUPPLIED FUNCTIONS (Contd)

Function	Category	Fast Call Name	Function	Category	Fast Call Name
VATAN2	X	FT_XVAT2	VFLOAT	NX	FT_XVFLT
VCABS	X	FT_XVCAB	VIABS	NX	FT_XVIAB
VCCOS	X	FT_XVCCS	VIDIM	X	FT_XVDIM
VCEXP	X	FT_XVCXP	VIFIX	NX	FT_XVFIX
VCLOG	X	FT_XVCLN	VINT	NX	FT_XVINT
VCMPLX	X	FT_XVCPX	VISIGN	X	FT_XVISN
VCONJG	X	FT_XVCJG	VMOD	X	FT_XVMOD
VCOS	X	FT_XVCOS	VREAL	X	FT_XVREL
VCSIN	X	FT_XVCSN	VSIGN	X	FT_XVSGN
VCSQRT	X	FT_XVCSR	VSIN	X	FT_XVSIN
VDBLE	X	FT_XVDBL	VSNGL	X	FT_XVSGL
VDIM	X	FT_XVDIM	VSQRT	NX	FT_XVSQT
VEXP	X	FT_XVEXP F_XVEXP†	VTAN	X	FT_XVTAN

N = In-Line

X = External

NX = In-line and external

†This entry point is used only when the 3 compile option is selected.

CYBER 200 FORTRAN STATEMENT SUMMARY F

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

		Page
integer	v = arithmetic expression	4-1
real	v = arithmetic expression	4-1
complex	v = arithmetic expression	4-1
double-precision	v = arithmetic expression	4-1
character	v = character expression	4-2
logical	v = logical expression	4-2
bit	v = bit expression	11-10

FLOW CONTROL STATEMENTS

	Page
GO TO s	5-1
GO TO iv[,](s[,...,s]) ASSIGN s TO iv	5-1
GO TO (s[,...,s])[,] iv	5-2
IF (arithmetic expression) s,s,s	5-2
IF (logical expression) statement	5-2

IF (logical expression) THEN
ELSE IF (logical expression) THEN
ELSE
END IF
	5-2

DO s iv = n,n[,n]	5-4
CONTINUE	5-5
PAUSE [five-digit integer constant] PAUSE [character constant]	5-5
STOP [five-digit integer constant] STOP [character constant]	5-5

SPECIFICATION STATEMENTS

	Page
IMPLICIT type (list of letters and ranges of letters) [,...,type (list of letters and ranges of letters)]	6-1
type pf [/initial value/] [,...,pf [/initial value/]]	6-1
CHARACTER [*K] pf [*k] [/initial value/] [,...,pf [*k] [/initial value/]]	6-1
DIMENSION a[,...,a]	6-2
ROWWISE a[,...,a]	6-2
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]]	6-2
EQUIVALENCE (va,va[,...,va]) [,...,(va,va[,...,va])]	6-3
EXTERNAL procedure name [,..., procedure name]	6-3
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	6-4
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.	6-6

PROCEDURE DEFINITION

	Page
PROGRAM [procedure name] [(fip[,...,fip])] where fip is a file information parameter	7-1
statement function name (arg[,...,arg]) = expression	7-2
[type] FUNCTION procedure name (arg[,...,arg]), except for type = CHARACTER	7-2

	Page
CHARACTER FUNCTION procedure name *K (arg[,...,arg]	7-2
SUBROUTINE procedure name [(arg[,...,arg])]	7-6
BLOCK DATA [subprogram name]	7-6

ENTRY procedure name [(arg[,...,arg])]
for subroutines
ENTRY procedure name (arg[,...,arg])
for functions
	7-6

RETURN [n]
for subroutines
RETURN
for functions
	5-6

CALL procedure name [(arg[,...,arg])]	5-6

INPUT/OUTPUT STATEMENTS

	Page
READ (u[,fmt] [,END=s] [,ERR=s]) [iolist] READ fmt [, iolist]	8-1
WRITE (u) iolist WRITE (u,fmt)[iolist]	8-2
PRINT fmt[,iolist]	8-2
PUNCH fmt[,iolist]	8-2
BUFFER IN (u,data transfer mode)(first location in buffer, last location in buffer)	8-1
BUFFER OUT (u, data transfer mode)(first location in buffer, last location in buffer)	8-1
ENCODE (record length,fmt,name of buffer of records) [iolist]	8-3
DECODE (record length,fmt,name of buffer of records) [iolist]	8-3
NAMELIST /group name/list of variables and arrays [.../group name/list of variables and arrays]	8-4
Namelist input: &group name Δ pn=constant list, pn=constant list,... &END	8-4
READ (u,group name) READ group name	8-4
WRITE (u,group name)	8-4
PRINT group name	8-4
PUNCH group name	8-4
REWIND u	8-5
BACKSPACE u	8-5
ENDFILE u	8-5
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	9-2
UNIT (u)	G-2
LENGTH (u)	G-2

ARRAY ASSIGNMENT

	Page
subarray = array expression	10-2
subarray = scalar expression	10-2

VECTOR STATEMENTS

	Page
ASSIGN d, vr ASSIGN d, .DYN. integer scalar expression	11-8
FREE	11-9
vr = vector arithmetic expression or scalar arithmetic expression	11-9
bit vr = vector bit expression	11-10
WHERE (vector bit expression) vector assignment statement	11-10
WHERE (vector bit expression)	11-11
OTHERWISE	11-11
END WHERE	11-11
DESCRIPTOR p [,..., p]	11-13
type FUNCTION procedure name (arg[,..., arg];*)	11-13
ENTRY procedure name (arg[,..., arg];*)	11-14
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)	11-13

INDEX

COMMENT SHEET

MANUAL TITLE: CYBER 200 FORTRAN Version 3 Reference Manual

PUBLICATION NO.: 60457040 **REVISION:** D

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

NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

FOLD ON DOTTED LINES AND TAPE

CUT ALONG LINE

AA3419 REV. 4/79 PRINTED IN U.S.A.

APE TAPE

OLD 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

OLD FOLD