FORTRAN 200 VERSION 1

FOR USE WITH
CDC® CYBER 200 VIRTUAL STORAGE
OPERATING SYSTEM
VERSION 2

REFERENCE MANUAL

60480200

REVISION RECORD

Revision	Description
A (06/30/83)	Prerelease.
B (11/30/83)	Manual released in conjunction with the official release of the product (PSR level 600). The manual content is the same as that of the prerelease version.
C (02/14/84)	Manual revised to include additional error messages and minor technical corrections (for PSR level 600).
D (03/30/84)	Manual updated for PSR level 607 release (VSOS 2.1.5).
E (04/01/85)	Manual updated for PSR level 631 release (VSOS 2.1.6).
F (10/31/85)	Manual revised to include the new features: dynamic file allocation with Q8NORED; the addition of the GO parameter on the FORTRAN control statement; and other minor technical corrections for PSR level 644 release (VSOS 2.2).
G (04/16/86)	Manual updated for PSR level 654 release (VSOS 2.2.5).
H (12/03/86)	Manual updated for PSR level 670 release (VSOS 2.3).
J (10/23/87)	Manual updated for PSR level 690 release (VSOS 2.3.5).

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

Address comments concerning this manual to:

CONTROL DATA CORPORATION
Technology and Publications Division
P. O. BOX 3492
SUNNYVALE, CALIFORNIA 94088-3492

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	-
Inside Front Cover	H
Title Page	-
ii	J
iii	J
iv	J
v/vi	H
vii thru ix	H
x	J
xi	J
xii thru xiv	H
xv/xvi	H
xvii	G
1-1	H
1-2	H
1-3	F
1-4	H
2-1	H
2-2	F
2-2.1/2-2.2	F
2-3	E
2-4	A
2-5	A
2-6	H
2-6.1/2-6.2	H
2-7	H
2-8	H
2-9	E
2-10	A
2-11	F
2-12	J
3-1 thru 3-3	A
3-4	H
3-4.1/3-4.2	H
3-5	F
3-6	A
3-7 thru 3-9	J
3-10 thru 3-14	A
4-1	A
4-2	A
4-3	H
4-4	H
4-5	A
4-6	E
4-7	A
5-1	H
5-2	H
5-3	E
5-4	A
5-5	H
5-6	J
5-7	E
5-8	H
5-9	A
6-1	H
6-2	C
6-2.1/6-2.2	C
6-3	A
6-4	H
6-5	C
6-6 thru 6-11	A
6-12	D
6-13 thru 6-15	A
6-16	D
6-17	F
6-18	C
6-19	A
6-20	A
6-21	C
6-22	C
6-23	H
6-24	C
6-25	E
6-26	C
6-27	C
6-28	H
6-29	E
6-30	E
6-31	A
6-32	A
6-33	C
6-34	E
6-35	C
6-36	H
6-37	C
6-38	A
6-39	A
6-40	H
6-41	A
6-42	F
6-43	A
6-44	E
6-45 thru 6-48	A
6-49 thru 6-52	E
6-53	D
6-54	J
6-55 thru 6-57	H
7-1	A
7-2	E
7-3 thru 7-6	A
7-7	F
7-8	F
7-8.1/7-8.2	F
7-9	A
7-10	A
7-11	F
7-12	A
8-1	J
8-2	A
8-3	E
8-4	E
9-1	E
9-2	A
9-3	F
9-4	A
9-5	E

Page	Revision
9-6	F
9-7	E
9-8 thru 9-12	A
9-13 thru 9-15	E
9-16	H
9-17	H
9-18 thru 9-21	J
9-22	H
9-23	H
9-24	J
10-1	D
10-2 thru 10-5	A
10-6	E
10-7	J
10-8	J
10-9	F
10-10	J
10-11 thru 10-17	D
10-18	A
10-19	A
10-20 thru 10-24	H
10-24.1/10-24.2	H
10-25 thru 10-28	H
10-28.1/10-28.2	E
10-29	H
10-30 thru 10-32	D
10-33 thru 10-37	A
10-38	J
10-38.1/10-38.2	J
10-39	D
10-40	D
10-41	G
10-42	G
10-43	D
11-1	E
11-2 thru 11-6	H
11-6.1/11-6.2	H
11-7 thru 11-9	A
11-10 thru 11-13	F
11-14	J
11-15	E
11-16 thru 11-18	C
11-19	E
12-1	A
12-2	G
12-3	E
12-4	G
12-4.1	J
12-4.2	G
12-4.3/12-4.4	G
12-5 thru 12-16	E
13-1	F
13-2	F
13-3 thru 13-5	H
13-6	F
13-7 thru 13-10	H
14-1 thru 14-4	G
14-5	H
14-6	H
14-6.1/14-6.2	J
14-7	H
14-8	A
14-9	A
14-10	H
14-11	H
14-12	J
14-13	A
14-14	A
14-15	H
14-16	J
A-1	H
A-2	H
B-1 thru B-8	A
B-9	E
B-10	C
B-11	J
B-12	J
B-12.1/B-12.2	J
B-13	F
B-14	H
B-15	E
B-16	F
B-17	H
B-18	F
B-19 thru B-25	E
B-26	F
B-26.1	G
B-26.2	G
B-27	A
B-28	F
B-29	J
B-30	J
B-30.1/B-30.2	H
B-31	E
B-32	F
B-33	F
B-34	A
B-35	J
B-36	J
B-37	A
B-38	A
B-39	C
B-40	J
B-40.1/B-40.2	G
B-41 thru B-43	J
B-44	H
C-1	A
C-2	A
C-3	H
D-1	F
D-2 thru D-5	A
E-1	J
E-2	H
E-3	A
F-1	H
F-2 thru F-7	F
G-1 thru G-10	J
Index-1 thru -13	J
Comment Sheet/Mailer	J
Inside Back Cover	H
Back Cover	-

PREFACE

This manual describes the CONTROL DATA® FORTRAN 200 programming language. FORTRAN 200 is available under the CDC® CYBER 200 Virtual Storage Operating System (VSOS) on CYBER 200 Computer Systems.

FORTRAN 200 is a superset of the American National Standards Institute FORTRAN language, which is described in ANSI document X3.9-1978. Many of the extensions of the FORTRAN 200 language enable you to use the vector processing capabilities of the CYBER 200 hardware.

Before using this manual, you should be familiar with the FORTRAN language in general. Familiarity with the Virtual Storage Operating System, CYBER 200 hardware, and vector processing concepts would be helpful.

Information relating to this manual can be found in the publications listed below.

Publication	Publication Number
CDC CYBER 200 Assembler Version 2 Reference Manual	60485010
CDC CYBER Model 205 Hardware Reference Manual	60256020
CDC CYBER 200 Virtual Storage Operating System Reference Manual, Volume 1 of 2	60459410
CDC CYBER 200 Virtual Storage Operating System Reference Manual, Volume 2 of 2	60459420
VSOS User's Guide for FORTRAN 200 Programmers	60455390

CDC manuals can be obtained from your local Control Data office. Sites within the United States can also order manuals from:

Control Data Corporation
Literature and Distribution Services
308 North Dale Street
St. Paul, Minnesota 55103

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

CONTENTS

APPENDIXES

INDEX

FIGURES

TABLES

NOTATIONS

Certain notations are used throughout this manual. The meanings of these notations are:

UPPERCASE	Uppercase letters in syntax represent elements, such as language keywords, that must appear exactly as shown.
lowercase	Lowercase letters in syntax represent entities that you must supply.
[]	Brackets surrounding an item in syntax denote that the item is optional.
{}	Braces surrounding two or more stacked items in syntax denote that one of the stacked items must be used.
Δ	A delta represents a blank that is not optional.
...	A horizontal ellipsis indicates that the previous item in syntax can be repeated more than once.
. . .	A vertical ellipsis indicates that one or more lines have been deleted from a listing or example.
Shading	Shading indicates features or parameters that are extensions or restrictions to standard FORTRAN as described in ANSI document X3.9-1978.

INTRODUCTION 1

The CONTROL DATA FORTRAN 200 compiler operates under control of the CYBER 200 Virtual Storage Operating System on the CYBER 200 series computer hardware. The FORTRAN 200 compiler translates FORTRAN source code into relocatable binary object code.

A FORTRAN program can be entered onto a file by using a card reader or a terminal, and a control statement can be used to invoke the FORTRAN compiler. The compiler reads the source code from the file and processes the program, generating a source listing and object code. The source listing is a list of source statements and any diagnostic information generated by the compiler. The object code can be loaded and executed by using control statements.

The FORTRAN 200 language is a superset of the standard FORTRAN language that is defined in the American National Standards Institute (ANSI) document ANSI X3.9-1978. The FORTRAN 200 compiler provides two types of extensions to the standard FORTRAN language:

- Extensions that are common in FORTRAN languages implemented on other Control Data computers
- Extensions that provide access to the vector processing capabilities of the CYBER 200 series computer hardware

This manual uses shading to indicate those features that are extensions to the standard FORTRAN language.

This section describes the general structure of a FORTRAN program and of FORTRAN statements.

PROGRAM STRUCTURE

A FORTRAN program consists of one or more separately-defined program units. A program unit, which consists of a series of comment lines and source lines containing FORTRAN statements, is either a main program or a subprogram. A FORTRAN program must contain one main program; it can also contain any number of subprograms.

A FORTRAN main program begins with an optional PROGRAM statement and ends with an END statement. If you omit the PROGRAM statement, the compiler supplies a default PROGRAM statement. The main program must end with an END statement. The PROGRAM statement and the END statement are described in section 7.

A FORTRAN subprogram begins with a SUBROUTINE, FUNCTION, or BLOCK DATA statement, and ends with an END statement. These statements are described in section 7. Subprograms can be written in languages other than FORTRAN, such as CYBER 200 assembly language, but special consideration must be given to the interface; see section 13 for more information.

The FORTRAN compiler provides a number of predefined subprograms that you can reference in your FORTRAN program. These subprograms are described in sections 10 through 13, and in appendix E.

See figure 1-1 for an example of a complete FORTRAN 200 program. The program in figure 1-1 consists of one main program; it does not contain any subprograms.

The program shown in figure 1-1 is written on a FORTRAN coding form. Each line of the coding form represents a source line that can be keypunched, or typed at a terminal.

STATEMENTS

There are two classes of statements in the FORTRAN language: executable statements and nonexecutable statements. Executable statements describe the operations that the compiled program performs. All the executable statements in a program together constitute the execution sequence of the program.

Executable statements do the following:

- Control the order in which the statements in the program execute
- Input, output, modify, and store data

Nonexecutable statements are not part of the execution sequence of the program. Instead, they perform the following functions:

- Describe the characteristics, arrangement, format, and initial values of data
- Classify program units
- Define entry points within subprograms
- Optionally, specify the file requirements of the program

See table 1-1 for a summary of the types of statements.

STATEMENT STRUCTURE

The FORTRAN 200 language is a fixed-format programming language; this means that the position of a FORTRAN statement in the source line is significant to the meaning of the statement. FORTRAN source lines can be a maximum of 96 columns long. FORTRAN source lines consist of four fixed fields:

- The label field is in columns 1 through 5 of each source line and can contain a statement label. Column 1 of the label field can also be

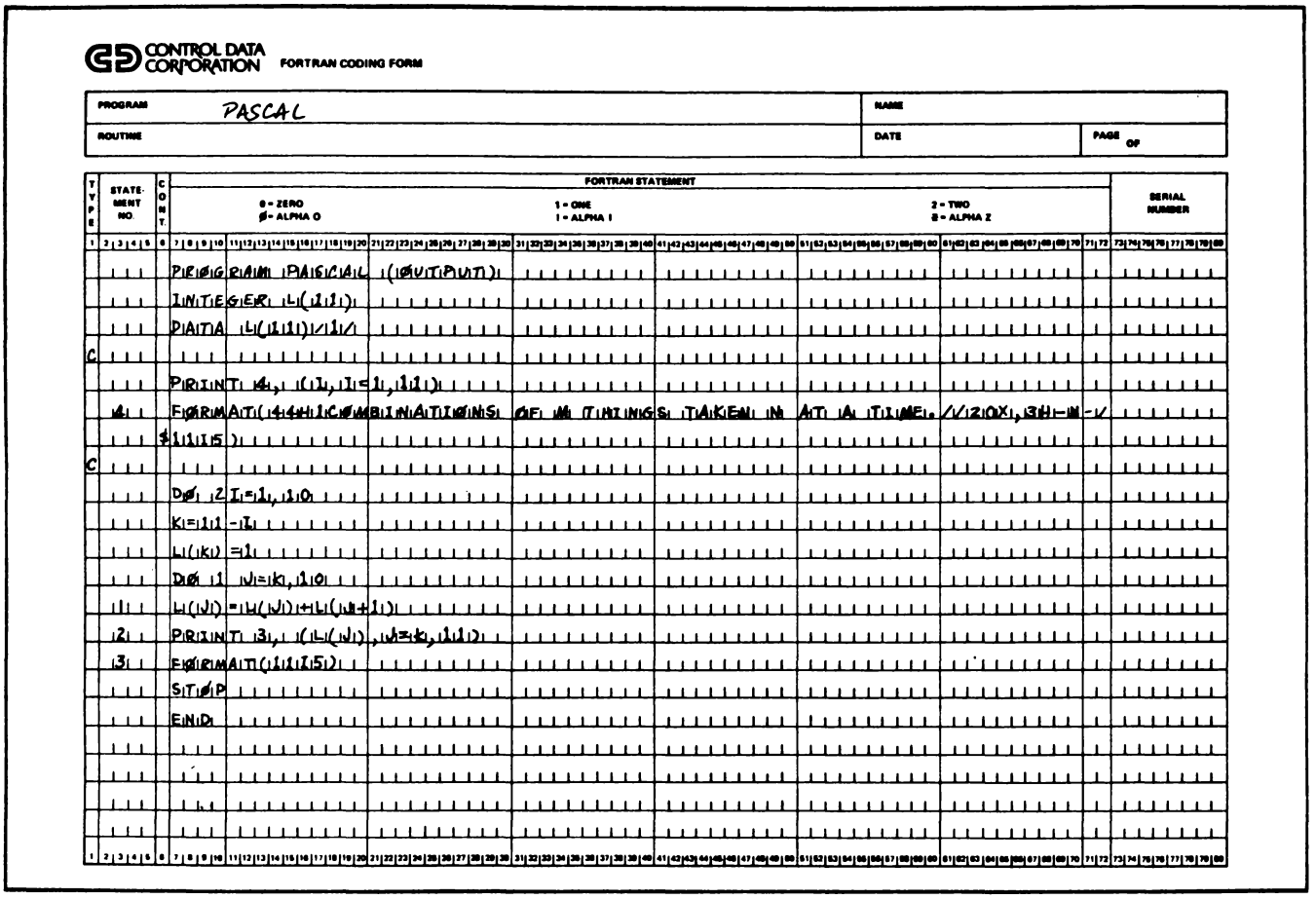


Figure 1-1. FORTRAN Program Example

TABLE 1-1. TYPES OF STATEMENTS

Executable	Nonexecutable
Assignment statements	Specification and initialization statements
Flow control statements	FORMAT and NAMELIST statements
Input/output statements	PROGRAM, FUNCTION, SUBROUTINE, and BLOCK DATA statements
END statement	

used to specify that the source line is a comment line. Statement labels and comments are described later in this section.

The continuation field is in column 6 of each source line and is used to specify that the source line is a continuation line for the statement that appears in the previous source line. Continuation lines are discussed later in this section.

The statement field is in columns 7 through 72 of each source line and contains the FORTRAN statement. The statement can appear anywhere in the statement field. Blanks in FORTRAN statements are ignored except in character constants and in Hollerith constants. If a statement is too long to be contained in the statement field of one source line, it can be continued in the statement field of subsequent source lines. Statement continuation is discussed later in this section.

The length of a source line can range from 80 columns (card input) to 96 columns. The identification field is ignored by the compiler; therefore, you can place any information in this field. The contents of the identification field are written on the source listing. One possible use of the identification field is to number the cards in a punched card deck.

See figure 1-2 for an illustration of the structure of FORTRAN statements.

STATEMENT LABELS

A statement label is a 1- through 5-digit integer. A statement label can appear in the label field of

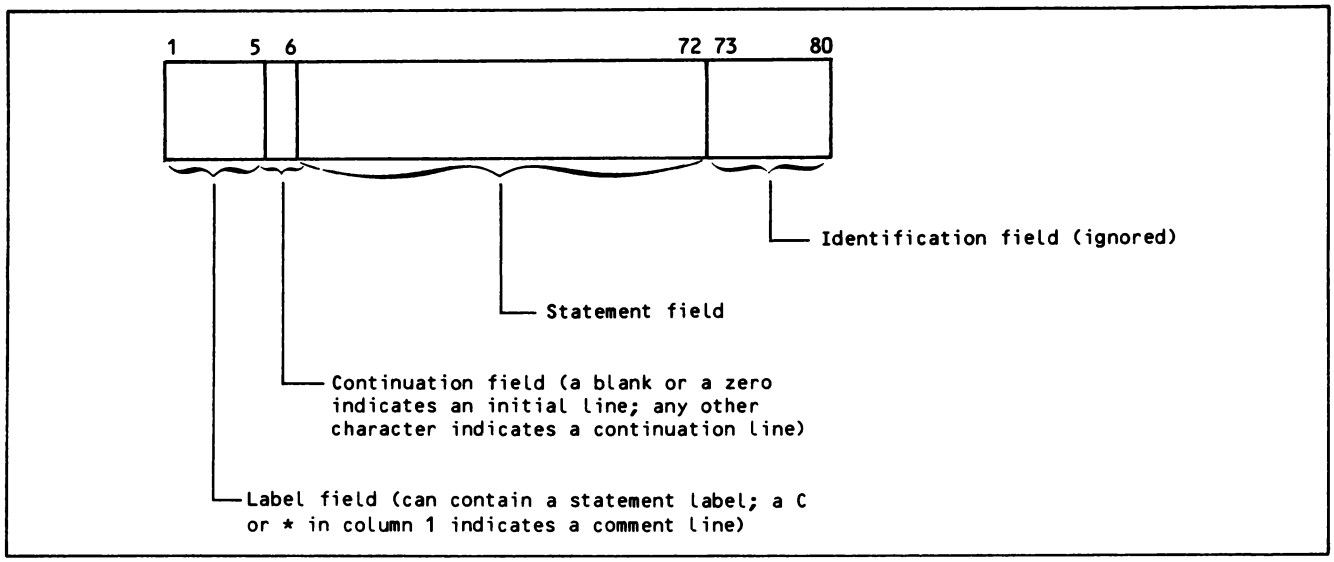


Figure 1-2. Statement Structure

a statement and identifies the statement so that it can be referenced from elsewhere in the program unit. A label can be referenced more than once, but it must not be defined more than once in a program unit. A label in one program unit cannot be referenced in another program unit. A statement does not need a label unless it is referenced in another statement. Blanks and leading zeros in labels are ignored. Labels on continuation statements are ignored. Labels do not have to appear in numerical order.

INITIAL LINES

An initial line is a source line in which a FORTRAN statement begins. An initial line can contain a statement label. Column 1 of an initial line must not contain the letter C or an asterisk. The continuation field of an initial line must contain a blank or a zero. The FORTRAN statement can appear anywhere in the statement field.

CONTINUATION LINES

A continuation line is a source line that contains a continuation of the statement that appears in the previous source line. You specify a source line to be a continuation line by placing a character other than a blank or zero in the continuation field.

The label field of a continuation line is ignored by the compiler; however, column 1 of a continuation line must not contain the letter C or an asterisk.

The statement field of a continuation line contains a portion of the FORTRAN statement that is not contained in the previous source line.

A FORTRAN statement can be continued on up to 19 continuation lines. The maximum length of a FORTRAN statement is 1320 characters. This is computed by multiplying 66 characters, which is the length of the statement field, by 20 lines (one initial line plus 19 continuation lines).

A continuation line can follow an initial line or another continuation line. Comment lines can appear between an initial line and a continuation line, and between two continuation lines.

COMMENT LINES

A comment line is a source line that can be used to document the program. You specify a source line to be a comment line by placing the letter C or an asterisk in column 1 of the source line. You can place any characters in the remaining columns of the source line. Any of the characters listed in appendix A can be used in comments, including those that are not in the FORTRAN character set.

Comment lines are printed on the source listing, but have no effect on program execution or on the object code produced by the compiler.

Source lines that contain blanks in columns 1 through 72 are considered to be comment lines.

Comment lines can appear anywhere in the program, including between an initial line and a continuation line, and between two continuation lines. Comment lines that are placed after an END statement are printed at the beginning of the next program unit.

STATEMENT ORDER

There are restrictions on the order in which statements can appear in a program unit. See figure 1-3 for an illustration of the statement order restrictions.

The figure shows that a PROGRAM, SUBROUTINE, FUNCTION, or BLOCK DATA statement must appear first in a program unit, although a comment line can precede the PROGRAM, SUBROUTINE, FUNCTION, or BLOCK DATA statement. If a PROGRAM statement is not specified in the main program, the compiler uses a default PROGRAM statement. Comment lines can appear anywhere in a program unit. Comment lines that

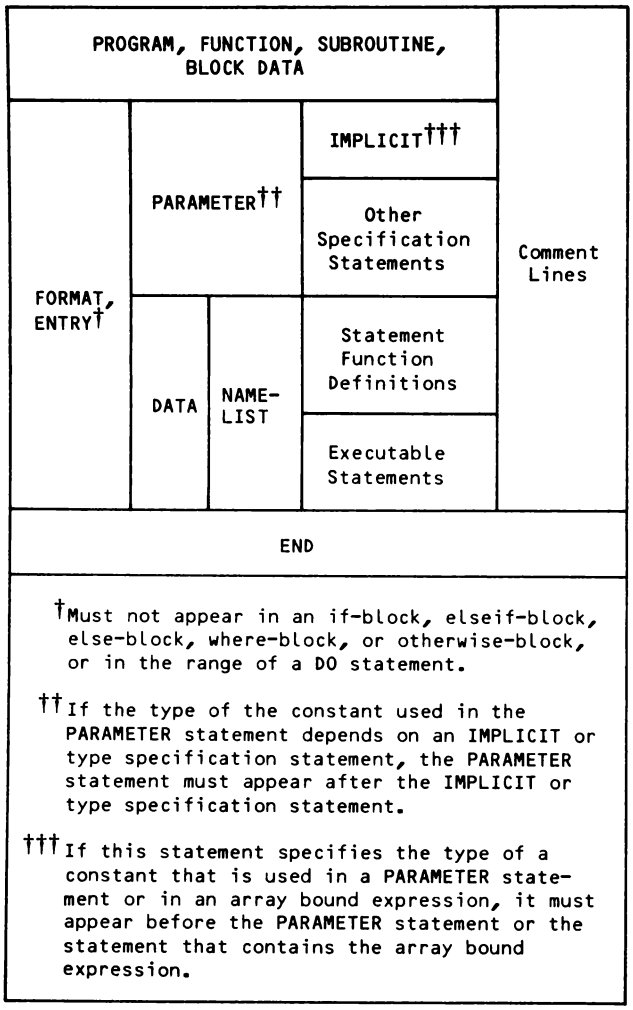


Figure 1-3. Statement Order

appear after an END statement are printed at the beginning of the next program unit.

FORMAT and ENTRY statements must appear before the END statement. The ENTRY statement must not appear in an if-block, elseif-block, else-block, where-block, or otherwise-block, or in the range of a DO statement.

PARAMETER statements must appear before any DATA statements. If the type of a constant that is used in a PARAMETER statement depends on an IMPLICIT or on a type specification statement, the IMPLICIT or type specification statement must precede the PARAMETER statement.

DATA statements must appear after any PARAMETER statements.

IMPLICIT statements must appear before any other specification statements (except PARAMETER statements), statement function definitions, and executable statements. If the type of a constant that is used in a PARAMETER statement depends on an IMPLICIT statement, the IMPLICIT statement must appear before the PARAMETER statement. If the type of an integer variable that is used in a dimension bound expression depends on an IMPLICIT statement, the IMPLICIT statement must appear before the statement in which the dimension bound expression appears.

Specification statements other than the IMPLICIT statement must appear after any IMPLICIT statements, and before any statement function definitions and executable statements. If the type of a constant that is used in a PARAMETER statement depends on a type specification statement, that type specification statement must appear before the PARAMETER statement. If the type of an integer variable that is used in a dimension bound expression depends on a type specification statement, that type specification statement must appear before the statement in which the dimension bound expression appears.

Statement function definitions must appear after any IMPLICIT and type specification statements, and must appear before any executable statements.

Executable statements must appear after any IMPLICIT and type specification statements, and statement function definitions.

The END statement must be the last statement in a program unit.

INPUT DATA

Input data are data that are transferred from an external medium, such as a disk or tape, to an area in memory that can be accessed by the program. Input data are not part of the source program record. Input statements cause data to be input to the program.

The data that appear on an input line can be in any format; there are no fixed fields for input data. The fields of an input line that are read by an input statement are determined by the input statement or by any associated FORMAT statement. See section 6 for more information about input data.

LANGUAGE ELEMENTS 2

This section describes the language elements that are used to construct FORTRAN 200 statements. The language elements include characters, symbolic names, keywords, constants, symbolic constants, variables, arrays, and substrings. This section also describes the internal representation of data.

CHARACTER SET

Any of the 52 characters listed in table 2-1 can be used in the syntax of a FORTRAN program. These characters are from the American Standard Character Set for Information Interchange (ASCII). CYBER 200 characters that are not listed in table 2-1 can be used in comments, character constants, and Hollerith constants.

TABLE 2-1. FORTRAN CHARACTER SET

Character Class	Characters
Alphabetic	Uppercase letters A through Z
Numeric	Digits 0 through 9
Special	Δ Blank or space = Equals sign + Plus sign - Minus sign or hyphen * Multiplication sign or asterisk / Division sign or slash (Left parenthesis) Right parenthesis , Comma . Decimal point or period & Ampersand ' Apostrophe : Colon ; Semicolon [Left bracket] Right bracket

See appendix A for the internal hexadecimal representations, printer graphic representations, and card punches for the CYBER 200 character set.

Some of the characters do not appear on all keypunches and terminals. If your keyboard lacks a character that you need, then use whatever character that is present which has the same internal hexadecimal representation as the needed character.

Blanks are not significant in FORTRAN statements except in character constants and Hollerith constants. Therefore, you can insert blanks in statements to make the program more readable. You can also insert blanks in language elements, such as symbolic names and constants. The symbol Δ is used in this manual to denote a blank that is not optional.

SYMBOLIC NAMES

A FORTRAN 200 symbolic name is user-supplied, can have up to eight letters and digits, and must begin with a letter. Symbolic names can be any of the following:

Main program name

Subroutine name

Function name

Block data subprogram name

Statement function name

Symbolic constant name

Variable name

Array name

Descriptor name

Descriptor array name

Common block name

Namelist group name

A symbolic name can be the same as a FORTRAN keyword. Conflicts can occur if a symbolic name is used to represent more than one program component. For example, a symbolic name must not be used as both a main program name and a variable name. Conflicts can also occur if a symbolic name duplicates the name of a predefined subroutine, the name of an intrinsic function, or a special call name.

See figure 2-1 for examples of legal and illegal symbolic names.

```
Legal symbolic names:

    LEGAL
    LEGAL1
    READ
    OKSYMBOL

Illegal symbolic names:

    ILLEGAL!
    1BADSYM
    SYMBOLTOOLONG
```

Figure 2-1. Symbolic Name Examples

FORTRAN KEYWORDS

A FORTRAN keyword is a name that has a special meaning to the FORTRAN compiler when used in the appropriate context. FORTRAN keywords are not reserved words; therefore, you can use the keywords as symbolic names.

Some of the characters in the FORTRAN character set have special meanings to the compiler when used in the appropriate context. For example, a comma used in a statement punctuates the statement.

CONSTANTS

A constant is a value that cannot be changed by the program. The 10 types of constants are: integer, half-precision, real, double-precision, complex, logical, character, Hollerith, hexadecimal, and bit.

An arithmetic constant is a constant that is of type integer, real, double-precision, half-precision or complex. An arithmetic constant can be either signed or unsigned. A signed constant is an arithmetic constant with a leading plus or minus sign. An unsigned constant is an arithmetic constant without a leading sign.

A constant is identified in a program by a symbolic name or by the constant value. A constant that is identified by a symbolic name is called a symbolic constant. Symbolic constants are described later in this section.

A constant that is identified by the constant value has a specific source program format that depends on the type of the constant. The source program formats of constants are described in the following paragraphs. The internal representation of each type of constant is described later in this section. See section 9 for a description of bit constants.

INTEGER CONSTANTS

An integer constant is a string of decimal digits that does not contain a decimal point or a comma. See figure 2-2 for the format of an integer constant. See figure 2-3 for examples of legal and illegal integer constants.

```
sign dec-digits

sign        A plus sign or a minus sign;
            optional. If a sign is not
            specified, the constant is
            positive.

dec-digits  A string of 1 to 14 of the decimal
            digits 0 through 9.
```

Figure 2-2. Integer Constants Format

```
Legal integer constants:

    0
    -1957
    1980
    12345678901234

Illegal integer constants:

    1.957
    123,456
    123456789012345
```

Figure 2-3. Integer Constants Examples

HALF-PRECISION CONSTANTS

A half-precision constant is a string of decimal digits that contains an exponent, or a decimal point and an exponent. A half-precision constant is written like a real constant or a double-precision constant except the letter S is used instead of the letter E or the letter D. The exponent portion of a half-precision constant must always be written. See figure 2-4 for the format of a half-precision constant.

```
sign man S sign exp

sign    A plus sign or a minus sign; optional.
        If a sign is not specified, the value
        that follows the sign is positive.

man     A string of one or more of the decimal
        digits 0 through 9 that represent the
        mantissa of the half-precision
        constant. One decimal point can appear
        anywhere in the string.

exp     A string of one or more of the decimal
        digits 0 through 9 that represent the
        base 10 exponent of the half-precision
        constant.
```

Figure 2-4. Half-Precision Constants Format

The value of a half-precision constant is the product of the mantissa and the result of 10 raised to the exponent. The minimum and maximum half-precision constants are approximately -2.177807S40 and 2.177807S40. See figure 2-5 for examples of legal and illegal half-precision constants. The values that the half-precision constants represent are shown in parentheses.

Legal half-precision constants:

8S10	$(8.0*10^{10})$
3.5S15	$(3.5*10^{15})$
4.2S-111	$(4.2*10^{-111})$
-3.5S-15	$(-3.5*10^{-15})$
-4.2S111	$(-4.2*10^{111})$

Illegal half-precision constants:

```
1957
1.957
1.957S1.957
200.9S
12.12.12S50
```

Figure 2-5. Half-Precision Constants Examples

REAL CONSTANTS

A real constant is a string of decimal digits that contains a decimal point or an exponent, or both. See figure 2-6 for the format of a real constant.

```
sign man E sign exp

sign    A plus sign or a minus sign; optional.
        If a sign is not specified, the value
        that follows the sign is positive.

man     A string of one or more of the decimal
        digits 0 through 9 that represent the
        mantissa of the real constant. One
        decimal point can appear anywhere in
        the string.

exp     A string of one or more of the decimal
        digits 0 through 9 that represent the
        base 10 exponent of the real constant;
        optional. If exp is not specified, the
        preceding E and sign must not be
        specified.
```

Figure 2-6. Real Constants Format

The value of a real constant is the product of the mantissa and the result of 10 raised to the exponent. See figure 2-7 for examples of legal and illegal real constants. The values that the legal constants represent are shown in parentheses. The minimum and maximum real constants are approximately -9.53E8644 and 9.53E8644.

```
Legal real constants:

    2E100                   (2.0*10^100)
    1.957                   (1.957)
    -19.84                  (-19.84)

    3.5E15                  (3.5*10^15)

    4.2E-111                (4.2*10^-111)

    -3.5E-15                (-3.5*10^-15)

    -4.2E111                (-4.2*10^111)

Illegal real constants:

    123,456.789
    1.957E1.957
    200.9E
    12.12.12E50
```

Figure 2-7. Real Constants Examples

DOUBLE-PRECISION CONSTANTS

A double-precision constant is a string of decimal digits that contains an exponent, or a decimal point and an exponent. A double-precision constant is written like a real constant or a half-precision constant except the letter D is used instead of the letter E or the letter S. The exponent portion of a double-precision constant must always be written. See figure 2-8 for the format of a double-precision constant.

The value of a double-precision constant is the product of the mantissa and the result of 10 raised to the exponent. See figure 2-9 for examples of legal and illegal double-precision constants. The values that the legal constants represent are shown in parentheses. The minimum and maximum double-precision constants are approximately -9.53D8644 and 9.53D8644, respectively.

```
sign man D sign exp

sign    A plus sign or a minus sign; optional.
        If a sign is not specified, the value
        that follows the sign is positive.

man     A string of one or more of the decimal
        digits 0 through 9 that represent the
        mantissa of the double-precision
        constant. One decimal point can appear
        anywhere in the string.

exp     A string of one or more of the decimal
        digits 0 through 9 that represent the
        base 10 exponent of the double-
        precision constant.
```

Figure 2-8. Double-Precision Constants Format

```
Legal double-precision constants:

    7D100                   (7.0*10^100)

    3.5D15                  (3.5*10^15)

    4.2D-111                (4.2*10^-111)

    -3.5D-15                (-3.5*10^-15)

    -4.2D111                (-4.2*10^111)

Illegal double-precision constants:

    1957
    1.957
    1.957D1.957
    200.9D
    12.12.12D50
```

Figure 2-9. Double-Precision Constants Examples

COMPLEX CONSTANTS

A complex constant is a pair of real or integer constants separated by a comma and enclosed in parentheses. See figure 2-10 for the format of a complex constant. See figure 2-11 for examples of legal and illegal complex constants.

```
(real-part,imag-part)

real-part       A real or integer constant that
                represents the real part of the
                complex constant

imag-part       A real or integer constant that
                represents the imaginary part of
                the complex constant
```

Figure 2-10. Complex Constants Format

```
Legal complex constants:

    (1957,1957)
    (1.957,3.5E15)
    (-4.2E-111,3.5E-15)

Illegal complex constants:

    1957,1957
    (1.957,3.5D15)
    (-4.2S-111,3.5E-15)
```

Figure 2-11. Complex Constants Examples

LOGICAL CONSTANTS

A logical constant is one of two specific strings of characters. See figure 2-12 for the format of a logical constant. The decimal points are part of the logical constant and must be written. See figure 2-13 for examples of legal and illegal logical constants.

```
.logical-value.

logical-value        One of the following
                     character strings:

                       TRUE
                       FALSE
```

Figure 2-12. Logical Constants Format

```
Legal logical constants:

    .TRUE.
    .FALSE.

Illegal logical constants:

    TRUE
    FALSE
    0
    1
```

Figure 2-13. Logical Constants Examples

CHARACTER CONSTANTS

A character constant is a string of one or more characters enclosed in apostrophes. See figure 2-14 for the format of a character constant.

```
'char-string'

char-string    A string of 1 through 65535
               characters from the CYBER 200
               character set
```

Figure 2-14. Character Constants Format

Blanks are significant in a character constant. Any of the characters listed in appendix A can be used in a character constant. In order to represent an apostrophe in a character constant, two consecutive apostrophes must be written.

Character constants, unlike Hollerith constants, can be used in character expressions and in character assignment statements. Character constants must not be used in arithmetic expressions or in arithmetic assignment statements.

See figure 2-15 for examples of legal and illegal character constants. The symbol Δ is used to denote blanks in the character constants shown.

```
Legal character constants:

    'LEGAL Δ CHARACTER Δ CONSTANT'
    '12345 Δ 67890 Δ'
    '!@#$%¢'
    'WHAT''S Δ UP?'

Illegal character constants:

    'ILLEGAL Δ CHARACTER Δ CONSTANT"
    'PI Δ IS Δ π'
```

Figure 2-15. Character Constants Examples

HOLLERITH CONSTANTS

A Hollerith constant is a string of one or more characters preceded by an unsigned integer and the letter H or the letter R. See figure 2-16 for the format of a Hollerith constant.

```
count H string
    or
count R string

count     An unsigned integer constant that
          specifies the exact number of
          characters in the Hollerith constant;
          count must be greater than 0 and no
          greater than 255.

string    A string of characters from the CYBER
          200 character set. This string begins
          in the next character position after
          the H or R and must contain exactly
          the number of characters specified in
          count.
```

Figure 2-16. Hollerith Constants Format

Blanks are significant in a Hollerith constant. Any of the characters listed in appendix A can be used in a Hollerith constant.

There are two types of Hollerith constants: H type and R type. An H type Hollerith constant is left-justified and blank-filled; an R type Hollerith constant is right-justified and binary-zero-filled. The internal representation of Hollerith constants is described later in this section.

Hollerith constants, unlike character constants, can be used in arithmetic expressions and in arithmetic assignment statements. Hollerith constants must not be used in character expressions or in character assignment statements.

See figure 2-17 for examples of legal and illegal Hollerith constants. The symbol Δ is used to denote blanks in the Hollerith constants shown.

```
Legal Hollerith constants:

    24HLEGAL Δ HOLLERITH Δ CONSTANT
    5R12345
    6H!@#$%¿
    10HWHAT'S Δ UP?

Illegal Hollerith constants:

    0HILLEGAL Δ HOLLERITH Δ CONSTANT
    7HPI Δ IS Δ π
```

Figure 2-17. Hollerith Constants Examples

HEXADECIMAL CONSTANTS

A hexadecimal constant is a string of hexadecimal digits enclosed in apostrophes and preceded by the letter X. See figure 2-18 for the format of a hexadecimal constant. See figure 2-19 for examples of legal and illegal hexadecimal constants.

```
X'hex-digits'

hex-digits    A string of 1 through 255 of the
              hexadecimal digits 0 through 9 and
              A through F. The hexadecimal
              digits correspond to the decimal
              values 0 through 15.
```

Figure 2-18. Hexadecimal Constants Format

BIT CONSTANTS

Bit constants are a vector programming feature of the FORTRAN 200 language. See section 9 for a description of bit constants.

SYMBOLIC CONSTANTS

A symbolic constant is a constant identified by a symbolic name. The value of a symbolic constant must not be changed by the program. A symbolic constant must be defined in a PARAMETER statement before it is used in a program. See section 3 for a description of the PARAMETER statement.

```
Legal hexadecimal constants:

    X'1957'
    X'ABCDEF'
    X'12A'

Illegal hexadecimal constants:

    X''
    X'WRONG'
    Z'12A'
```

Figure 2-19. Hexadecimal Constants Examples

The eight types of symbolic constants are: integer, half-precision, real, double-precision, complex, logical, character, and bit. The type of a symbolic constant is specified by the first letter of the symbolic name or by a type specification statement. If the type of a symbolic constant depends on a type specification or IMPLICIT statement, the type specification or IMPLICIT statement must appear before the PARAMETER statement that defines the constant. The internal representation of each type of symbolic constant is described later in this section.

Certain restrictions apply to symbolic constants. A symbolic constant must not appear as part of another constant. For example, if X is a real symbolic constant, (0.,X) is not a complex constant. A symbolic constant must not be used in a PROGRAM or FORMAT statement. A symbolic constant must not appear as input data.

See figure 2-20 for examples of symbolic constants. The program segment shown in figure 2-20 defines four symbolic constants:

MARY is a symbolic constant that represents the integer value 10.

CATHY is a symbolic constant that represents the real value 9.5.

KAREN is a symbolic constant that represents the character value SHY.

BETH is a symbolic constant the represents the logical value .FALSE..

```
      .
      .
      .
      CHARACTER*3 KAREN
      LOGICAL BETH
      PARAMETER(MARY=10,CATHY=9.5,KAREN='SHY',BETH=.FALSE.)
      .
      .
      .
```

Figure 2-20. Symbolic Constants Examples

CONSTANT EXPRESSIONS

A constant expression is an expression in which only constants (or symbolic constants) and operators are used. If an arithmetic expression is written using only constants and operators, the expression is an arithmetic constant expression. If a logical or character expression is written using only constants and operators, the expression is a logical constant expression or a character constant expression, respectively.

Note that variable, array element, and function references are not allowed. See section 4 for a complete discussion of expressions.

VARIABLES

A variable is an entity whose value can be changed during execution of the program. A variable is identified by a symbolic name, which is called a variable name. The FORTRAN compiler associates the variable name with a storage location; whenever the variable name is referenced in the program, the value that is stored in that storage location is referenced.

The eight types of variables are: integer, half-precision, real, double-precision, complex, logical, character, and bit. The type of a variable is specified by the first letter of the variable name or by a type specification statement. The internal representation of each type of variable is described later in this section.

See figure 2-21 for examples of variables. The program segment declares two variables whose types are specified by the first letter of the variable name:

NEPTUNE is an integer variable.

EARTH is a real variable.

```
.
.
.
CHARACTER PLUTO
LOGICAL MARS
.
.
.
NEPTUNE = 5
EARTH = 2.5
.
.
.
```

Figure 2-21. Variables Examples

The program segment also declares two variables whose types are specified by type specification statements:

PLUTO is a character variable.

MARS is a logical variable.

ARRAYS

An array is an ordered set of elements identified by a single symbolic name, which is called an array name. The value of each element of an array can be changed during program execution. The FORTRAN compiler associates each element of an array with a storage location; whenever an array element is referenced, the value that is in the corresponding storage location is referenced. Whenever an entire array is referenced, the values of all of the array elements are referenced.

The eight types of arrays are: integer, half-precision, real, double-precision, complex, logical, character, and bit. The type of an array is specified by the first letter of the array name or by a type specification statement. The internal representation of elements of each type of array is described later in this section.

ARRAY DECLARATION

An array must be declared in a program unit before it can be referenced in that program unit. All declarations of a particular array must be the same in all program units. An array can be declared only once in a program unit and can be declared in any one of the following statements:

DIMENSION statement

ROWWISE statement

COMMON statement

Any type specification statement

See section 3 for a description of these statements. An array declarator is used in these statements to specify the array name and the array size. See figure 2-22 for the format of an array declarator.

See figure 2-23 for examples of array declarations.

An array is treated exactly like an assumed size array, under the following conditions:

The array is a formal parameter.

No lower bound is specified.

The upper bound is one.

The ABC option is not selected (no subscript checking).

When the array is the first dimension bound declarator for a rowwise array and the last for a columnwise array.

For example:

```
FUNCTION F (A, B, C, N)
REAL A(N,1), B(1)
ROWWISE C(1,N)
```

is equivalent to:

```
FUNCTION F (A, B, C, N)
REAL A(N,*), B(*)
ROWWISE C(*,N)
```

```
a-name(dims)
    or
a-name(dims)*cl
    or
a-name*cl(dims)
```

a-name — A symbolic name that is the array name.

dims — A list of 1 to 7 dimension bound declarators separated by commas. Each dimension bound declarator has the following form:

lower:upper

- lower — An integer expression that specifies the lower bound of the dimension; optional. Any integer variables or integer array element references that appear in the expression must appear in the dummy argument list of the subprogram, or in a common block. Integer variables and integer array element references must not be used in the expression if the array declarator appears in the main program. If lower is not specified, the colon must not be written. If lower is not specified, the lower bound of the dimension is 1.
- upper — An integer expression that specifies the upper bound of the dimension. Any integer variables or integer array element references that appear in the expression must appear in the dummy argument list of the subprogram, or in a common block. Integer variables and integer array element references must not be used in the expression if the array declarator appears in the main program. In array declarators that do not appear in ROWWISE statements, the last upper in dims can be an asterisk. In array declarators that appear in ROWWISE statements, the first upper in dims can be an asterisk. The asterisk specifies that the upper bound of the dimension is unknown.

cl — An unsigned integer constant greater than 0, an integer constant expression enclosed in parentheses, an asterisk enclosed in parentheses, or a simple integer variable; optional. The cl specifies the length in characters of each element of a character array. If cl is an asterisk enclosed in parentheses, the array must be a dummy argument in a subprogram. The asterisk specifies that elements of the dummy array are the same length as those in the actual array. The array declarator forms that use cl can only be used when the array declarator appears in a CHARACTER statement.

Figure 2-22. Array Declaration Format

```
      .
      .
      .
      COMMON K
      DIMENSION MOOSE(2,5:7,*)
      REAL LION(-2:1,5:*)
      ROWWISE BEAR(*,2:3,2)
      ROWWISE MONKEY(3:*,5)
      CHARACTER CHIMP(10)*8
      .
      .
      .
      MOOSE(1,6,1) = MONKEY(3,2)
      LION(-2,20) = BEAR(1,2,1)
      .
      .
      .
      END
      SUBROUTINE SWINGERS(I,J,CHIMP)
      COMMON K
      DIMENSION GORILLA(I+3,J:K-5,*)
      CHARACTER CHIMP(10)*(*)
      .
      .
      .
      GORILLA(I,J,1) = 3.5E47
      .
      .
      .
      END
```

Figure 2-23. Array Declarations and References Examples

ARRAY REFERENCES

Particular elements of an array can be referenced in a program unit by specifying the array name and a list of subscripts. The subscripts specify the position of the element in the array. See figure 2-24 for the format of an array element reference.

The number of subscripts in an array element reference must be the same as the number of dimensions declared for the array in the array declaration. This restriction does not apply if the array element reference appears in an EQUIVALENCE statement.

See figure 2-23 for examples of array element references.

An entire array can be referenced by specifying the array name without subscripts. This causes all elements of the array to be referenced except when the array name appears in an EQUIVALENCE statement or in namelist input. When an array name without subscripts appears in an EQUIVALENCE statement or in namelist input, only the first element of the array is referenced.

```
a-name(subs)

a-name    A symbolic name that is the array
          name.

subs      A list of scalar arithmetic
          expressions of type integer, half-
          precision, real, or double-precision
          separated by commas. The number of
          subscripts in the list must be the
          same as the number of array
          declarators specified in the array
          declaration. The value of each
          subscript must not be less than the
          lower bound of the dimension or
          greater than the upper bound of the
          dimension. If an expression in subs
          is not integer, the result is
          truncated to an integer.
```

Figure 2-24. Array Element References Format

ARRAY SIZE

The size of an array depends on the number of elements specified in the array declaration and the type of the array.

The number of elements in an array is computed by multiplying the number of elements in each dimension. The number of elements in each dimension is computed by subtracting the lower bound from the upper bound and adding the value 1.

The amount of storage required for an array is computed by multiplying the number of elements in the array by the number of words required for each element. The number of words required for each element depends on the type of the array. See table 2-2 for the number of bits per array element for each data type; knowing that there are 64 bits per fullword, you can compute the number of fullwords per array element.

See figure 2-25 for the mathematical formulas used to compute the size of an array. See figure 2-26 for an example of an array size computation.

ARRAY STORAGE

Arrays can have one to seven dimensions; therefore, you can think of an array geometrically. For example, a 1-dimensional array can be thought of as a linear list, a 2-dimensional array can be thought of as a matrix, and a 3-dimensional array can be thought of as a series of matrices.

TABLE 2-2. ARRAY ELEMENT SIZE

Type of Array	Number of Bits Per Array Element
Integer	64
Real	64
Double-precision	128
Half-precision	32
Complex	128
Logical	64
Character	8 per character (the number of characters per element can be specified in the CHARACTER statement; if it is not specified, 1 character per element is used)
Bit	1

The number of elements in an array is computed by:

$$N = ((upper - lower + 1)_1 * \ldots * (upper - lower + 1)_n)$$

where:

N is the number of elements in the array, upper is the upper bound of the dimension, lower is the lower bound of the dimension, and n is the number of dimension declarators specified in the array declaration.

The amount of storage required for an array is computed by:

$$S = N * E$$

where:

S is the number of fullwords required for the array, N is the number of elements in the array, and E is the number of fullwords required for each array element.

Figure 2-25. Array Size Computation Formulas

GIVEN:

The following array declaration:

```
COMPLEX HOWBIG(5,3:5)
```

PROBLEM:

Find the number of elements in HOWBIG and the amount of storage required for HOWBIG.

SOLUTION:

The number of elements in HOWBIG is computed by:

```
N = ((5 - 1 + 1) * (5 - 3 + 1))
  = (5 * 3)
  = 15 elements
```

The amount of storage required for HOWBIG is computed by:

```
S = 15 elements * 2 fullwords per element
  = 30 fullwords
```

Figure 2-26. Array Size Computation Example

However, all arrays are stored internally as linear lists. Mathematical formulas are applied to the subscripts in order to translate the subscripts into a particular position in a linear list. Therefore, in order to determine the internal position of a particular array element, you must know the mathematical formulas used to map the subscripts into the linear list.

The mathematical formula used depends on the order in which array elements are stored in memory and the number of dimensions specified in the array declaration. There are two ways array elements can be stored: in columnwise order and in rowwise order.

The order of array elements depends on how the array is declared. If the array is declared in a DIMENSION, COMMON, or type specification statement, the elements are stored in columnwise order. If the array is declared in a ROWWISE statement, the elements are stored in rowwise order.

The order of elements in a columnwise array is determined by varying the subscripts through their entire range of values such that the leftmost subscript varies most rapidly. The order of elements in a rowwise array is determined by varying the subscripts through their entire range of values such that the rightmost subscript varies most rapidly. See table 2-3 for a comparison of columnwise and rowwise arrays.

TABLE 2-3. SUBSCRIPTING ORDER FOR A THREE-DIMENSIONAL ARRAY A(2,3,4)

ROWWISE Subscript Succession	Ordinality	Conventional Subscript Succession
A(1,1,1)	1	A(1,1,1)
A(1,1,2)	2	A(2,1,1)
A(1,1,3)	3	A(1,2,1)
A(1,1,4)	4	A(2,2,1)
A(1,2,1)	5	A(1,3,1)
A(1,2,2)	6	A(2,3,1)
A(1,2,3)	7	A(1,1,2)
A(1,2,4)	8	A(2,1,2)
A(1,3,1)	9	A(1,2,2)
A(1,3,2)	10	A(2,2,2)
A(1,3,3)	11	A(1,3,2)
A(1,3,4)	12	A(2,3,2)
A(2,1,1)	13	A(1,1,3)
A(2,1,2)	14	A(2,1,3)
A(2,1,3)	15	A(1,2,3)
A(2,1,4)	16	A(2,2,3)
A(2,2,1)	17	A(1,3,3)
A(2,2,2)	18	A(2,3,3)
A(2,2,3)	19	A(1,1,4)
A(2,2,4)	20	A(2,1,4)
A(2,3,1)	21	A(1,2,4)
A(2,3,2)	22	A(2,2,4)
A(2,3,3)	23	A(1,3,4)
A(2,3,4)	24	A(2,3,4)

In order to find the internal position of a particular array element, use the appropriate formula from table 2-4. The result of this formula indicates the position of the element in the internal linear list of elements. The elements are numbered beginning with 1. See figure 2-27 for an example of an array element position computation.

```
GIVEN:

The following array declarations:

    COMPLEX POSITION(5,3,2:4)

PROBLEM:

Find the position in the array of array element
POSITION(1,2,3).

SOLUTION:

The position of array element POSITION(1,2,3)
is computed by:

    position = 1+((1-1)+(5-1+1)*(2-1))+
               ((5-1+1)*(3-1+1)*(3-2))
             = 1+5+15
             = 21

Thus, POSITION(1,2,3) is element 21 of the
array.
```

Figure 2-27. Array Element Position Computation Example

SUBSTRINGS

A substring is a reference to a portion of a character string. The character string can be contained in a variable or in an array element. The variable name or the array name must be declared in a CHARACTER statement. See figure 2-28 for the format of a substring. See figure 2-29 for examples of substrings.

```
char-name(left-char:right-char)

char-name     A variable or array element of type
              character.

left-char     An integer expression greater than
              zero that specifies the character
              position in char-name of the first
              character in the substring;
              optional. The left-char value must
              be less than or equal to
              right-char. If left-char is not
              specified, the first character in
              char-name is the first character of
              the substring.

right-char    An integer expression greater than
              or equal to left-char that specifies
              the character position in char-name
              of the last character in the sub-
              string; optional. The right-char
              value must be less than or equal to
              the position of the last character
              in char-name. If right-char is not
              specified, the last character in
              char-name is the last character in
              the substring.
```

Figure 2-28. Substring Format

Variable name or array element reference	Contents of variable or array element	Substring reference	Characters referenced
FOX	BROWN	FOX(2:4)	ROW
CHICK(3)	YELLOW	CHICK(3)(4:6)	LOW
BEAVER	BLUE	BEAVER(:)	BLUE

Figure 2-29. Substring Examples

TABLE 2-4. ARRAY POSITION FORMULAS

Number of Dimensions and Dimension Declarator	Subscript	Array Element Location Formula
1 $(A_L:A_U)$	(a)	$1+(a-A_L)$
2 $(A_L:A_U,B_L:B_U)$ $(B_L:B_U,A_L:A_U)$	(a,b) (b,a)	$1+(a-A_L)+[(A_U-A_L+1)*(b-B_L)]$
3 $(A_L:A_U,B_L:B_U,C_L:C_U)$ $(C_L:C_U,B_L:B_U,A_L:A_U)$	(a,b,c) (c,b,a)	$1+(a-A_L)+[(A_U-A_L+1)*(b-B_L)]$ $+[(A_U-A_L+1)*(B_U-B_L+1)*(c-C_L)]$
4 $(A_L:A_U,B_L:B_U,C_L:C_U,D_L:D_U)$ $(D_L:D_U,C_L:C_U,B_L:B_U,A_L:A_U)$	(a,b,c,d) (d,c,b,a)	$1+(a-A_L)+[(A_U-A_L+1)*(b-B_L)]$ $+[(A_U-A_L+1)*(B_U-B_L+1)*(c-C_L)]$ $+[(A_U-A_L+1)*(B_U-B_L+1)*(C_U-C_L+1)*(d-D_L)]$
5 $(A_L:A_U,B_L:B_U,C_L:C_U,D_L:D_U,E_L:E_U)$ $(E_L:E_U,D_L:D_U,C_L:C_U,B_L:B_U,A_L:A_U)$	(a,b,c,d,e) (e,d,c,b,a)	$1+(a-A_L)+[(A_U-A_L+1)*(b-B_L)]$ $+[(A_U-A_L+1)*(B_U-B_L+1)*(c-C_L)]$ $+[(A_U-A_L+1)*(B_U-B_L+1)*(C_U-C_L+1)*(d-D_L)]$ $+[(A_U-A_L+1)*(B_U-B_L+1)*(C_U-C_L+1)*(D_U-D_L+1)*(e-E_L)]$
6 $(A_L:A_U,B_L:B_U,C_L:C_U,D_L:D_U,E_L:E_U,F_L:F_U)$ $(F_L:F_U,E_L:E_U,D_L:D_U,C_L:C_U,B_L:B_U,A_L:A_U)$	(a,b,c,d,e,f) (f,e,d,c,b,a)	$1+(a-A_L)+[(A_U-A_L+1)*(b-B_L)]$ $+[(A_U-A_L+1)*(B_U-B_L+1)*(C-C_L)]$ $+[(A_U-A_L+1)*(B_U-B_L+1)*(C_U-C_L+1)*(D-D_L)]$ $+[(A_U-A_L+1)*(B_U-B_L+1)*(C_U-C_L+1)*(D_U-D_L+1)*(e-E_L)]$ $+[(A_U-A_L+1)*(B_U-B_L+1)*(C_U-C_L+1)*(D_U-D_L+1)*(E_U-E_L+1)*(f-F_L)]$
7 $(A_L:A_U,B_L:B_U,C_L:C_U,D_L:D_U,E_L:E_U,F_L:F_U,G_L:G_U)$ $(G_L:G_U,F_L:F_U,E_L:E_U,D_L:D_U,C_L:C_U,B_L:B_U,A_L:A_U)$	(a,b,c,d,e,f,g) (g,f,e,d,c,b,a)	$1+(a-A_L)+[(A_U-A_L+1)*(b-B_L)]$ $+[(A_U-A_L+1)*(B_U-B_L+1)*(c-C_L)]$ $+[(A_U-A_L+1)*(B_U-B_L+1)*(C_U-C_L+1)*(d-D_L)]$ $+[(A_U-A_L+1)*(B_U-B_L+1)*(C_U-C_L+1)*(D_U-D_L+1)*(e-E_L)]$ $+[(A_U-A_L+1)*(B_U-B_L+1)*(C_U-C_L+1)*(D_U-D_L+1)*(E_U-E_L+1)*(F_U-F_L+1)*(g-G_L)]$

DATA ELEMENT REPRESENTATION

The following paragraphs describe the internal representation of data elements. A data element is a constant, variable, or array element. The way in which a data element is represented internally depends on the type of the data element.

A data element can have one of the following data types: integer, half-precision, real, double-precision, complex, logical, character, Hollerith, hexadecimal, or bit. A symbolic constant cannot be Hollerith, hexadecimal, or bit; a variable or an array cannot be Hollerith or hexadecimal.

See section 9 for the internal representation of bit elements.

INTEGER ELEMENTS

An integer element occupies one word of storage. Bits 0 through 15 are undefined; bits 16 through 63 contain the two's complement representation of the integer value. See figure 2-30 for a diagram of the internal representation of an integer element.

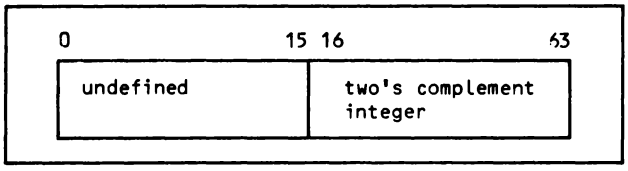


Figure 2-30. Integer Element Representation

The minimum and maximum integer elements are -2**47 and 2**47-1.

HALF-PRECISION ELEMENTS

A half-precision element occupies one-half word of storage. Bits 0 through 7 contain a two's complement integer that represents the binary exponent of the half-precision value; bits 8 through 31 contain a two's complement integer that represents the mantissa of the half-precision value. See figure 2-31 for a diagram of the internal representation of a half-precision element.

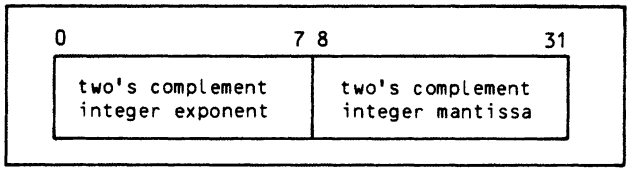


Figure 2-31. Half-Precision Element Representation

Half-precision elements are represented in normalized form: the most significant bit is always placed in bit 9, and the exponent is adjusted appropriately.

The minimum and maximum half-precision elements are approximately -2.177807S40 and 2.177807S40. The smallest half-precision value greater than zero that can be represented is 8.077936S-28; the largest half-precision value less than zero that can be represented is -8.077938S-28. Half-precision elements are precise to about seven decimal digits.

REAL ELEMENTS

A real element occupies one word of storage. Bits 0 through 15 contain a two's complement integer that represents the binary exponent of the real value; bits 16 through 63 contain a two's complement integer that represents the mantissa of the real value. See figure 2-32 for a diagram of the internal representation of a real element.

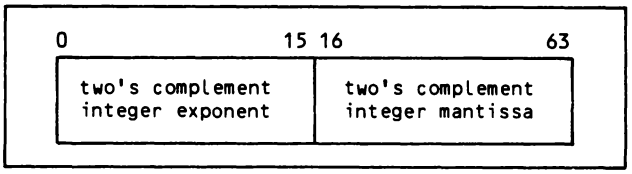


Figure 2-32. Real Element Representation

Real elements are represented in normalized form: the most significant bit of the mantissa is always placed in bit 17, and the exponent is adjusted appropriately.

The minimum and maximum real elements are approximately -9.53E8644 and 9.53E8644. The smallest real value greater than zero that can be represented is 5.19E-8618; the largest real value less than zero that can be represented is -5.19E-8618. Real elements are precise to about 14 decimal digits.

DOUBLE-PRECISION ELEMENTS

A double-precision element occupies two consecutive words of storage. The first word has the same format as a real data element; the first word expresses the most significant portion of the double-precision element. The second word has the same format as the first word except the exponent is 47 less than the exponent of the first word, and the mantissa is not normalized. The second word is always zero or positive. See figure 2-33 for a diagram of the internal representation of a double-precision element.

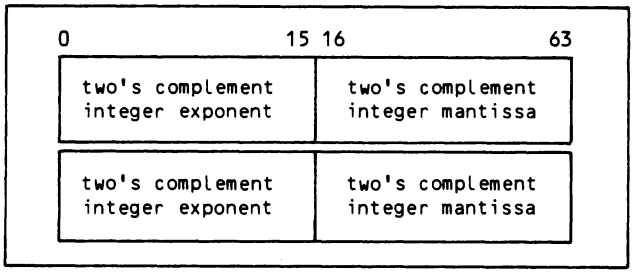


Figure 2-33. Double-Precision Element Representation

The first word of a double-precision element is represented in normalized form: the most significant bit is always placed in bit 17, and the exponent is adjusted appropriately.

The minimum and maximum double-precision elements are approximately -9.53D8644 and 9.53D8644. The smallest double-precision value greater than zero that can be represented is 5.19D-8618; the largest double-precision value less than zero that can be represented is -5.19D-8618. The smallest double-precision value greater than zero that can be used in comparison is 5.19D-8617; the largest double-precision value less than zero that can be used in comparison is -5.19D-8617. Double-precision elements are precise to about 28 decimal digits.

COMPLEX ELEMENTS

A complex element occupies two consecutive words of storage. Each word has the same format as real data elements. The first word represents the real part of the complex value; the second word represents the imaginary part of the complex value. See figure 2-34 for a diagram of the internal representation of a complex element.

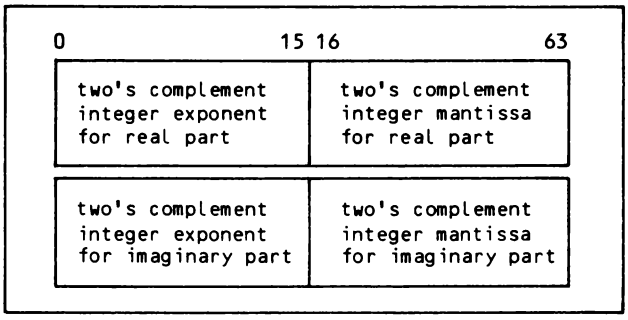


Figure 2-34. Complex Element Representation

LOGICAL ELEMENTS

A logical element occupies one word of storage. Bits 0 through 62 contain zeros; bit 63 contains either a 0 or a 1, corresponding to .FALSE. and .TRUE., respectively. See figure 2-35 for a diagram of the internal representation of a logical element.

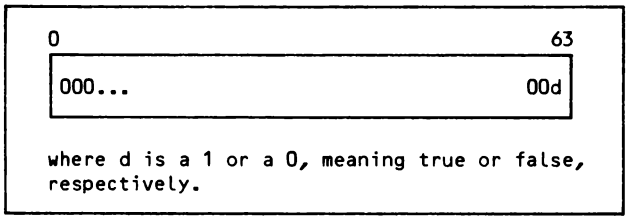


Figure 2-35. Logical Element Representation

HOLLERITH ELEMENTS

A Hollerith element occupies one byte for each character in the Hollerith value. The characters in a Hollerith element are stored in consecutive bytes. A byte is 8 bits. A word is 8 bytes.

An H type Hollerith element is left-justified in each word. If an H type Hollerith element does not completely fill a word, the unused portion of the word, which is on the right, is filled with blanks. An H type Hollerith element that is too long is truncated on the right.

An R type Hollerith element is right-justified in each word. If an R type Hollerith element does not completely fill a word, the unused portion of the word, which is on the left, is filled with binary zeros. An R type Hollerith element that is too long causes an error.

CHARACTER ELEMENTS

A character element occupies one byte for each character in the character value. The characters in a character element are stored in consecutive bytes. A byte is 8 bits. A word is 8 bytes.

A character element is left-justified in a variable or array element. If a character element does not completely fill a variable or array element, the unused portion, which is on the right, is filled with blanks. A character element that is too long is truncated on the right.

HEXADECIMAL ELEMENTS

A hexadecimal element occupies 4 bits for each hexadecimal digit in the hexadecimal value. Each hexadecimal digit in a hexadecimal element is stored in consecutive 4-bit groups.

A hexadecimal element is right-justified in a variable or array element. If a hexadecimal element does not completely fill a variable or array element, the unused portion, which is on the left, is filled with binary zeros.

BIT ELEMENTS

Bit elements are a vector programming feature of the FORTRAN 200 language. See section 9 for a description of the internal representation of bit elements.

Specification statements are nonexecutable statements that define the characteristics of symbolic names. Specification statements define the type of a symbolic name, the dimensions of an array, the length of a character variable or array element, and how storage is to be shared.

Any specification statements must appear before all DATA statements, NAMELIST statements, statement function definitions, and executable statements in the program unit.

This section describes each of the specification statements. The specification statements are:

Type specification statements

IMPLICIT statement

DIMENSION statement

ROWWISE statement

COMMON statement

EQUIVALENCE statement

EXTERNAL statement

INTRINSIC statement

SAVE statement

PARAMETER statement

DESCRIPTOR statement

The DESCRIPTOR statement is a vector programming feature of the CYBER 200 FORTRAN language. See section 9 for a description of the DESCRIPTOR statement.

This section also describes the DATA statement, which is a nonexecutable statement used for initialization. The DATA statement is not a specification statement.

TYPE SPECIFICATION STATEMENTS

A type specification statement is a specification statement that associates a list of symbolic names with a data type. A type specification statement can also be used to initialize variables and entire arrays. Initialization is described later in this section. The type specification statements are: INTEGER, HALF PRECISION, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, CHARACTER, and BIT.

If a type specification statement is not used to associate a symbolic name with a data type, the first letter of the symbolic name determines the data type with which the symbolic name is associated. Symbolic names that begin with the letters I, J, K, L, M, and N are associated with the integer data type. Symbolic names that begin with any other letter are associated with the real data type. This convention is called the first-letter rule.

The first-letter rule can be changed by using the IMPLICIT statement; the IMPLICIT statement is described later in this section.

The following symbolic names must be associated with a data type either by using a type specification statement or by using the first-letter rule:

Symbolic constant names

Variable names

Array names

Function names (except for intrinsic function names)

Descriptor names

Descriptor array names

The intrinsic function names have predefined types or have types that are determined by the actual arguments appearing in the function reference. The first-letter rule does not affect the intrinsic function names. An intrinsic function name need not appear in a type specification statement. If an intrinsic function name does appear in a type specification statement, the type specification statement has no effect on the predefined type of the intrinsic function name.

The following paragraphs describe each of the type specification statements.

INTEGER STATEMENT

The INTEGER statement can be used to associate a list of variable names, array names, symbolic constant names, and function names with the integer data type. The INTEGER statement can also be used to initialize variables and entire arrays. Initialization is described later in this section. See figure 3-1 for the format of the INTEGER statement.

INTEGER $v_1/d_1/$, ... ,$v_n/d_n/$

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

d_i An integer constant; optional. This specifies an initial value for v_i. Initialization is described later in this section. If v_i is a function name, d_i must be omitted. If d_i is omitted, the surrounding slashes must also be omitted.

Figure 3-1. INTEGER Statement Format

See figure 3-2 for an example of the INTEGER statement. The INTEGER statement in the example associates DOG, CAT, and MOUSE with the integer data type. The INTEGER statement also declares CAT to be an array of two elements, and intializes DOG to 1, CAT(1) to 2, and CAT(2) to 3. The PARAMETER statement declares MOUSE to be an integer symbolic constant having the value 4.

```
.
.
.
INTEGER DOG/1/,CAT(2)/2,3/,MOUSE
PARAMETER(MOUSE=4)
.
.
.
```

Figure 3-2. INTEGER Statement Example

HALF PRECISION STATEMENT

The HALF PRECISION statement can be used to associate a list of variable names, array names, symbolic constant names, and function names with the half-precision data type. The HALF PRECISION statement can also be used to initialize variables and entire arrays. Initialization is described later in this section. See figure 3-3 for the format of the HALF PRECISION statement.

HALF PRECISION $v_1/d_1/$, ... ,$v_n/d_n/$

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

d_i A half-precision constant; optional. This specifies an initial value for v_i. Initialization is described later in this section. If v_i is a function name, d_i must be omitted. If d_i is omitted, the surrounding slashes must also be omitted.

Figure 3-3. HALF PRECISION Statement Format

See figure 3-4 for an example of the HALF PRECISION statement. The HALF PRECISION statement in the example associates WOLF, FOX, and COYOTE with the half-precision data type. The HALF PRECISION statement also declares FOX to be an array of two elements, and initializes WOLF to 1.0S+10, FOX(1) to 2.0S+10, and FOX(2) to 3.0S+10. The PARAMETER statement declares COYOTE to be a half-precision symbolic constant having the value 4.0S+10.

REAL STATEMENT

The REAL statement can be used to associate a list of variable names, array names, symbolic constant names, and function names with the real data type. The REAL statement can also be used to initialize variables and entire arrays. Initialization is described later in this section. See figure 3-5 for the format of the REAL statement.

REAL $v_1/d_1/$, ... ,$v_n/d_n/$

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

d_i A real constant; optional. This specifies an initial value for v_i. Initialization is described later in this section. If v_i is a function name, d_i must be omitted. If d_i is omitted, the surrounding slashes must also be omitted.

Figure 3-5. REAL Statement Format

See figure 3-6 for an example of the REAL statement. The REAL statement in the example associates HORSE, COW, and SHEEP with the real data type. The REAL statement also declares COW to be an array of two elements, and intializes HORSE to 0.1, COW(1) to 1.1, and COW(2) to 2.1. The PARAMETER statement declares SHEEP to be a real symbolic constant having the value 3.1.

```
.
.
.
REAL HORSE/0.1/,COW(2)/1.1,2.1/,SHEEP
PARAMETER(SHEEP=3.1)
.
.
.
```

Figure 3-6. REAL Statement Example

DOUBLE PRECISION STATEMENT

The DOUBLE PRECISION statement can be used to associate a list of variable names, array names, symbolic constant names, and function names with the double-precision data type. The DOUBLE PRECISION statement can also be used to initialize variables and entire arrays. Initialization is described later in this section. See figure 3-7 for the format of the DOUBLE PRECISION statement.

```
.
.
.
HALF PRECISION WOLF/1.0S+10/,FOX(2)/2.0S+10,3.0S+10/,COYOTE
PARAMETER(COYOTE=4.0S+10)
.
.
.
```

Figure 3-4. HALF PRECISION Statement Example

DOUBLE PRECISION $v_1/d_1/$, ... ,$v_n/d_n/$

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

d_i A double-precision constant; optional. This specifies an initial value for v_i. Initialization is described later in this section. If v_i is a function name, d_i must be omitted. If d_i is omitted, the surrounding slashes must also be omitted.

Figure 3-7. DOUBLE PRECISION Statement Format

See figure 3-8 for an example of the DOUBLE PRECISION statement. The DOUBLE PRECISION statement in the example associates BEAVER, OTTER, and MUSKRAT with the double-precision data type. The DOUBLE PRECISION statement also declares OTTER to be an array of two elements, and initializes BEAVER to 2.0D+10, OTTER(1) to 3.0D+10, and OTTER(2) to 4.0D+10. The PARAMETER statement declares MUSKRAT to be a double-precision symbolic constant having the value 5.0D+10.

COMPLEX STATEMENT

The COMPLEX statement can be used to associate a list of variable names, array names, symbolic constant names, and function names with the complex data type. The COMPLEX statement can also be used to initialize variables and entire arrays. Initialization is described later in this section. See figure 3-9 for the format of the COMPLEX statement.

See figure 3-10 for an example of the COMPLEX statement. The COMPLEX statement in the example associates TADPOLE, FROG, and TOAD with the complex data type. The COMPLEX statement also declares FROG to be an array of two elements, and initializes TADPOLE to (0.0,0.0), FROG(1) to (1.0,1.0), and FROG(2) to (2.0,2.0). The PARAMETER statement declares TOAD to be a complex symbolic constant having the value (3.0,3.0).

LOGICAL STATEMENT

The LOGICAL statement can be used to associate a list of variable names, array names, symbolic constant names, and function names with the logical data type. The LOGICAL statement can also be used to initialize variables and entire arrays. Initialization is described later in this section. See figure 3-11 for the format of the LOGICAL statement.

```
.
.
.
DOUBLE PRECISION BEAVER/2.0D+10/,OTTER(2)/3.0D+10,4.0D+10/,MUSKRAT
PARAMETER(MUSKRAT=5.0D+10)
.
.
.
```

Figure 3-8. DOUBLE PRECISION Statement Example

COMPLEX $v_1/d_1/$, ... ,$v_n/d_n/$

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

d_i A complex constant; optional. This specifies an initial value for v_i. Initialization is described later in this section. If v_i is a function name, d_i must be omitted. If d_i is omitted, the surrounding slashes must also be omitted.

Figure 3-9. COMPLEX Statement Format

```
.
.
.
COMPLEX TADPOLE/(0.0,0.0)/,FROG(2)/(1.0,1.0),(2.0,2.0)/,TOAD
PARAMETER(TOAD=(3.0,3.0))
.
.
.
```

Figure 3-10. COMPLEX Statement Example

```
LOGICAL v1/d1/, ... ,vn/dn/
```

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

d_i A logical constant; optional. This specifies an initial value for v_i. Initialization is described later in this section. If v_i is a function name, d_i must be omitted. If d_i is omitted, the surrounding slashes must also be omitted.

Figure 3-11. LOGICAL Statement Format

See figure 3-12 for an example of the LOGICAL statement. The LOGICAL statement in the example associates FINCH, HERON, and PARROT with the logical data type. The LOGICAL statement also declares HERON to be an array of two elements, and initializes FINCH to .TRUE., HERON(1) to .TRUE., and HERON(2) to .FALSE.. The PARAMETER statement declares PARROT to be a logical symbolic constant having the value .FALSE..

CHARACTER STATEMENT

The CHARACTER statement can be used to associate a list of variable names, array names, symbolic constant names, and function names with the character data type. The CHARACTER statement can also be used to initialize variables and entire arrays. Initialization is described later in this section. See figure 3-13 for the format of the CHARACTER statement.

```
      .
      .
      .
LOGICAL FINCH/.TRUE./,HERON(2)/.TRUE.,.FALSE./,PARROT
PARAMETER(PARROT=.FALSE.)
      .
      .
      .
```

Figure 3-12. LOGICAL Statement Example

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

K Optional. Specifies the length in characters of each v_i (maximum character length is 65,535). The character length syntax is the same as k_i (described below).

The length specified in K is overridden by any k_i. If K is omitted, a length of 1 byte is used for each v_i that is not accompanied by a k_i (regardless of any length specification that appears in an IMPLICIT statement). If K is omitted, the preceding asterisk must also be omitted.

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

k_i Optional. Specifies the length in characters of a specific variable, v_i. The maximum character length is 65,535; k_i is either a constant or variable expression or an asterisk depending on v_i:

k_i is a constant expression and v_i is unrestricted; k_i must be parenthesized (enclosed in parentheses) unless it is a simple constant.

k_i is a variable expression and v_i must be a dummy argument; k_i must be parenthesized and all simple variables or arrays in k_i must be dummy arguments or in a common block.

k_i is a parenthesized asterisk (*) and v_i must be a dummy argument or symbolic constant name.

Any variable or symbolic constants in k_i must be associated with the integer data type before the CHARACTER statement.

If k_i is an expression, it is truncated to an integer character length. If k_i is an asterisk and v_i is a dummy argument, the length of v_i is the same as the length of the corresponding actual argument in the subprogram reference. If k_i is an asterisk and v_i is a symbolic constant name, the length of v_i is the length of its value in the PARAMETER statement.

If v_i is an array declarator, k_i must appear between the array name and the dimension specification.

If k_i is omitted, the length of v_i is determined by K. If k_i is omitted, the preceding asterisk must also be omitted.

d_i A character constant or a Hollerith constant; optional. This specifies an initial value for v_i. Initialization is described later in this section. If v_i is a function name, d_i must be omitted. If d_i is omitted, the surrounding slashes must also be omitted.

Figure 3-13. CHARACTER Statement Format

See figure 3-14 for examples of the CHARACTER statement. The CHARACTER statements in the example associate BIRD, FISH, ANIMAL, and TREE with the character data type. The CHARACTER statements also declare BIRD to be five characters long, FISH to be four, ANIMAL to be an array of two elements each of which is three characters long, and TREE to be five characters long. The CHARACTER statements initialize BIRD to the character value FINCH, FISH to the character value CARP, ANIMAL(1) to the character value CAT, and ANIMAL(2) to the character value DOG. The PARAMETER statement declares TREE to be a character symbolic constant having the value BIRCH.

```
      .
      .
      .
      CHARACTER*4 BIRD*5/'FINCH'/,FISH/'CARP'/
      CHARACTER ANIMAL*3(2)/'CAT','DOG'/,TREE*5
      PARAMETER(TREE='BIRCH')
      .
      .
      .
```

Figure 3-14. CHARACTER Statement Examples

BIT STATEMENT

The BIT statement is a vector programming feature of the CYBER 200 FORTRAN language. See section 9 for a description of the BIT statement.

IMPLICIT STATEMENT

The IMPLICIT statement is used to change the first-letter rule. The first-letter rule associates symbolic names with data types when the symbolic names do not appear in type specification statements. The first-letter rule normally functions as follows:

Symbolic names that begin with the letters I, J, K, L, M, and N are associated with the integer data type.

Symbolic names that begin with the letters A through H and O through Z are associated with the real data type.

The IMPLICIT statement changes the first-letter rule such that symbolic names that begin with the letters you specify are associated with the data types you specify. See figure 3-15 for the format of the IMPLICIT statement.

The IMPLICIT statement must precede all other specification statements except PARAMETER statements; however, if an IMPLICIT statement determines the type of a symbolic constant, the IMPLICIT statement must precede the PARAMETER statement that defines the symbolic constant.

An IMPLICIT statement that appears in a function or subroutine affects the data type of the dummy arguments and the function name as well as other symbolic names in the subprogram.

IMPLICIT typ_1 ($list_1$), ... ,typ_m ($list_m$)

typ_i The name of a data type: INTEGER, HALF PRECISION, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, CHARACTER, or BIT. If CHARACTER is used, you can specify the length after the data type name; see the description of the CHARACTER statement.

$list_i$ A list of the form:

$$v_1, \ldots, v_n$$

where v_i is a letter or a range of letters. A range of letters consists of two letters separated by a hyphen. Letters specified in a range must be in alphabetical order from left to right.

Figure 3-15. IMPLICIT Statement Format

An IMPLICIT statement must not associate a particular character with more than one data type. If a particular character is not associated with a data type in an IMPLICIT statement, the normal first-letter rule is used for that particular character. Thus, the IMPLICIT statement overrides the normal first-letter rule only for the characters you specify in the IMPLICIT statement.

The first letter of a symbolic name is used to associate a symbolic name with a data type only when the symbolic name is not associated with a data type by a type specification statement.

Intrinsic function names have predefined types or have types that are determined by the actual arguments appearing in the function reference. The first-letter rule does not affect intrinsic function names.

See figure 3-16 for an example of the IMPLICIT statement. The IMPLICIT statement in the example changes the first-letter rule such that the following statements are true:

All symbolic names beginning with the letters A and B that do not appear in a type specification statement are associated with the real data type.

All symbolic names beginning with the letter C that do not appear in a type specification statement are associated with the character data type. Each of these variables and array elements are eight characters long.

All symbolic names beginning with the letters D through K that do not appear in a type specification statement are associated with the real data type.

All symbolic names beginning with the letter L that do not appear in a type specification statement are associated with the logical data type.

```
      .
      .
      .
      PARAMETER(MOUSE=1)
      IMPLICIT CHARACTER*8(C),REAL(D-K),LOGICAL(L)
      .
      .
      .
      PARAMETER(LOST=.TRUE.)
      .
      .
      .
```

Figure 3-16. IMPLICIT Statement Example

All symbolic names beginning with the letters M and N that do not appear in a type specification statement are associated with the integer data type.

All symbolic names beginning with the letters O through Z that do not appear in a type specification statement are associated with the real data type.

The first PARAMETER statement in the example declares MOUSE to be an integer symbolic constant having the value 1. The second PARAMETER statement in the example declares LOST to be a logical symbolic constant having the value .TRUE..

DIMENSION STATEMENT

The DIMENSION statement declares symbolic names to be array names and specifies the bounds of each dimension. An array that is declared by using the DIMENSION statement is a columnwise array. See section 2 for a description of columnwise arrays. See figure 3-17 for the format of the DIMENSION statement.

DIMENSION a_1, ... ,a_n

a_i An array declarator. See section 2 for a description of array declarators.

Figure 3-17. DIMENSION Statement Format

An array declarator that appears in a DIMENSION statement must not appear in any other statement in the program unit. However, an array name that is declared in a DIMENSION statement can appear without an array declarator in a type specification statement or in a COMMON statement.

See figure 3-18 for an example of the DIMENSION statement. The DIMENSION statement in the example declares HEN to be a columnwise array of ten elements and CHICK to be a columnwise array of five elements. Subsequently, the INTEGER statement associates the array HEN with the integer data type and the COMMON statement places both HEN and CHICK in the unnamed common block.

```
      .
      .
      .
      DIMENSION HEN(10),CHICK(5)
      INTEGER HEN
      COMMON // HEN,CHICK
      .
      .
      .
```

Figure 3-18. DIMENSION Statement Example

ROWWISE STATEMENT

The ROWWISE statement declares symbolic names to be array names and specifies the bounds of each dimension. An array that is declared by using the ROWWISE statement is a rowwise array. See section 2 for a description of rowwise arrays. See figure 3-19 for the format of the ROWWISE statement.

ROWWISE a_1, ... ,a_n

a_i An array declarator. See section 2 for a description of array declarators.

Figure 3-19. ROWWISE Statement Format

An array declarator that appears in a ROWWISE statement must not appear in any other statement in the program unit. However, an array name that is declared in a ROWWISE statement can appear without an array declarator in a type specification or COMMON statement.

See figure 3-20 for an example of the ROWWISE statement. The ROWWISE statement in the example declares HEN to be a rowwise array of ten elements and CHICK to be a rowwise array of six elements. Subsequently, the INTEGER statement associates the array HEN with the integer data type and the COMMON statement places both HEN and CHICK in the unnamed common block.

```
      .
      .
      .
      ROWWISE HEN(2,5),CHICK(2,3)
      INTEGER HEN
      COMMON // HEN,CHICK
      .
      .
      .
```

Figure 3-20. ROWWISE Statement Example

COMMON STATEMENT

The COMMON statement is used to declare common blocks. A common block is an area of storage that can be referenced and defined by more than one program unit. See figure 3-21 for the format of the COMMON statement.

COMMON /blk_1/$list_1$, ... ,/blk_n/$list_n$

blk_i A symbolic name that represents the name of the common block. If blk_i is not specified, the COMMON statement defines the unnamed common block. If the first common block defined by a COMMON statement is the unnamed common block, the slashes can be omitted as well as blk_i.

$list_i$ A list of elements that are members of the block blk_i. The list is of the form:

u_1, ... ,u_m

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

The comma that follows $list_i$ can be omitted.

Figure 3-21. COMMON Statement Format

The two types of common blocks are named common blocks and unnamed common blocks. A named common block is identified by a symbolic name. A named common block that is shared between two or more program units must have the same name in each program unit. The name of a named common block can be the same as a variable name or an array name that is in the common block. The size of a named common block must be the same in all program units. Variables and arrays that are contained in a named common block can be initialized by DATA statements or by type specification statements. Initialization is described later in this section.

An unnamed common block, which is also called blank common, is not identified by a symbolic name. There can be only one unnamed common block in each program. The size of an unnamed common block does not have to be the same in all program units. Variables and arrays that are contained in an unnamed common block must not be initialized by DATA statements or by type specification statements.

Variables and entire arrays can be placed in a common block by using the COMMON statement. A variable or an array can appear in only one COMMON statement per program unit. An array element reference must not appear in a COMMON statement. Using both character and noncharacter variables in the same common block is allowed.

An array can be declared in a COMMON statement by specifying an array declarator in the COMMON statement. An array declarator that appears in a COMMON statement must not appear in any other statement in the program unit. However, an array name that is declared in a COMMON statement can appear without an array declarator in a type specification statement. Arrays that are declared in COMMON statements are columnwise arrays. See section 2 for a description of columnwise arrays.

Variables and arrays listed in the COMMON statement are stored in the order in which they appear in the COMMON statement. The first variable or array begins on a doubleword boundary. Subsequent variables or arrays begin in the first available byte after the previous variable or array; alignment is performed. See table 3-1 for the alignment of each type of variable and array.

TABLE 3-1. ALIGNMENT REQUIREMENTS

Type	Boundary
Integer	Fullword
Half-precision	Halfword
Real	Fullword
Double-precision	Fullword
Complex	Fullword
Logical	Fullword
Character	Byte
Bit	Bit

A common block name can appear more than once in a COMMON statement or in several COMMON statements in a program unit; the elements are stored cumulatively in the order of their occurrence in all COMMON statements in the program unit.

NOTE

The length of a named common block cannot exceed $2^{31}-1$ words.

The amount of storage required for each variable in a common block depends on the type of the variable. See the description of the internal representation of data elements in section 2. The amount of storage required for each array in a common block depends on the array size. See the discussion of array size in section 2.

The names of variables and arrays that appear in a common block do not have to be the same in each program unit that uses the common block. No type conversion is performed if the data types of corresponding elements of a common block are different in different program units. The name of a variable or array that appears in a named common block can be the same as the name of the common block.

If a program unit does not use all of the variables and arrays in a named or unnamed common block, placeholders can be inserted in the COMMON statement to force proper correspondence of the variables and arrays in the common block. Placeholders are variable names or array names that are not used in the program unit.

An unnamed common block does not have to be the same size in all program units that use it. Therefore, if a program unit does not use one or more variables or arrays that appear at the end of the common block, those variables and arrays do not have to be declared in the COMMON statement for that program unit. A named common block must always be the same size in all program units that use it, however.

The dummy arguments of a subprogram must not appear in a common block.

See figure 3-22 for examples of the COMMON statement. The COMMON statements in the example create two common blocks: the named common block PRECIOUS and the unnamed common block. Diagrams of each common block are given.

EQUIVALENCE STATEMENT

The EQUIVALENCE statement specifies that two or more variable names or array names in the same program unit identify the same storage location. See figure 3-23 for the format of the EQUIVALENCE statement.

An entire array cannot be referenced in an EQUIVALENCE statement; only individual array elements can be referenced. If you specify an array name without a subscript in an EQUIVALENCE statement, the first element of the array is referenced.

An array element can be referenced in an EQUIVALENCE statement by specifying the array name and a list of subscript expressions. The subscript expressions must be integer expressions that contain constants only. The number of subscript expressions specified must be the same as the number of array declarators specified in the array declaration.

Array elements can also be referenced in an EQUIVALENCE statement by specifying the array name and a single subscript expression. The subscript expression must be an integer expression that contains constants only. The single subscript is interpreted as though the subscript expression were the leftmost subscript and the missing subscript expressions each have their respective lower dimension bound value. See the description of array storage in section 2 for more information about the internal linear representation of arrays.

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

group$_i$ A list of the form:

v_1, ... ,v_n

where v_i is a variable name, array element, array name, or substring. Array declarators are not permitted. There must be at least two items specified in each group. Commas that separate the groups are optional.

Figure 3-23. EQUIVALENCE Statement Format

Two or more variables or arrays of different types can share the same storage location. If your program conforms to ANSI standards, storage alignment is handled automatically. If you use non-ANSI types, you must ensure that the variables and arrays are aligned on the proper boundary. See table 3-2 for the alignment requirements of variables and arrays. Note that character and noncharacter types can be equivalenced.

Equivalencing different data types does not cause type conversion or imply mathematical equivalence. Arithmetic operations on mixed data types can produce unexpected results.

```
PROGRAM STONES(INPUT,OUTPUT)
.
.
.
DIMENSION JADE(2)
COMMON JADE,SHALE /PRECIOUS/ DIAMOND(3),EMERALD
.
.
.
END
SUBROUTINE ROCKS
COMMON JADE1,JADE2
.
.
.
END
SUBROUTINE GEM
COMMON /PRECIOUS/ DIA,DUMMY(3)
.
.
.
END
```

Diagrams of common blocks:

Unnamed Common

JADE(1) JADE1	JADE(2) JADE2	SHALE

Common Block PRECIOUS

DIAMOND(1) DIA	DIAMOND(2) DUMMY(1)	DIAMOND(3) DUMMY(2)	EMERALD DUMMY(3)

Figure 3-22. COMMON Statement Examples

TABLE 3-2. ALIGNMENT REQUIREMENTS FOR EQUIVALENCE(X1,X2)

X1	X2			
	Integer Real Double-precision Complex Logical	Half-precision	Character	Bit
Integer Real Double-precision Complex Logical	Fullword	Fullword	Fullword	Fullword
Half-precision	Fullword	Halfword	Halfword	Halfword
Character	Fullword	Halfword	Byte	Byte
Bit	Fullword	Halfword	Byte	Bit

A variable or array that is in a common block can share storage with another variable or array; however, the variable or array with which it shares storage must not also be in a common block. An EQUIVALENCE statement can lengthen a common block as long as the common block is extended after the last variable or array in the common block. An EQUIVALENCE statement must not extend a common block before the first variable or array in the common block.

A dummy argument must not appear in an EQUIVALENCE statement.

An EQUIVALENCE statement must not be used to cause a single storage location to contain more than one element of the same array.

See figure 3-24 for examples of the EQUIVALENCE statement. The first EQUIVALENCE statement causes the variables VOLUME and GALLONS to share one storage location and SPEED and RATE to share another storage location. The second and third EQUIVALENCE statements cause three arrays to partially overlap. The overlapping arrays are shown in the figure.

```
.
.
.
INTEGER ARRAYA(10),ARRAYB(10),VARA(5)
EQUIVALENCE(VOLUME,GALLONS),(SPEED,RATE)
EQUIVALENCE(ARRAYA(1),ARRAYB(5))
EQUIVALENCE(VARA,ARRAYA)
.
.
.

Storage is shared by the overlapping arrays:

                    ARRAYB(1)
                    ARRAYB(2)
                    ARRAYB(3)
                    ARRAYB(4)
    ARRAYA(1)       ARRAYB(5)       VARA(1)
    ARRAYA(2)       ARRAYB(6)       VARA(2)
    ARRAYA(3)       ARRAYB(7)       VARA(3)
    ARRAYA(4)       ARRAYB(8)       VARA(4)
    ARRAYA(5)       ARRAYB(9)       VARA(5)
    ARRAYA(6)       ARRAYB(10)
    ARRAYA(7)
    ARRAYA(8)
    ARRAYA(9)
    ARRAYA(10)
```

Figure 3-24. EQUIVALENCE Statement Examples

EXTERNAL STATEMENT

The EXTERNAL statement specifies that a symbolic name is defined outside of the program unit that contains the EXTERNAL statement. The EXTERNAL statement can be used to accomplish any of the following:

To pass a subprogram name to another subprogram

To specify which version of an intrinsic function is to be used for those intrinsic functions that have both an inline version and an external version

To write and reference a subprogram that has the same name as an intrinsic function

See figure 3-25 for the format of the EXTERNAL statement.

EXTERNAL $proc_1$, ... ,$proc_n$

$proc_i$ The name of an external procedure, dummy procedure, or block data subprogram

Figure 3-25. EXTERNAL Statement Format

If a subprogram name appears in the actual argument list of a reference to another subprogram, the subprogram name must appear in an EXTERNAL statement.

If a function reference appears in the actual argument list of a subprogram reference, the name of the function does not have to appear in an EXTERNAL statement, however, because the result of the function reference is the actual argument, rather than the function name.

For an intrinsic function that has both an inline version and an external version, you can use the EXTERNAL statement to control which version of the intrinsic function is used. In order to cause the external version to be used, declare the intrinsic function name in an EXTERNAL statement and do not supply a function that has an entry point of the same name as the intrinsic function. If the intrinsic function name is not declared in an EXTERNAL statement, the inline version of the function is used. See section 10 for a description of inline and external intrinsic functions.

You can reference a function that you have supplied that has an entry point of the same name as an intrinsic function. To do this, you must declare the entry point name in an EXTERNAL statement and supply the function that has that entry point. If you declare the entry point name in an EXTERNAL statement and provide an entry point of that name, the intrinsic function cannot be referenced in that program unit.

A symbolic name that appears in an EXTERNAL statement must not be used as the name of an intrinsic function, statement function, variable, or array.

See figure 3-26 for an example of the EXTERNAL statement. The EXTERNAL statement enables the subprogram names SUBROU1 and SUBROU2 to be passed to the subroutine CALC.

```
      PROGRAM EXT(OUTPUT)
      EXTERNAL SUBROU1,SUBROU2
      .
      .
      .
      CALL CALC(SUBROU1,K)
      CALL CALC(SUBROU2,K)
      .
      .
      .
      END
      SUBROUTINE CALC(SUB,K)
      .
      .
      .
      CALL SUB
      .
      .
      .
      RETURN
      END
```

Figure 3-26. EXTERNAL Statement Example

INTRINSIC STATEMENT

The INTRINSIC statement specifies that a symbolic name is the name of a specific intrinsic function. The INTRINSIC statement can be used to enable a specific intrinsic function name to be passed as an argument to another subprogram. See figure 3-27 for the format of the INTRINSIC statement.

INTRINSIC int_1, ... ,int_n

int_i The name of an intrinsic function. See section 10 for a list of the intrinsic functions.

Figure 3-27. INTRINSIC Statement Format

If a specific intrinsic function name appears in the actual argument list of a reference to another subprogram, the specific intrinsic function name must be declared in an INTRINSIC statement.

If a specific intrinsic function reference appears in the actual argument list of a subprogram reference, the name of the specific intrinsic function does not have to be declared in an INTRINSIC statement, however, because the result of the function reference is the actual argument, rather than the function name.

The two kinds of intrinsic functions are specific intrinsic functions and generic intrinsic functions:

A specific intrinsic function accepts arguments of a particular type only. The name of a specific intrinsic function can appear in an INTRINSIC statement. The appearance of type conversion, lexical relationship, and maximum and minimum intrinsic functions in an INTRINSIC statement has no effect on their use as actual arguments.

A generic intrinsic function accepts arguments of various types. The name of a generic intrinsic function can appear in an INTRINSIC statement, but its appearance in an INTRINSIC statement has no effect on its use as an actual argument.

Some intrinsic functions are both generic and specific. The name of an intrinsic function that is both specific and generic can appear in an INTRINSIC statement, but the function name is assumed to be the specific function name. See section 10 for more information on intrinsic functions.

A symbolic name that appears in an INTRINSIC statement must not be used as the name of an external subprogram, statement function, variable, or array.

See figure 3-28 for an example of the INTRINSIC statement. The INTRINSIC statement in the example enables the specific function names SIN and COS to be passed to the subroutine CALC.

```
      PROGRAM INT(OUTPUT)
      INTRINSIC SIN,COS
      .
      .
      .
      CALL CALC(SIN,A,B)
      .
      .
      .
      CALL CALC(COS,A,B)
      .
      .
      .
      END
      SUBROUTINE CALC(SUB,A,B)
      B = SUB(A)
      RETURN
      END
```

Figure 3-28. INTRINSIC Statement Example

SAVE STATEMENT

The SAVE statement specifies that the values of variables and arrays in a subprogram are to be preserved after execution of the subprogram is completed. Normally, the values of variables and arrays are not preserved after the RETURN statement or the END statement of a subprogram is executed. See figure 3-29 for the format of the SAVE statement.

SAVE $name_1$, ... ,$name_n$

$name_i$	A variable name, array name, or common block name. If $name_i$ is a block name, then the block name must be enclosed in slashes, for example: SAVE /blockname/. Do not use slashes with the other two types. A particular name can appear in a SAVE statement only once per program unit and must not be a dummy argument, subprogram name, or common block member. A SAVE statement that lists no names is equivalent to one that lists all the names.

Figure 3-29. SAVE Statement Format

If no variable names, array names, or common block names are specified in a SAVE statement, the values of all variables, arrays, and common blocks that are accessible to the program unit in which the SAVE statement appears are preserved. If such a SAVE statement appears in a program unit, no other SAVE statements can appear in that program unit.

If a common block name is specified in a SAVE statement in a subprogram, the common block name must be specified in a SAVE statement in every subprogram that uses the common block.

See figure 3-30 for an example of the SAVE statement. The SAVE statement in the example causes the value of the variable TOTAL to be saved after execution of the subroutine SUM.

```
      SUBROUTINE SUM(A)
      SAVE TOTAL
      TOTAL = TOTAL + A
      IF(A .EQ. 0.0) GO TO 2
      PRINT 1 TOTAL
    2 RETURN
      END
```

Figure 3-30. SAVE Statement Example

PARAMETER STATEMENT

The PARAMETER statement defines the names and values of symbolic constants. A symbolic constant is a constant that is identified by a symbolic name. See figure 3-31 for the format of the PARAMETER statement.

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

$name_i$	A symbolic constant name; $name_i$ must be associated with a data type before it is used in a PARAMETER statement.
$value_i$	A constant or constant expression of type integer, half-precision, real, double-precision, complex, logical, character, or bit. If $name_i$ is of an arithmetic type, $value_i$ must be of an arithmetic type; the particular type can differ. If the particular arithmetic data types are different, $value_i$ is converted to the data type of $name_i$. If $name_i$ is not of an arithmetic type, $value_i$ must be of the same data type as $name_i$.

Figure 3-31. PARAMETER Statement Format

The eight types of symbolic constants are: integer, half-precision, real, double-precision, complex, logical, character, and bit. The type of a symbolic constant is specified by the first letter of the symbolic name or by a type specification statement. If the type of a symbolic constant depends on a type specification or IMPLICIT statement, the type specification or IMPLICIT statement must appear before the PARAMETER statement that defines the constant. See section 2 for a description of symbolic constants.

See figure 3-32 for an example of the PARAMETER statement. The PARAMETER statement in the example declares ONE to be an integer symbolic constant having the value 1, TWO to be an integer symbolic constant having the value 2, and POINT2 to be a real symbolic constant having the value 0.2.

```
      .
      .
      .
      INTEGER ONE,TWO
      REAL POINT2
      PARAMETER(ONE=1,TWO=2,POINT2=0.2)
      .
      .
      .
```

Figure 3-32. PARAMETER Statement Example

DESCRIPTOR STATEMENT

The DESCRIPTOR statement is a vector programming feature of the CYBER 200 FORTRAN language. See section 9 for a description of the DESCRIPTOR statement.

VARIABLE, ARRAY, AND SUBSTRING INITIALIZATION

Variables, arrays, and substrings can be initialized by using an initialization statement. The initial value that is assigned to a variable, array, or substring is the value that the variable, array, or substring has when execution of the compiled program begins.

If a variable, array, or substring is not initialized by an initialization statement, the value of the variable, array, or substring is not defined; therefore, the executing program itself must assign a value to the variable, array, or substring. A value must be assigned to a variable, array, or substring before the variable, array, or substring can be referenced.

Any variable, array, or substring can be initialized by an initialization statement unless it is a dummy argument, it is the same as the name of the function in which it appears, it appears in an unnamed common block, or it is equivalenced to a variable, array, or substring that appears in an unnamed common block. Variables, arrays, and substrings that appear in a named common block can be initialized by an initialization statement. A block data subprogram can be used for initialization of variables, arrays, and substrings that appear in named common blocks. See section 7 for a description of block data subprograms.

Variables and arrays can be initialized in one of two ways:

- By using a type specification statement
- By using a DATA statement

Substrings can be initialized only in DATA statements.

Each of these initialization methods are described in the following paragraphs. The rules for initialization are also described.

INTIALIZATION USING TYPE SPECIFICATION STATEMENTS

The type specification statements can be used to assign initial values to variables and arrays. The type specification statements are: INTEGER, HALF PRECISION, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, CHARACTER, and BIT. See the descriptions of these statements for their formats.

Initialization of variables and arrays using type specification statements differs from initialization using the DATA statement in the following ways:

- The initial value of a variable or array must appear immediately after the variable name or array name in the type specification statement. A list of names followed by a list of constants is not permitted.
- Only entire arrays can be initialized in type specification statements. Particular elements of an array cannot be initialized unless all elements are initialized.
- An implied DO loop cannot be used in a type specification statement. Implied DO loops are described later in this section.

The initialization rules are described later in this section. See the descriptions of the type specification statements for examples of variable and array initialization using type specification statements.

INITIALIZATION USING THE DATA STATEMENT

The DATA statement can be used to assign initial values to variables, arrays, and substrings. See figure 3-33 for the format of the DATA statement. An implied DO loop can appear in a DATA statement. See figure 3-34 for the format of an implied DO loop.

Initialization of variables and arrays using the DATA statement differs from initialization using type specification statements in the following ways:

- A list of variable names and array names followed by a list of constants can be specified in a DATA statement. The first variable or the first array element is assigned the value of the first constant in the list, the second variable or the second array element is assigned the value of the second constant in the list, and so on.
- Particular elements of an array can be initialized without initializing all elements of the array.

DATA $v_1/k_1/$, ... ,$v_m/k_m/$

v_i A variable list of the form:

w_1, ... ,w_n

where w_i is a variable, array element, array, substring, or implied DO loop. Subscript expressions and substring expressions used to identify array elements and substrings must be integer constant expressions, but subscript expressions can include references to the control variable of any containing implied DO loop.

k_i A data list of the form:

$j*d_1$, ... ,$j*d_m$

where d_i is an optionally signed constant. The $j*$ is an optional repeat specification; j must be an unsigned integer constant.

Figure 3-33. DATA Statement Format

An implied DO loop can be used in a DATA statement. An implied DO loop that appears in a DATA statement is processed in a manner similar to the way in which an implied DO loop in an input/output statement is processed, except that it has no effect on the definition status of the control variable of the implied DO. See section 6 for a description of implied DO loops in input/output statements.

INITIALIZATION RULES

Certain rules must be followed when you initialize variables, arrays, and substrings. Furthermore, there are special restrictions that apply to the initialization of variables and arrays of type bit.

The following paragraphs describe the rules for initializing variables, arrays, and substrings.

Initializing Non-Bit Items

The rules for initializing variables and arrays of type integer, half-precision, real, double-precision, complex, character, and logical, and for initializing substrings are:

If you specify a variable, array element, or substring in the list of variable names, array names, and substrings, you must specify one constant for each variable, array element, or substring.

If you specify an array name or an implied DO loop in the list of variable names, array names, array elements, and substrings, you must provide one constant for each element of the array that is to be initialized.

(list,cvar=$aexp_1$,$aexp_2$,$aexp_3$)

list A list of array elements and implied DO loops.

cvar A simple integer variable; cvar is used as the control variable for the implied DO loop. The control variable cvar must not also be the control variable of a containing implied DO loop. A DATA statement does not affect the definition status of any variable having the same name as cvar.

$aexp_1$ An arithmetic expression of type integer; $aexp_1$ is used as the initial value for the control variable. The expression can contain only constants and references to the control variables of containing implied DO loops.

$aexp_2$ An arithmetic expression of type integer; $aexp_2$ is used as the terminal value for the control variable. The expression can contain only constants and references to the control variables of containing implied DO loops.

$aexp_3$ An arithmetic expression of type integer; optional; $aexp_3$ is used as the incrementation value for the control variable. If $aexp_3$ is not specified, the incrementation value is 1. If the result of $aexp_3$ is positive, $aexp_1$ must be less than or equal to $aexp_2$. If the result of $aexp_3$ is negative, $aexp_1$ must be greater than or equal to $aexp_2$. The result of $aexp_3$ must not be zero. The expression can contain only constants and references to the control variables of containing implied DO loops.

Figure 3-34. Implied DO Loop Format for DATA Statements

Dummy arguments must not be initialized.

Variables and arrays that are in an unnamed common block must not be initialized.

If you initialize a variable, array element, or substring with a character constant or a Hollerith constant, the character constant or Hollerith constant is padded or truncated to the size of the variable or array element.

The type of a variable or array that is being initialized does not have to be the same type as the constant that is assigned to it by an initialization statement. See table 3-3 for the rules for mixed mode initialization.

If a variable or array of any type except bit is initialized with a bit constant, the constant is padded on the left with zero bits or truncated on the left to fit the variable.

TABLE 3-3. INITIALIZATION CONVERSIONS

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

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

See figure 3-35 for examples of initialization of non-bit variables and arrays using the DATA statement. The first DATA statement in the example initializes LION to 1, TIGER(1) to 2.0, TIGER(2) to 2.5, and BEAR(3) to POLAR. The second DATA statement in the example initializes the ten elements of array HUNTER to 0; the real constant 0.0 is converted to integer.

```
      .
      .
      .
      DIMENSION TIGER(2)
      CHARACTER*8 BEAR(10)
      INTEGER HUNTER(10)
      DATA LION,TIGER,BEAR(3)/1,2.0,2.5,'POLAR'/
      DATA (HUNTER(I),I=1,10)/10*0.0/
      .
      .
      .
```

Figure 3-35. DATA Statement Examples

Initializing Bit Items

Bit data items are a vector programming feature of the CYBER 200 FORTRAN language. See section 9 for a description of bit item initialization.

SCALAR EXPRESSIONS AND SCALAR ASSIGNMENT STATEMENTS

4

This section describes how expressions are written and evaluated and how values are assigned to variables and arrays. The expressions and assignment statements described in this section are scalar. See section 9 for a description of vector expressions and vector assignment statements.

SCALAR EXPRESSIONS

A scalar expression is a string of operators and scalar operands that defines the rules for computing a value. A scalar expression is evaluated during program execution. There are four kinds of scalar expressions:

Scalar arithmetic expressions

Scalar character expressions

Scalar relational expressions

Scalar logical expressions

Scalar expressions are described in the following paragraphs.

SCALAR ARITHMETIC EXPRESSIONS

A scalar arithmetic expression is an expression that yields a numeric value. A scalar arithmetic expression can appear in a scalar arithmetic assignment statement, scalar relational expression, vector arithmetic assignment statement, or vector relational expression. See figure 4-1 for the format of a scalar arithmetic expression.

The operators that can be used in a scalar arithmetic expression are called arithmetic operators. See table 4-1 for a list of the arithmetic operators.

The order in which a scalar arithmetic expression is evaluated depends on the precedence of the operators specified. The order of expression evaluation is described later in this section.

TABLE 4-1. ARITHMETIC OPERATORS

Operator	Operation
+	Addition or unary plus
-	Subtraction or unary minus
*	Multiplication
/	Division
**	Exponentiation

```
aexp

aexp    A scalar arithmetic expression of one
        of the forms:

           term
           + term
           - term
           aexp + term
           aexp - term

term    An arithmetic term of one of the forms:

           fact
           term * fact
           term/fact

fact    An arithmetic factor of one of the
        forms:

           prim
           prim ** fact

prim    An arithmetic primary, which can be
        any of the following:

           unsigned arithmetic constant
           arithmetic symbolic constant
           arithmetic variable
           arithmetic array element
           scalar arithmetic function
              reference
           scalar arithmetic expression
              enclosed in parentheses.
```

Figure 4-1. Scalar Arithmetic Expression Format

Operators that are mathematically associative or commutative might be reordered during compilation. You can force a definite ordering of mathematically associative operators of equal precedence by using parentheses. Expressions involving the division of integers are not reordered during compilation.

If the result of the division of two integers is not an integer, then the fractional portion of the result is discarded.

The appearance of an array element or a function reference in an expression requires the evaluation of the subscripts or the actual arguments. This evaluation does not affect the type of the expression result; however, the type of the actual arguments of some predefined generic functions affects the type of the function result. Evaluation of a function must not alter the value of any other element in the statement in which the function reference appears.

An expression that is not mathematically defined cannot be evaluated. For example, division by zero or the square root of a negative number must not be specified in an expression.

The operands that can appear in a scalar arithmetic expression are, in order of decreasing dominance:

Complex

Double-precision

Real

Half-precision

Integer

When an arithmetic operator functions on two operands of different types, the value of the dominated operand is converted to the type of the dominant operand before the operation is performed. The result is of the type of the dominant operand. See table 4-2 for the resulting data types for + - * / operations. See table 4-3 for resulting data types for ** operations.

See figure 4-2 for examples of scalar arithmetic expressions.

```
3.5
3.5 + N
-(3.5+N)/2**M
(XBAR+(B(I,J+I,K)/3.0))
-(C+DELTA*AERO)
(-Y-SQRT(Y**2-(4*A*C)))/(2.0*A)
GROSS-TAX*0.04
TEMP+V(M,AMAX1(A,Q))*Y**C/(H-FACT(K+3))


N, M, I, J, and K are integer variables, XBAR,
C, DELTA, AERO, A, GROSS, TAX, Q, TEMP, Y, and
H are real variables, B and FACT are real
arrays, and V is a real function.
```

Figure 4-2. Scalar Arithmetic Expression Examples

SCALAR CHARACTER EXPRESSIONS

A scalar character expression yields a character value. A scalar character expression can appear in a scalar character assignment statement and in a scalar relational expression. See figure 4-3 for the format of a scalar character expression.

TABLE 4-2. RESULT TYPE FOR ARITHMETIC OPERATIONS + - * /

Type of OP1	Type of OP2				
	Integer	Real	Double-Precision	Half-Precision	Complex
Integer	Integer	Real	Double-precision	Half-precision	Complex
Real	Real	Real	Double-precision	Real	Complex
Double-precision	Double-precision	Double-precision	Double-precision	Double-precision	Complex
Half-precision	Half-precision	Real	Double-precision	Half-precision	Complex
Complex	Complex	Complex	Complex	Complex	Complex

TABLE 4-3. RESULT TYPE FOR OP1 ** OP2

Type of OP1	Type of OP2				
	Integer	Real	Double-Precision	Half-Precision	Complex
Integer	Integer	Real	Double-precision	Half-precision	Complex
Real	Real	Real	Double-precision	Real	Complex
Double-precision	Double-precision	Double-precision	Double-precision	Double-precision	Complex
Half-precision	Half-precision	Real	Double-precision	Half-precision	Complex
Complex	Complex	Complex	Complex	Complex	Complex

```
cexp
    or
cexp // cexp


cexp    A character constant, character variable,
        character array element, character
        function reference, substring, or scalar
        character expression enclosed in
        parentheses
```

Figure 4-3. Scalar Character Expression Format

The operator that can be used in a scalar character expression is called a character operator. The character operator is the concatenation symbol //.

The order in which a scalar character expression is evaluated depends on the precedence of the operators specified. The order of expression evaluation is described later in this section.

All of the operands that appear in a scalar character expression must be of type character.

See figure 4-4 for examples of scalar character expressions.

```
'ING'
CHARVAR
CHARVAR(1:2)
CHARVAR // 'ING'
CHARARR(I)
CHARARR(I)(J:J+1)
CHARARR(I) // 'ING'
CHARFUN(X+Y)


CHARVAR is a character variable, CHARARR is a
character array, and CHARFUN is a character
function.
```

Figure 4-4. Scalar Character Expression Examples

SCALAR RELATIONAL EXPRESSIONS

A scalar relational expression yields a logical value. A scalar relational expression can appear in a scalar logical expression. See figure 4-5 for the format of a scalar relational expression.

```
aexp op aexp
    or
cexp op cexp


aexp     A scalar arithmetic expression

cexp     A scalar character expression

op       A relational operator
```

Figure 4-5. Scalar Relational Expression Format

The operators that can be used in a scalar relational expression are called relational operators. See table 4-4 for a list of the relational operators.

TABLE 4-4. RELATIONAL OPERATORS

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

The order in which a scalar relational expression is evaluated depends on the precedence of the operators specified. The order of expression evaluation is described later in this section.

A scalar relational expression that contains scalar arithmetic expressions is evaluated as follows:

1. Each scalar arithmetic expression is evaluated. The data types of the arithmetic expression results can be, in order of decreasing dominance:

   Complex

   Double-precision

   Real

   Half-precision

   Integer

2. If the types of the results of the scalar arithmetic expressions differ, the value of the dominated result is converted to the type of the dominant result.

3. The relational operation is then performed.

Scalar arithmetic expressions of type complex can appear in a scalar relational expression only when the operators .EQ. or .NE. are used.

When a scalar relational expression contains scalar character expressions, the corresponding characters in the results of the two expressions are compared one character at a time from left to right. The internal hexadecimal representations of the characters are compared; see appendix A for the internal hexadecimal representations of characters. If the results of the two scalar character expressions have different lengths, the shorter of the expressions is padded on the right with blanks until the lengths are equal.

See figure 4-6 for examples of scalar relational expressions.

```
SALARY .LT. EXPENSES
INDEX .EQ. LIMIT
X+Y/3.0*Z .NE. X
A(I) .GE. SQRT(R)
'STRING' .LE. CHARVAR


SALARY, EXPENSES, X, Y, Z, and R are real
variables, INDEX and LIMIT are integer
variables, A is a real array, and CHARVAR is a
character variable.
```

Figure 4-6. Scalar Relational Expression Example

SCALAR LOGICAL EXPRESSIONS

A scalar logical expression is an expression that yields a logical value. A scalar logical expression can appear in a scalar logical expression and in a scalar logical assignment statement. A scalar logical expression can be a single scalar relational expression. See figure 4-7 for the format of a scalar logical expression.

```
lexp
    or
lexp op lexp
    or
.NOT. lexp


lexp    A scalar relational expression, logical
        constant, logical variable, logical
        array element, logical function
        reference, or scalar logical expression
        enclosed in parentheses

op      One of the logical operators .AND.,
        .OR., .XOR., .EQV., or .NEQV.
```

Figure 4-7. Scalar Logical Expression Format

The operators that can be used in a scalar logical expression are called logical operators. See table 4-5 for a list of the logical operators. See table 4-6 for the truth table definitions of the logical operators.

The order in which a scalar logical expression is evaluated depends on the precedence of the operators specified. The order of expression evaluation is described later in this section.

TABLE 4-5. LOGICAL OPERATORS

Operator	Operation
.AND.	Logical and
.OR.	Logical inclusive or
.XOR.	Logical exclusive or
.NOT.	Logical negation
.EQV.	Logical equivalence
.NEQV.	Logical nonequivalence

If two .NOT. operators are adjacent to each other, the second .NOT. operator and its operand must be enclosed in parentheses. If a .NOT. operator is adjacent to any other logical operator, the .NOT. operator must appear to the right of the other logical operator.

The logical operators .AND., .OR., .XOR., .EQV., and .NEQV. must not appear adjacent to each other.

See figure 4-8 for examples of scalar logical expressions.

```
LX .EQV. LY
LX .AND. .NOT. LY
X+2.0.NE.Y/3.0.AND.LX.OR.LY
.NOT.(LX .AND. .NOT. LY)
.NOT.(.NOT.(LX .AND. .NOT. LY))
LOGVAR
LOGARR(I)
.TRUE.
.FALSE.


LX, LY, and LOGVAR are logical variables, LOGARR
is a logical array, and X and Y are real
variables.
```

Figure 4-8. Scalar Logical Expression Examples

TABLE 4-6. TRUTH TABLE DEFINITIONS OF LOGICAL OPERATORS

p	q	p.AND.q	p.OR.q	p.XOR.q	p.EQV.q	p.NEQV.q	.NOT.p
T	T	T	T	F	T	F	F
T	F	F	T	T	F	T	F
F	T	F	T	T	F	T	T
F	F	F	F	F	T	F	T

ORDER OF EXPRESSION EVALUATION

The order in which an expression is evaluated depends on the precedence of the operators specified in the expression. Operators of higher precedence are evaluated before operators of lower precedence. See table 4-7 for the precedence of operators.

TABLE 4-7. PRECEDENCE OF OPERATORS

Precedence	Operators	Category
First	**	Arithmetic
Second	*,/	Arithmetic
Third	+,-	Arithmetic
Fourth	//	Character
Fifth	.LT.,.LE.,.EQ., .NE.,.GE.,.GT.	Relational
Sixth	.NOT.	Logical
Seventh	.AND.	Logical
Eighth	.OR.	Logical
Ninth	.XOR.,.EQV.,. NEQV.	Logical

When arithmetic operators of equal precedence are specified in an arithmetic expression, the order in which those operators are evaluated can affect the result of the expression. When two or more operators of equal precedence appear in an arithmetic expression, the operators are evaluated in an order that is mathematically equivalent to evaluating them from left to right except for the exponentiation operator. When two exponentiation operators appear in an expression, the exponentiation operators are evaluated in an order that is mathematically equivalent to evaluating them from right to left.

You can change the order of expression evaluation by enclosing portions of the expression in parentheses. The portions of an expression that are enclosed in parentheses are evaluated first beginning with the deepest nesting of parentheses. The normal rules for expression evaluation apply within the parenthesized portions of an expression.

SCALAR ASSIGNMENT STATEMENTS

A scalar assignment statement is a statement that causes the result of a scalar expression to be assigned to a variable or an array element. A scalar assignment statement is performed during program execution. There are four kinds of scalar assignment statements:

Scalar arithmetic assignment statements

Scalar character assignment statements

Scalar logical assignment statements

Statement label assignment statements

Scalar assignment statements are described in the following paragraphs.

SCALAR ARITHMETIC ASSIGNMENT STATEMENTS

A scalar arithmetic assignment statement assigns the result of a scalar arithmetic expression to an arithmetic variable or an arithmetic array element. See figure 4-9 for the format of a scalar arithmetic assignment statement.

```
var = aexp

var     A simple variable or array element of
        type integer, half-precision, real,
        double-precision, or complex

aexp    A scalar arithmetic expression
```

Figure 4-9. Scalar Arithmetic Assignment Statement Format

If the type of the variable or array element that appears to the left of the equals sign differs from the type of the expression that appears to the right of the equals sign, type conversion is performed. The result of the expression is converted to the type of the variable or array element and replaces the value of the variable or array element. See table 4-8 for the rules for type conversion during arithmetic assignment.

See figure 4-10 for examples of scalar arithmetic assignment statements. The first, third, and fifth assignment statements in the example require no type conversion. The second assignment statement truncates the fractional part of 5.75 and assigns it to I; therefore, the value 5 is assigned to I. The fourth assignment statement converts the value of I to real and assigns it to A.

```
I = I + 1
I = 5.75
A = SQRT(B)
A = I
ARR(I) = ARR(J) + ARR(I+J)


I and J are integer variables, A and B are real
variables, and ARR is a real array.
```

Figure 4-10. Scalar Arithmetic Assignment Statement Examples

TABLE 4-8. TYPE CONVERSION FOR SCALAR ARITHMETIC ASSIGNMENT

Variable or Array Element Type	Expression Result Type				
	Integer	Real	Double-precision	Half-Precision	Complex
Integer	No conversion	Truncate fractional part	Convert to real then truncate fractional part	Convert to real then truncate fractional part	Truncate real part; discard imaginary part
Real	Convert to real	No conversion	Convert to real	Convert to real	Use real part; discard imaginary part
Double-precision	Convert to double-precision	Convert to double-precision	No conversion	Convert to double-precision	Convert real part to double-precision; discard imaginary part
Half-precision	Convert to half-precision	Convert to half-precision	Convert to half-precision	No conversion	Convert real part to half-precision; discard imaginary part
Complex	Convert to real part; use 0 for imaginary part	Use for real part; use 0 for imaginary part	Convert to real for real part; use 0 for imaginary part	Convert to real for real part; use 0 for imaginary part	No conversion

SCALAR CHARACTER ASSIGNMENT STATEMENTS

A scalar character assignment statement assigns the result of a scalar character expression to a character variable, character array element, or substring. See figure 4-11 for the format of a scalar character assignment statement.

```
var = cexp

var    A character variable, character array
       element, or substring reference; var
       must not be part of any operand in cexp.

cexp   A scalar character expression.
```

Figure 4-11. Scalar Character Assignment Statement Format

When the length of the variable or array element that appears to the left of the equals sign and the length of the expression result that appears to the right of the equals sign are the same, the scalar character assignment statement causes the value of the variable or array element to be replaced with the expression result.

When the length of the variable or array element is longer than the length of the expression result, the expression result is extended on the right with blanks so that the lengths are equal. Assignment is then performed.

When the length of the variable or array element is shorter than the length of the expression result, the expression result is truncated on the right so that the lengths are equal. Assignment is then performed.

See figure 4-12 for examples of scalar character assignment statements.

```
VOWELS = 'AEIOU'
CHARARR(I) = CHARVAR
CHARVAR(1:2) = CHARVAR(3:4)

VOWELS and CHARVAR are character variables, and
CHARARR is a character array.
```

Figure 4-12. Scalar Character Assignment Statement Examples

SCALAR LOGICAL ASSIGNMENT STATEMENTS

A scalar logical assignment statement assigns the result of a scalar logical expression to a logical variable or a logical array element. See figure 4-13 for the format of a scalar logical assignment statement.

```
var = lexp

var     A logical variable or logical array
        element

lexp    A scalar logical expression
```

Figure 4-13. Scalar Logical Assignment Statement Format

A scalar logical assignment statement causes the result of the expression that appears to the right of the equals sign to be assigned to the variable or array element that appears to the left of the equals sign.

See figure 4-14 for examples of scalar logical assignment statements.

```
LOGVAR = .TRUE.
LOGVAR = X .GT. Y
LOGVAR = X .GT. Y .AND. X .LE. 0
LOGARR(I) = L1 .OR. L2
LOGARR(I) = X .GT. Y .OR. LOGVAR


LOGVAR, L1, and L2 are logical variables, X and
Y are real variables, and LOGARR is a logical
array.
```

Figure 4-14. Scalar Logical Assignment Statement Examples

STATEMENT LABEL ASSIGNMENT STATEMENT

A statement label assignment statement assigns a statement label to an integer variable. See figure 4-15 for the format of a statement label assignment statement.

A statement label must be assigned to an integer variable if the integer variable is used in an assigned GO TO statement or as a format identifier in an input/output statement. See section 5 for a description of the assigned GO TO statement. See section 6 for a description of format identifiers in input/output statements. An integer variable that contains a statement label must not be used in any other way.

```
ASSIGN sl TO var

sl      A statement label

var     An integer variable
```

Figure 4-15. Statement Label Assignment Statement Format

An integer variable that contains a statement label can be redefined with the same statement label, with a different statement label, or with any integer value.

The ASSIGN statement that is used for statement label assignment is not related to the descriptor ASSIGN statement. The descriptor ASSIGN statement is a vector programming feature of the CYBER 200 FORTRAN language. See section 9 for a description of the descriptor ASSIGN statement.

See figure 4-16 for an example of a statement label assignment statement. The statement label assignment statement in the example causes the GO TO statement in the example to transfer control to the statement labeled 100.

```
      .
      .
      .
ASSIGN 100 TO LABEL
GO TO LABEL(100,200,300)
      .
      .
      .
```

Figure 4-16. Statement Label Assignment Statement Example

Flow control statements are executable statements that alter the normal flow of control in an executing program. Normally, statements are executed in the order of their appearance in the program, except when a condition such as an end-of-file condition or a data flag branch occurs.

The flow control statements are:

- GO TO statements
- IF statements
- DO statement
- CONTINUE statement
- PAUSE statement
- STOP statement
- CALL statement
- RETURN statement

This section describes each of the flow control statements.

GO TO STATEMENTS

The GO TO statement is an executable statement that transfers control to another executable statement. The three types of GO TO statements are the unconditional GO TO statement, the assigned GO TO statement, and the computed GO TO statement. Each type of GO TO statement is described in the following paragraphs.

UNCONDITIONAL GO TO

The unconditional GO TO statement is an executable statement that transfers control to another executable statement in the same program unit. See figure 5-1 for the format of the unconditional GO TO statement.

When an unconditional GO TO statement is executed, control transfers to the statement whose statement label is specified in the unconditional GO TO statement.

```
GO TO sl

sl      A statement label that appears in the
        label field of an executable statement
        in the same program unit that contains
        the GO TO statement
```

Figure 5-1. Unconditional GO TO Statement Format

The statement that appears after an unconditional GO TO statement should have a statement label; otherwise, the statement can never be executed.

See figure 5-2 for an example of the unconditional GO TO statement. The GO TO statement in the example transfers control to the statement labeled 200.

```
                .
                .
                .
                GO TO 200
100             I = I + 1
                .
                .
                .
200             I = I + 2
                .
                .
                .
```

Figure 5-2. Unconditional GO TO Statement Example

ASSIGNED GO TO STATEMENT

The assigned GO TO statement is an executable statement that transfers control to another executable statement in the same program unit depending on the value of an integer variable. See figure 5-3 for the format of the assigned GO TO statement.

```
GO TO var
    or
GO TO var,(sl1, ... ,sln)

var     A simple integer variable whose value is
        assigned by a statement label assignment
        statement before the GO TO statement is
        executed

sli     A statement label that appears in the
        label field of an executable statement in
        the same program unit that contains the
        GO TO statement

The comma that separates var from the statement
label list is optional.
```

Figure 5-3. Assigned GO TO Statement Format

When an assigned GO TO statement is executed, control transfers according to the following rules:

If a list of statement labels is not specified in the assigned GO TO statement, control transfers to the statement label that is contained in the integer variable.

If a list of statement labels is specified in the assigned GO TO statement, control transfers to the statement label that is contained in the integer variable; however, the statement label contained in the integer variable must appear in the list of statement labels.

The integer variable must be defined before the assigned GO TO statement is executed. The value of the integer variable must be the statement label of an executable statement in the same program unit in which the assigned GO TO statement appears. The statement label ASSIGN statement is used to assign the value to the integer variable. See section 4 for a description of this statement.

The statement that appears after an assigned GO TO statement should have a statement label; otherwise, the statement can never be executed.

See figure 5-4 for an example of the assigned GO TO statement. The first assigned GO TO statement in the example transfers control to the statement labeled 200. The second assigned GO TO statement in the example transfers control to the statement labeled 400.

```
          .
          .
          .
          ASSIGN 200 TO LABEL
          GO TO LABEL
100       I = I + 1
          .
          .
          .
200       I = I + 2
          ASSIGN 400 TO LABEL
          GO TO LABEL,(100,200,300,400,500)
300       I = I + 3
          .
          .
          .
400       I = I + 4
          .
          .
          .
```

Figure 5-4. Assigned GO TO Statement Example

COMPUTED GO TO STATEMENT

The computed GO TO statement is an executable statement that transfers control to another executable statement in the same program unit depending on the value of an integer expression. See figure 5-5 for the format of the computed GO TO statement.

When a computed GO TO statement is executed, control transfers to one of the statement labels that appears in the computed GO TO statement. The integer expression that appears in the computed GO TO selects the statement label to which control transfers. If the result of the expression is 1, control transfers to the first statement label in the list. If the result of the expression is 2, control transfers to the second statement label in the list, and so on. If the result of the expression is less than 1 or greater than the number of statement labels in the list, control transfers to the statement that follows the computed GO TO statement.

```
GO TO(sl1, ... ,sln),aexp

sli       A statement label that appears in the
          label field of an executable statement
          in the same program unit that contains
          the GO TO statement

aexp      A scalar integer expression

The comma that separates the statement label list
from aexp is optional.
```

Figure 5-5. Computed GO TO Statement Format

See figure 5-6 for an example of the computed GO TO statement. The computed GO TO statement in the example is executed four times. The first time the computed GO TO statement is executed, it transfers control to the statement labeled 200; the second time the computed GO TO statement is executed, it transfers control to the statement labeled 300; the third time the computed GO TO statement is executed, it transfers control to the statement labeled 400; the fourth time the computed GO TO statement is executed, it transfers control to the statement that follows the computed GO TO statement.

```
          .
          .
          .
          I = 0
          N = 0
100       N = N + 1
          GO TO(200,300,400),N
          I = I * 23
          .
          .
          .
200       I = I + 5
          GO TO 100
300       I = I + 10
          GO TO 100
400       I = I + 20
          GO TO 100
          .
          .
          .
```

Figure 5-6. Computed GO TO Statement Example

IF STATEMENTS

The IF statement is an executable statement that determines whether one or more statements are executed depending on a specified condition. The three types of IF statements are the arithmetic IF statement, the logical IF statement, and the block IF statement. Each type of IF statement is described in the following paragraphs.

ARITHMETIC IF STATEMENT

The arithmetic IF statement is an executable statement that transfers control to another executable statement in the same program unit depending on the result of an arithmetic expression. See figure 5-7 for the format of the arithmetic IF statement.

```
IF (aexp) sl1,sl2,sl3

aexp      A scalar arithmetic expression of any
          type except complex

sli       A statement label that appears in the
          label field of an executable statement
          in the same program unit that contains
          the IF statement
```

Figure 5-7. Arithmetic IF Statement Format

When an arithmetic IF statement is executed, the arithmetic expression is evaluated. Control transfers to one of the three statement labels specified in the arithmetic IF statement. If the result of the expression is negative, control transfers to the first statement label in the arithmetic IF statement. If the result of the expression is zero, control transfers to the second statement label in the arithmetic IF statement. If the result of the expression is positive, control transfers to the third statement label in the arithmetic IF statement.

See figure 5-8 for an example of the arithmetic IF statement. The arithmetic IF statement in the example transfers control to the statement labeled 100 if the result of A - B is negative; the arithmetic IF statement transfers control to the statement labeled 200 if the result of A - B is zero or positive.

```
                .
                .
                .
                IF(A-B) 100,200,200
100             TEMP = A
                A = B
                B = TEMP
200             C = 25.0
                .
                .
                .
```

Figure 5-8. Arithmetic IF Statement Example

LOGICAL IF STATEMENT

The logical IF statement is an executable statement that controls the execution of the statement that appears in the logical IF statement depending on the result of a logical expression. See figure 5-9 for the format of the logical IF statement.

When a logical IF statement is executed, the logical expression is evaluated. If the result of the expression is .TRUE., the statement that appears in the logical IF statement is executed; then the statement that follows the logical IF statement is executed unless the executable statement in the logical IF statement transfers control elsewhere.

If the result of the expression is .FALSE., the statement that appears in the logical IF statement is not executed; instead, the statement that follows the logical IF statement is executed.

```
IF (lexp) st

lexp      A scalar logical expression

st        Any executable statement except a DO
          statement, logical IF statement, block
          IF statement, ELSE IF statement, ELSE
          statement, END IF statement, block
          WHERE statement, OTHERWISE statement,
          END WHERE statement, or END statement
```

Figure 5-9. Logical IF Statement Format

The C64 option on the FTN200 control statement controls how comparisons are performed for logical expressions that appear in logical IF statements. If the C64 option is specified, integer comparisons for the relational operators .EQ. and .NE. are fullword comparisons; thus, all 64 bits of the integer operands are compared.

If the C64 option is not specified, integer comparisons for the relational operators .EQ. and .NE. are not fullword comparisons; only bits 16 through 63 are compared.

Because bits 0 through 15 of an integer value are always 0, the C64 option is not normally used. The C64 option is used mainly in programs that use integer variables to contain Hollerith data.

See figure 5-10 for an example of the logical IF statement. The logical IF statement in the example causes the GO TO statement to be executed if A is greater than or equal to B. The GO TO statement is not executed if A is less than B.

```
                .
                .
                .
                IF(A.GE.B) GO TO 200
                TEMP = A
                A = B
                B = TEMP
200             C = 25.0
                .
                .
                .
```

Figure 5-10. Logical IF Statement Example

BLOCK IF STATEMENT

The block IF statement is an executable statement that controls the execution of blocks of executable statements in the same program unit depending on the result of a logical expression. See figure 5-11 for the format of the block IF statement.

```
IF (lexp) THEN

lexp    A scalar logical expression
```

Figure 5-11. Block IF Statement Format

Nesting Block IF Structures and DO Loops

A nested block IF structure is a block IF structure that appears in an if-block, elseif-block, or else-block of another block IF structure. A nested block IF structure must appear entirely within an if-block, elseif-block, or else-block. Control can transfer from an if-block, elseif-block, or else-block of a nested block IF structure to the if-block, elseif-block, or else-block of the outer block IF structure in which the nested block IF structure appears. However, the converse is not true: control must not transfer from an if-block, elseif-block, or else-block of an outer block IF structure to an if-block, elseif-block, or else-block of a nested block IF structure. See figure 5-21 for a nested block IF structure.

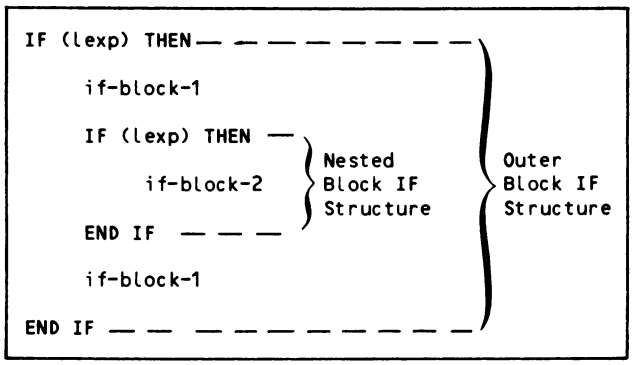


Figure 5-21. Nested Block IF Structure

A block IF structure that appears in the range of a DO statement must be entirely in the range of the DO statement. An END IF statement must not be the terminal statement of a DO loop. A DO statement can appear in an if-block, elseif-block, or else-block, but the entire range of the DO statement must appear in the if-block, elseif-block, or else-block.

DO STATEMENT

The DO statement is an executable statement that causes a group of statements to be executed repeatedly. See figure 5-22 for the format of the DO statement.

Every DO statement has a range. The range of a DO statement consists of all of the executable statements beginning with the first executable statement after the DO statement and ending with the terminal statement specified in the DO statement.

DO LOOPS

A DO loop consists of a DO statement and the range of the DO statement. See figure 5-23 for the format of a DO loop.

A DO statement must be the first statement of a DO loop. A DO loop can be entered only through the DO statement.

The terminal statement of a DO loop can be any statement except the following:

RETURN statement

DO sl cvar = $aexp_1$,$aexp_2$,$aexp_3$

sl	The statement label that appears in the label field of the last statement in the DO loop.
cvar	A variable of any arithmetic type except complex; cvar is used as the control variable for the DO loop. The control variable cvar must not also be the control variable of an outer DO loop.
$aexp_1$	A scalar arithmetic expression of any type except complex; $aexp_1$ is used as the initial value for the control variable.
$aexp_2$	A scalar arithmetic expression of any type except complex; $aexp_2$ is used as the terminal value for the control variable.
$aexp_3$	A scalar arithmetic expression of any type except complex; optional; $aexp_3$ is used as the incrementation value for the control variable. If $aexp_3$ is not specified, the incrementation value for the control variable is 1. The result of $aexp_3$ must not be zero.

Figure 5-22. DO Statement Format

Figure 5-23. DO Loop Format

STOP statement

END statement

Unconditional GO TO statement

Assigned GO TO statement

Special call statement (unless the label is the target of a GO TO statement)

DO statement

Arithmetic IF statement

Block IF statement

ELSE IF statement

ELSE statement

END IF statement

Block WHERE statement

OTHERWISE statement

For compatibility with CYBER 200 FORTRAN, a DO statement can also have an extended range. The extended range of a DO loop consists of the statements executed when control transfers out of and then returns to the range of the DO loop. 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. The extended range of the DO loop must not contain a DO loop that has its own extended range.

When a DO statement is executed, the following operations are performed:

1. The expressions in the DO statement are evaluated. (If an expression uses the control variable, it must be defined before the DO statement.)

2. The control variable is initialized with the initial value specified in the DO statement.

3. If the incrementation value is positive, the value of the control variable is compared to the terminal value specified in the DO statement. If the value of the control variable is less than or equal to the terminal value, the range of the DO statement is executed. If the value of the control variable is greater than the terminal value, control transfers to the statement that follows the terminal statement.

   If the incrementation value is negative, the value of the control variable is compared to the terminal value specified in the DO statement. If the value of the control variable is greater than or equal to the terminal value, the range of the DO statement is executed. If the value of the control variable is less than the terminal value, control transfers to the statement that follows the terminal statement.

   Thus, a DO loop can be executed zero times.

4. The value of the control variable is incremented by the amount specified in the DO statement; then step 3 is repeated.

The control variable of a DO statement must not be redefined in the range of the DO statement. However, variables that specify the initial value, terminal value, and incrementation value for the control variable can be redefined in the range of the DO statement. Redefining those variables has no effect on the execution of the DO loop.

The compiler might generate more efficient object code for a DO loop if you specify the DO=1 option on the FTN200 control statement.

The compiler can generate vector machine instructions for some DO loops if you specify the V compilation option on the FTN200 control statement. See section 9 for a description of loop vectorization.

See figure 5-24 for an example of a DO loop. The range of the DO statement in the example consists of all of the statements shown in the figure except the DO statement. The DO loop in the example is executed five times. The values that are assigned to the control variable I are 1, 3, 5, 7, and 9.

```
                .
                .
                .
                DO 100 I = 1,10,2
                IF(A(I).LT.B(I)) THEN
                TEMP = A(I)
                A(I) = B(I)
                B(I) = TEMP
                END IF
          100   CONTINUE
                .
                .
                .
```

Figure 5-24. DO Loop Example

NESTING DO LOOPS AND BLOCK IF STRUCTURES

A nested DO loop is a DO loop that appears within another DO loop or within an if-block, elseif-block, or else-block. DO loops can be nested to any level. A nested DO loop must be entirely within the outer DO loop. When DO loops are nested, each DO loop must have a unique control variable.

The terminal statement of a nested DO loop can be the same as the terminal statement of an outer DO loop or can appear before the terminal statement of the outer DO loop. If more than one DO loop has the same terminal statement, control can transfer to the terminal statement only from within the range of the innermost DO statement.

A DO loop that is nested in an if-block, elseif-block, or else-block must be entirely within the if-block, elseif-block, or else-block. A block IF structure that is nested in a DO loop must be entirely within the DO loop.

See figures 5-24 and 5-25 for examples of nested DO loops and block IF structures.

```
                .
                .
                .
                DO 100 I = 1,10
                DO 100 J = 1,5
                A(I,J) = 0.0
          100   CONTINUE
                .
                .
                .
                IF(A.LT.B) THEN
                DO 200 I = 1,10
                C(I) = 100.0
          200   CONTINUE
                END IF
                .
                .
                .
```

Figure 5-25. Nested DO Loops Example

CONTINUE STATEMENT

The CONTINUE statement is an executable statement that performs nothing. See figure 5-26 for the format of the CONTINUE statement.

```
CONTINUE
```

Figure 5-26. CONTINUE Statement Format

When a CONTINUE statement is executed, no operation is performed. The flow of control is not interrupted. The CONTINUE statement is used to carry a statement label. For example, a CONTINUE statement can be used as the terminal statement of a DO loop when a statement such as an unconditional GO TO or arithmetic IF statement would otherwise be the terminal statement.

See figure 5-27 for an example of the CONTINUE statement.

```
          .
          .
          .
          DO 100 I = 1,10
          .
          .
          .
          IF(A.LT.B) A = B
100       CONTINUE
          .
          .
          .
```

Figure 5-27. CONTINUE Statement Example

PAUSE STATEMENT

The PAUSE statement is an executable statement that temporarily halts execution of the program. See figure 5-28 for the format of the PAUSE statement.

```
PAUSE disp

disp    One to five decimal digits or a
        character constant; optional
```

Figure 5-28. PAUSE Statement Format

When a PAUSE statement is executed, the string of decimal digits or the character constant is displayed in the job dayfile or at your terminal. If no string is specified in the PAUSE statement, the character string PAUSE is displayed in the job dayfile or at your terminal. The PAUSE statement then halts program execution until a response is received.

If the job is executing in batch mode, the operator must enter a carriage return from the console in order to resume execution of the program. If the job is executing interactively, you must enter a carriage return from your terminal in order to resume execution of the program. Program execution resumes with the next executable statement after the PAUSE statement.

See figure 5-29 for an example of the PAUSE statement.

```
          .
          .
          .
          I = I + J + K
          PAUSE 'HI THERE'
          L = I + 2
          .
          .
          .
```

Figure 5-29. PAUSE Statement Example

STOP STATEMENT

The STOP statement is an executable statement that permanently halts execution of the program. See figure 5-30 for the format of the STOP statement.

```
STOP disp

disp    One to five decimal digits or a
        character constant; optional
```

Figure 5-30. STOP Statement Format

When a STOP statement is executed, the string of decimal digits or the character constant, if specified, is displayed in the job dayfile or at your terminal. The string is also written on the output file of the program. If no string is specified in the STOP statement, the character string STOP is displayed in the job dayfile or at your terminal. The string is also written on the output file of the program. The STOP statement then halts program execution and returns control to the operating system.

See figure 5-31 for an example of the STOP statement.

```
          .
          .
          .
          IF(A.EQ.0.0) THEN
          STOP 'A IS 0'
          ELSE
          C = B/A
          END IF
          .
          .
          .
```

Figure 5-31. STOP Statement Example

CALL STATEMENT

The CALL statement is an executable statement that transfers control to a subroutine subprogram or to a predefined subroutine. See section 7 for a description of the CALL statement.

RETURN STATEMENT

The RETURN statement is an executable statement that returns control from a subroutine or function subprogram to the program unit that called the subroutine or function subprogram. See section 7 for a description of the RETURN statement.

INPUT/OUTPUT STATEMENTS 6

Input/output statements transfer data between files and internal storage. Some input/output statements manipulate files, and some input/output statements inquire about the properties of files.

The types of input/output statements are:

- Sequential access formatted input/output statements
- Direct access formatted input/output statements
- Sequential access unformatted input/output statements
- Direct access unformatted input/output statements
- List-directed input/output statements
- Namelist input/output statements
- Buffer input/output statements
- Internal file input/output statements
- Extended internal file input/output statements
- Concurrent input/output statements
- Direct calls to System Interface Language (SIL) routines
- Auxiliary input/output statements
- File positioning statements

This section describes records, files, and the input/output statements. The buffer input/output statements are provided for compatibility with other FORTRAN compilers; see appendix E for a description of the buffer input/output statements. The concurrent input/output statements are written as calls to predefined subroutines; see section 11 for a description of the concurrent input/output subroutines. See section 13 for a description of direct calls to SIL routines.

All types of FORTRAN 200 I/O statements result in calls to SIL subroutines that perform the physical I/O. Each FORTRAN I/O statement specifies one or more records of data to be read or written. When writing data, the SIL subroutine inserts the data into the record structure of the file; when reading data it extracts the data from the record structure.

The SIL record structure is transparent to FORTRAN 200 I/O processing. A FORTRAN 200 program cannot specify the SIL record structure. If the file already exists, the record structure already defined for the file is used; if the FORTRAN program creates a sequential access file, the control word delimited (W) record type is used; if the FORTRAN program creates a direct access file, the fixed-length (F) record type is used.

RECORDS

Input/output statements transfer records of data between files and internal storage. A record is a sequence of values or a sequence of characters. For example, a punched card is a record. A record need not always correspond to a physical entity, however. The three types of records are formatted records, unformatted records, and endfile records. Each type of record is described in the following paragraphs.

FORMATTED RECORDS

A formatted record consists of a sequence of character values that can be read and written only by sequential access formatted input/output statements, direct access formatted input/output statements, list-directed input/output statements, namelist input/output statements, internal file input/output statements, extended internal file input/output statements, concurrent input/output statements, and direct calls to SIL routines. The length (in characters) of a formatted record cannot be less than zero or greater than $2^{24}-1$.

UNFORMATTED RECORDS

An unformatted record consists of a sequence of values that can be read and written only by sequential access unformatted input/output statements, direct access unformatted input/output statements, and buffer input/output statements. The length of an unformatted record cannot be less than zero bytes; unlike formatted records, it can be greater than $2^{24}-1$ characters.

ENDFILE RECORDS

An endfile record consists of an end-of-file mark that can be written only by an ENDFILE statement. An endfile record must occur only as the last record of a file. An endfile record does not have a length property.

FORTRAN 200 writes a SIL end-of-group delimiter as the endfile record.

FILES

A file is a sequence of records. A file exists if it is a local file, an attached permanent file, an attached pool file, or a public file. See the operating system reference manual for a description of these types of files. All input/output statements can refer to files that exist. The INQUIRE statement, OPEN statement, CLOSE statement, ENDFILE statement, and all WRITE, PRINT, and PUNCH statements can also refer to files that do not exist.

In order to reference a file in a FORTRAN program, the file must be connected to a unit. A unit is a path between a FORTRAN program and a file. A unit is identified by an integer constant from 0 to 999 or by an H type Hollerith constant. A file cannot be referenced in an input/output statement unless it is connected to a unit; a file that is not connected to a unit can be referenced in an OPEN statement, CLOSE statement, or INQUIRE statement, however.

A file can be connected to a unit by using the OPEN statement; a file can be disconnected from a unit by using the CLOSE statement. A file can be explicitly preconnected to a unit by using the PROGRAM statement. However, in the absence of PROGRAM statement specifications, a unit is implicitly preconnected to a file whose name is determined by the unit identifier. See the descriptions of these statements for more information about how to connect a file to a unit.

If an input/output statement references a unit that has not been explicitly connected to a file, the unit is implicitly preconnected to a file whose name is derived from the unit identifier. See the description of the UNIT specifier for more information about processor-determined connection of units and files.

A file can be connected to more than one unit at the same time, but more than one file cannot be connected to the same unit at the same time. Also, the unit to which a file is connected can be changed during program execution.

Each file has an initial point and a terminal point. The initial point of a file is the position before the first record of the file. The terminal point is the position after the last record of the file.

If a file is positioned within a record, that record is the current record. The record that appears before the current record is the preceding record, and the record that appears after the current record is the next record.

If a file is positioned between two records, the record that appears before the file position is the preceding record, and the record that appears after the file position is the next record.

The three types of files are external files, internal files, and extended internal files. Each type of file is described in the following paragraphs.

EXTERNAL FILES

An external file is a sequence of records that is contained on an external device, such as a disk. Each external file is identified by a file name. A file name is a string of one to eight letters or digits. The first character of the file name must be a letter; however, files created by the operating system can have file names that begin with a number. You must not use file names which begin with any of the following characters: Q5, Q6, Q7, Q8, or Q9.

Input/output statements affect the position of an external file. The position of an external file is the position of the file after execution of the last input/output statement that referenced the file. If an input/output statement did not previously reference the file, the position of the file is the initial point of the file.

You can access external files sequentially or directly. The method by which you access a file is determined when the file is connected to a unit.

A sequential access external file has the following properties:

The order of the records is the order in which the records are written. A record that is beyond the last record written must not be read.

The records of the file can be either all formatted records or all unformatted records. The last record of the file can be an endfile record.

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

A direct access external file has the following properties:

The order of the records is the order of their record numbers. The records can be read or written in any order. A record that has not been written since the file was created must not be read.

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

The records of the file can be read and written only by direct access input/output statements.

All records of the file must have the same length.

Each record of the file is identified by a positive record number. The record number is specified when the record is written and can never be changed. A record cannot be deleted, but a record can be rewritten.

INTERNAL FILES

An internal file is a sequence of records contained in memory. An internal file cannot be identified by a file name.

An internal file is always positioned at the initial point prior to execution of a data transfer input/output statement.

You can access internal files only through sequential access formatted input/output statements. The method by which you access a file is specified when the file is connected to a unit.

A sequential access internal file has the following properties:

An internal file is a character variable, character array element, character array, or substring.

If an internal file is a character variable, character array element, or a substring, the internal file consists of a single record whose length is the same as the length of the character variable, character array element, or substring. If an internal file is a character array, the internal file is a sequence of character array elements; each element is a record of the file. The order of the records in the file is the same as the order of the elements in the character array. Every record of the file has the same length, which is the length of the elements of the character array.

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

A record can be read only if the character variable, character array element, or substring that is the record has been defined.

A character variable, character array element, or substring that is a record of an internal file can be defined without using an output statement. For example, the character variable, character array element, or substring could be defined by using a scalar character assignment statement.

An internal file can be read or written only by sequential access formatted input/output statements that do not specify list-directed or namelist formatting.

Internal files must not be referenced by auxiliary input/output statements.

The implied value of the BLANK specifier is NULL for internal files. The BLANK specifier is described later in this section.

EXTENDED INTERNAL FILES

An extended internal file is a sequence of records that is contained in memory. An extended internal file cannot be identified by a file name.

An extended internal file is always positioned at the initial point prior to execution of a data transfer input/output statement.

An extended internal file has the following properties:

An extended internal file is a noncharacter variable, noncharacter array element, or noncharacter array.

A record of an extended internal file is a number of consecutive bytes. The length (number of bytes) of a record of an extended internal file is specified in the ENCODE/DECODE statement. Every record of an extended internal file has the same length. The first byte of the first record is the first byte of the variable, array, or array element. The first byte of the second record is the byte immediately following the last byte of the first record, and so forth.

The variable, array element, or array that is the record of the extended internal file is defined by writing the record (with an ENCODE statement). If the number of characters encoded in a record is less than the length of the record, the remaining portion of the record is filled with blanks.

A record can be read (decoded) only if that portion of the variable, array element, or array that is the extended internal file has been defined.

A record of an extended internal file can be defined without using an ENCODE statement. For example, the variable, array element, or array could be defined by using an assignment statement.

An extended internal file can be read or written only by ENCODE and DECODE statements.

Extended internal files must not be referenced by auxiliary input/output statements.

The implied value of the BLANK specifier is NULL for extended internal files. The BLANK specifier is described later in this section.

An extended internal file is assumed to have enough records to contain all the data written with an ENCODE statement or read with a DECODE statement.

INPUT/OUTPUT STATEMENT COMPONENTS

An input/output statement consists of a FORTRAN keyword, an optional control information list, and an optional input/output list. The keyword specifies the kind of input/output operation that is to be performed.

The control information list controls execution of the input/output statement. The input/output list specifies the variables and arrays that are used for the input/output operation. The control information list and the input/output list are described in the following paragraphs.

CONTROL INFORMATION LIST

A control information list is a set of specifiers that control the way in which an input/output operation is performed. The specifiers that appear in a control information list must be separated by commas. The specifiers that can appear in a control information list are summarized in table 6-1. The table shows the purpose of each specifier. Each of the specifiers is described in the following paragraphs.

ACCESS Specifier

The ACCESS specifier indicates the access method for a particular file. The access method can be sequential access or direct access. See figure 6-1 for the format of the ACCESS specifier.

```
ACCESS = cexp

cexp    A scalar character expression that
        specifies an access method. The
        values that can be specified are:

        SEQUENTIAL    Indicates sequential
                      access

        DIRECT        Indicates direct access

        If the ACCESS specifier is omitted,
        ACCESS='SEQUENTIAL' is used.

        If the ACCESS specifier appears in an
        INQUIRE statement, cexp must be a
        character variable, character array
        element, or substring, and the value
        of cexp is returned by the INQUIRE
        statement.
```

Figure 6-1. ACCESS Specifier Format

TABLE 6-1. CONTROL INFORMATION LIST SPECIFIERS

Specifier	Purpose
ACCESS	Indicates the access method for a particular file
BLANK	Indicates how blanks in numeric input fields are interpreted by formatted input statements
BUFS	Indicates the buffer length in 512 word blocks for a particular unit
DIRECT	Indicates if a particular file is a direct access file
END	Indicates the statement to which control transfers when an end-of-file condition occurs
ERR	Indicates the statement to which control transfers when an input/output error occurs
EXIST	Indicates if a particular file exists or if a particular unit exists
FILE	Indicates the name of the file that is to be connected or for which the INQUIRE statement is to return specifier values
FMT	Indicates the format specification for a formatted input/output operation
FORM	Indicates whether a particular file is connected for formatted input/output or for unformatted input/output
FORMATTED	Indicates if formatted input/output can be performed on a particular file
IOSTAT	Indicates if an input/output error condition exists
NAME	Indicates the name of the file referenced in an INQUIRE statement
NAMED	Indicates if a particular file has a name
NEXTREC	Indicates the next record that would be read or written by a direct access input/output statement
NUMBER	Indicates the identifier of the unit to which a particular file is connected
OPENED	Indicates if a particular file is connected to a unit or if a particular unit is connected to a file
REC	Indicates the number of the record to be read or written by a direct access input/output statement
RECL	Indicates the length of each record of a direct access file
SEQUENTIAL	Indicates if a particular file is a sequential access file
STATUS	Indicates the status and disposition of a file connected to a particular unit
UNFORMATTED	Indicates if unformatted input/output can be performed on a particular file
UNIT	Indicates the unit on which an input/output statement functions

An ACCESS specifier that appears in an OPEN statement for a new file establishes the access method for the file. An ACCESS specifier that appears in an OPEN statement for an existing file must specify an access method permitted by the operating system.

If direct access is specified, the record length must also be specified with the RECL specifier.

BLANK Specifier

The BLANK specifier indicates how blanks in numeric input fields are interpreted by formatted input statements. See figure 6-2 for the format of the BLANK specifier.

You can use the BLANK specifier only with files that contain formatted records.

BUFS Specifier

The BUFS specifier indicates the buffer size in 512 word blocks for a particular file. See figure 6-3 for the format of the BUFS specifier.

The BUFS specifier must not change the buffer length of a file already connected to a unit.

DIRECT Specifier

The DIRECT specifier indicates if a particular file is a direct access file. See figure 6-4 for the format of the DIRECT specifier.

```
BLANK = cexp

cexp    A scalar character expression that
        specifies how blanks in numeric input
        fields are interpreted by formatted
        input statements.  The values that can
        be specified are:

        NULL    Indicates that blanks in
                numeric input fields are
                ignored by the formatted input
                statements

        ZERO    Indicates that blanks in
                numeric input fields are
                interpreted as zeros by
                formatted input statements

        If the BLANK specifier is omitted,
        BLANK='NULL' is used.

        If the BLANK specifier appears in an
        INQUIRE statement, cexp must be a
        character variable, character array
        element, or substring, and the value
        of cexp is returned by the INQUIRE
        statement.
```

Figure 6-2. BLANK Specifier Format

```
BUFS = aexp

aexp    A scalar integer expression that
        specifies the size in 512 word blocks
        of the buffer for a unit.  The value
        of aexp must be no less than 1 and no
        greater than 24.

        If the BUFS specifier is omitted,
        BUFS=8 is used.

        If the BUFS specifier appears in an
        INQUIRE statement, aexp must be an
        integer variable or an integer array
        element, and the value of aexp is
        returned by the INQUIRE statement.
```

Figure 6-3. BUFS Specifier Format

END Specifier

The END specifier indicates the statement to which control transfers when an end-of-file condition occurs. See figure 6-5 for the format of the END specifier.

If an end-of-file condition occurs, the following steps are taken:

1. Execution of the input statement terminates.
2. If the input statement contains an IOSTAT specifier, the input statement assigns a negative value to the integer variable or integer array element specified in the IOSTAT specifier.
3. Control transfers to the statement label that is specified in the END specifier.

```
DIRECT = cvar

cvar    A character variable, character array
        element, or substring; the value of
        cvar is returned by the INQUIRE
        statement in which the DIRECT specifier
        appears.  The values that can be
        returned are:

        YES         Indicates that a particular
                    file is a direct access file

        NO          Indicates that a particular
                    file is not a direct access
                    file

        UNKNOWN     Indicates that the access
                    method of a particular file
                    is not known
```

Figure 6-4. DIRECT Specifier Format

```
END = sl

sl      The statement label that appears in the
        label field of the statement to which
        control transfers when an end-of-file
        condition occurs during an input
        operation.  The statement to which
        control transfers must be in the same
        program unit as the input statement that
        contains the END specifier.

        If the END specifier is omitted, an
        execution-time error occurs when an
        end-of-file condition occurs during an
        input operation.
```

Figure 6-5. END Specifier Format

ERR Specifier

The ERR specifier indicates the statement to which control transfers when an input/output error occurs. See figure 6-6 for the format of the ERR specifier.

```
ERR = sl

sl      The statement label that appears in the
        label field of the statement to which
        control transfers when an error occurs
        during an input/output operation.  The
        statement to which control transfers must
        be in the same program unit as the
        input/output statement that contains the
        ERR specifier.

        If the ERR specifier is omitted, an error
        message is issued.
```

Figure 6-6. ERR Specifier Format

If an input/output error occurs, the following steps are taken:

1. Execution of the input/output statement terminates.

2. The position of the file specified in the input/output statement becomes undefined.

3. If the input/output statement contains an IOSTAT specifier, the input/output statement assigns the number of the execution-time error that occurred to the integer variable or integer array element specified in the IOSTAT specifier. See appendix B for the numbers and descriptions of the execution-time errors.

4. Control transfers to the statement label that is specified in the ERR specifier.

EXIST Specifier

The EXIST specifier indicates if a particular file exists or if a particular unit exists. See figure 6-7 for the format of the EXIST specifier.

```
EXIST = lvar

lvar     A logical variable or logical array
         element; the value of lvar is returned
         by the INQUIRE statement in which the
         EXIST specifier appears. The values
         that can be returned are:

         .TRUE.     Indicates that a particular
                    file or unit exists

         .FALSE.    Indicates that a particular
                    file or unit does not exist
```

Figure 6-7. EXIST Specifier Format

FILE Specifier

The FILE specifier indicates the name of the file that is to be connected or for which the INQUIRE statement is to return specifier values. See figure 6-8 for the format of the FILE specifier.

```
FILE = cexp

cexp     A scalar character expression that
         specifies the name of a file. A FILE
         specifier that appears in an OPEN
         statement specifies the name of the
         file to be connected to a unit. If
         the file name specified for cexp does
         not exist, a new file of that name is
         created.

         If the FILE specifier is omitted from
         an OPEN statement and if the unit
         specified in the OPEN statement is not
         connected to a file, the unit is
         connected to a file as follows:

         • If STATUS='SCRATCH' is specified
           in the OPEN statement, the unit is
           connected to a
           processor-determined file.

         • If the STATUS specifier is not
           SCRATCH, and if the unit number
           specified is no less than 0 and no
           greater than 999, the unit is
           connected to the file TAPEunum,
           where unum is the unit number.

         • If the STATUS specifier is not
           SCRATCH, and if the unit number
           specified is of the form nHf,
           where f is a valid system file
           name, the unit is connected to the
           file f.

         • In all other cases, the unit is
           not connected to a file.

         A FILE specifier that appears in an
         INQUIRE statement specifies the name
         of the file for which the INQUIRE
         statement returns values of other
         specifiers. The file name specified
         for cexp need not exist.

         If the FILE specifier is omitted from
         an INQUIRE statement, the UNIT
         specifier must appear in the INQUIRE
         statement.
```

Figure 6-8. FILE Specifier Format

FMT Specifier

The FMT specifier indicates the format specification to be used for a formatted, list-directed, or namelist input/output operation. See figure 6-9 for the format of the FMT specifier.

FORM Specifier

The FORM specifier indicates whether a particular file is connected for formatted input/output or for unformatted input/output. See figure 6-10 for the format of the FORM specifier.

A FORM specifier that appears in an OPEN statement for a new file establishes the type of input/output that can be performed on the file. A FORM specifier that appears in an OPEN statement for an existing file must specify the type of input/output that was established for the file when the file was created.

FORMATTED Specifier

The FORMATTED specifier indicates if formatted input/output can be performed on a particular file. See figure 6-11 for the format of the FORMATTED specifier.

FMT = fid

fid A format identifier; fid can be any of the following:

- The statement label that appears in the label field of a FORMAT statement. The FORMAT statement must appear in the same program unit as the input/output statement that contains the FMT specifier.
- An integer variable that has been assigned the statement label of a FORMAT statement by a statement label assignment statement. The FORMAT statement must appear in the same program unit as the input/output statement that contains the FMT specifier.
- A scalar character expression whose result is a format specification. Format specification is described later in this section. An expression that involves the concatentation of an operand whose length specification is unknown (specified by an asterisk in the CHARACTER statement) is not permitted unless the operand is a symbolic constant.
- A character or noncharacter array, character or noncharacter array element, or substring that contains a format specification. Format specification is described later in this section.
- A namelist group name.
- An asterisk, which indicates that the input/output statement is a list-directed input/output statement.

The characters FMT= can be omitted, but if they are omitted, fid must be the second item in the control information list. The first item in the control information list must be the unit specifier without the characters UNIT=.

Figure 6-9. FMT Specifier Format

FORM = cexp

cexp A scalar character expression that specifies the type of input/output that can be performed on a file. The values that can be specified are:

FORMATTED	Indicates that a particular file is being connected for formatted input/output
UNFORMATTED	Indicates that a particular file is being connected for unformatted input/output

If the FORM specifier is omitted, FORM='UNFORMATTED' is used if the file is being connected for direct access input/output; FORM='FORMATTED' is used if the file is being connected for sequential access input/output.

If the FORM specifier appears in an INQUIRE statement, cexp must be a character variable, character array element, or substring, and the value of cexp is returned by the INQUIRE statement. If the type of input/output that can be performed on a file is not known, the value UNKNOWN is returned by the INQUIRE statement.

Figure 6-10. FORM Specifier Format

FORMATTED = cvar

cvar A character variable, character array element, or substring; the value of cvar is returned by the INQUIRE statement in which the FORMATTED specifier appears. The values that can be returned are:

YES	Indicates that formatted input/output can be performed on a particular file
NO	Indicates that formatted input/output cannot be performed on a particular file
UNKNOWN	Indicates that the type of input/output that can be performed on a particular file is not known

Figure 6-11. FORMATTED Specifier Format

IOSTAT Specifier

The IOSTAT specifier indicates if an input/output error condition exists. See figure 6-12 for the format of the IOSTAT specifier.

```
IOSTAT = avar

avar    An integer variable or an integer
        array element; the value of avar is
        returned by the statement in which the
        IOSTAT specifier appears. The values
        that can be returned are:

        -1      Indicates that an end-of-file
                condition exists, but no error
                condition exists

         0      Indicates that neither an
                end-of-file condition nor an
                error condition exists

        >0      Indicates that an error
                condition exists. The value
                is the number of the execution-
                time error message. See ap-
                pendix B for the numbers and
                descriptions of the execution-
                time errors.
```

Figure 6-12. IOSTAT Specifier Format

NAME Specifier

The NAME specifier indicates the name of the file referenced in an INQUIRE statement. See figure 6-13 for the format of the NAME specifier.

```
NAME = cvar

cvar    A character variable, character array
        element, or substring; the value of
        cvar is returned by the INQUIRE
        statement in which the NAME specifier
        appears. The value that is returned is
        the name of the file referenced in the
        INQUIRE statement. If the file has no
        name, cvar is undefined.

        If the NAME specifier appears in an
        INQUIRE statement that also contains a
        FILE specifier, the value returned for
        cvar is not necessarily the same as the
        file specified in the FILE specifier.
        For example, the value returned for
        cvar could be a file name qualified by
        a user identification.

        The value returned for cvar is always
        suitable for use in a FILE specifier
        that appears in an OPEN statement.
```

Figure 6-13. NAME Specifier Format

NAMED Specifier

The NAMED specifier indicates if a particular file has a name. See figure 6-14 for the format of the NAMED specifier.

```
NAMED = lvar

lvar    A logical variable or a logical array
        element; the value of lvar is returned
        by the INQUIRE statement in which the
        NAMED specifier appears. The values
        that can be returned are:

        .TRUE.      Indicates that a particular
                    file has a name

        .FALSE.     Indicates that a particular
                    file does not have a name.
```

Figure 6-14. NAMED Specifier Format

NEXTREC Specifier

The NEXTREC specifier indicates the next record that would be read or written by a direct access input/output statement. See figure 6-15 for the format of the NEXTREC specifier.

```
NEXTREC = avar

avar    An integer variable or an integer array
        element; the value of avar is returned
        by the INQUIRE statement in which the
        NEXTREC specifier appears. The value
        that is returned is the record number
        of the next record of a particular file
        to be read or written by a direct
        access input/output statement. If n is
        the record number of the most recent
        record read or written on a file, n+1
        is the record number that is returned
        for avar. If no records were read or
        written previously, the value 1 is
        returned for avar.

        If the file is not connected for direct
        access input/output, or if the position
        of the file is unknown because of a
        previous input/output error, avar is
        undefined.
```

Figure 6-15. NEXTREC Specifier Format

NUMBER Specifier

The NUMBER specifier indicates the unit identifier of the unit to which a particular file is connected. See figure 6-16 for the format of the NUMBER specifier.

```
NUMBER = avar

avar    An integer variable or an integer array
        element; the value of avar is returned
        by the INQUIRE statement in which the
        NUMBER specifier appears. The value
        that is returned is the identifier of
        the unit to which the file referenced
        in the INQUIRE statement is connected.

        If the unit identifier is an integer
        that is no less than 0 and no greater
        than 999, the integer value is returned
        for avar.

        If the unit identifier is of the form
        nHTAPEk, where k is an integer that is
        no less than 0 and no greater than 999,
        the integer value k is returned for
        avar.

        If the unit identifier is of the form
        nHf, where f is a string of one through
        eight characters, the value nHf is
        returned for avar.

        If the file is not connected to a unit,
        avar is undefined.
```

Figure 6-16. NUMBER Specifier Format

OPENED Specifier

The OPENED specifier indicates if a particular file is connected to a unit, or if a particular unit is connected to a file. See figure 6-17 for the format of the OPENED specifier.

```
OPENED = lvar

lvar    A logical variable or a logical array
        element; the value of lvar is returned
        by the INQUIRE statement in which the
        OPENED specifier appears. The values
        that can be returned are:

        .TRUE.    Indicates that a particular
                  file is connected to a unit,
                  or that a particular unit is
                  connected to a file

        .FALSE.   Indicates that a particular
                  file is not connected to a
                  unit, or that a particular
                  unit is not connected to a
                  file
```

Figure 6-17. OPENED Specifier Format

REC Specifier

The REC specifier indicates the number of the record to be read or written by a direct access input/output statement. See figure 6-18 for the format of the REC specifier.

```
REC = aexp

aexp    A positive scalar integer expression
        that specifies the number of the record
        to be read or written in a file
        connected for direct access input/
        output.
```

Figure 6-18. REC Specifier Format

RECL Specifier

The RECL specifier indicates the length of each record of a file. See figure 6-19 for the format of the RECL specifier.

```
RECL = aexp

aexp    A positive scalar integer expression
        that specifies the length of each of
        the records of a direct access file.
        The length is measured in bytes for
        both formatted and unformatted
        records. A byte is 8 bits. A
        character is represented as 1 byte.

        If the RECL specifier appears in an
        INQUIRE statement, aexp must be an
        integer variable or an integer array
        element, and the value of aexp is
        returned by the INQUIRE statement.
```

Figure 6-19. RECL Specifier Format

A RECL specifier that appears in an OPEN statement for a new file establishes the record length for the records of the file. A RECL specifier that appears in an OPEN statement for an existing file must specify the record length that was established for the records of the file when the file was created.

You must use the RECL specifier for files that are being connected for direct access input/output.

SEQUENTIAL Specifier

The SEQUENTIAL specifier indicates if a particular file is a sequential access file. See figure 6-20 for the format of the SEQUENTIAL specifier.

```
SEQUENTIAL = cvar

cvar    A character variable, character array
        element, or substring; the value of
        cvar is returned by the INQUIRE
        statement in which the SEQUENTIAL
        specifier appears.  The values that can
        be returned are:

        YES         Indicates that a particular
                    file is a sequential access
                    file

        NO          Indicates that a particular
                    file is not a sequential
                    access file

        UNKNOWN     Indicates that the access
                    method of a particular file
                    is not known.
```

Figure 6-20. SEQUENTIAL Specifier Format

STATUS Specifier

The STATUS specifier indicates the status and disposition of a file connected to a particular unit. See figure 6-21 for the format of the STATUS specifier.

UNFORMATTED Specifier

The UNFORMATTED specifier indicates if unformatted input/output can be performed on a particular file. See figure 6-22 for the format of the UNFORMATTED specifier.

UNIT Specifier

The UNIT specifier indicates the unit on which an input/output statement functions. See figure 6-23 for the format of the UNIT specifier.

A unit that appears in a UNIT specifier must be connected unless the UNIT specifier appears in an OPEN, CLOSE, or INQUIRE statement. All units whose identifiers are integers in the range 0 through 999 are implicitly preconnected to files whose names are of the form TAPEn, where n is the unit identifier. All units whose identifiers are Hollerith values of the form nHf, where f is a valid file name, are implicitly preconnected to the named files.

A unit can be explicitly preconnected by using the PROGRAM statement or execution control statement. A unit can be connected during program execution by using the OPEN statement. See the descriptions of these statements for more information about unit connection.

```
STATUS = cexp

cexp    A scalar character expression that
        specifies the status and disposition of
        a file when the file is connected and
        disconnected.  If the STATUS specifier
        appears in an OPEN statement, the
        values that can be specified for cexp
        are:

        OLD         Indicates that the file
                    already exists.  If OLD is
                    specified, a FILE specifier
                    must also appear in the OPEN
                    statement.

        NEW         Indicates that the file does
                    not already exist.  If NEW
                    is specified, a FILE
                    specifier must also appear
                    in the OPEN statement.

        SCRATCH     Indicates that the file is
                    to be connected to the unit
                    during program execution,
                    and deleted when the file is
                    disconnected.  If SCRATCH is
                    specified, the file must not
                    have a name.

        UNKNOWN     Indicates that the status of
                    the file is
                    processor-dependent.

        If the STATUS specifier is omitted from
        an OPEN statement, STATUS='UNKNOWN' is
        used.

        If the STATUS specifier appears in a
        CLOSE statement, the values that can be
        specified for cexp are:

        KEEP        Indicates that the file is
                    to exist after execution of
                    the CLOSE statement if it
                    exists before execution of
                    the CLOSE statement.  KEEP
                    must not be specified for a
                    file whose status is SCRATCH.

        DELETE      Indicates that the file is
                    not to exist after execution
                    of the CLOSE statement.

        If the STATUS specifier is omitted from
        a CLOSE statement, STATUS='KEEP' is
        used unless the status specified in the
        OPEN statement was SCRATCH, in which
        case STATUS='DELETE' is used.
```

Figure 6-21. STATUS Specifier Format

```
UNFORMATTED = cvar

cvar    A character variable, character array
        element, or substring; the value of
        cvar is returned by the INQUIRE
        statement in which the UNFORMATTED
        specifier appears.  The values that can
        be returned are:

        YES         Indicates that unformatted
                    input/output can be
                    performed on a particular
                    file

        NO          Indicates that unformatted
                    input/output cannot be
                    performed on a particular
                    file

        UNKNOWN     Indicates that the type of
                    input/output that can be
                    performed on a particular
                    file is not known
```

Figure 6-22. UNFORMATTED Specifier Format

```
UNIT = unum

unum    A unit identifier; unum can be any of
        the following:

        • A scalar integer expression whose
          result is no less than 0 and no
          greater than 999.

        • A scalar integer expression whose
          result is of the form nHf, where f
          is a string of one through eight
          characters.

        • A character variable, character
          array element, or substring that
          identifies an internal file.

        • An asterisk, which indicates that
          the unit is a processor-determined
          unit that is preconnected for
          formatted sequential input/output.
          See the description of the PROGRAM
          statement.

        The characters UNIT= can be omitted,
        but if they are omitted, unum must be
        the first item in the control
        information list.
```

Figure 6-23. UNIT Specifier Format

If a unit that appears in a UNIT specifier is not connected to a file, and if that UNIT specifier is used in an input/output statement other than an OPEN, CLOSE, or INQUIRE statement, the processor connects that unit to a file as follows:

If the unit was explicitly preconnected to a file by the PROGRAM statement or by the execution control statement, and if the unit has never been closed, the unit is connected to the file to which it was explicitly preconnected.

If the unit is implicitly preconnected to a file, it is connected to that file.

For all other cases an error occurs.

INPUT/OUTPUT LIST

An input/output list specifies the entities whose values are transferred by an input/output statement. An input/output list can contain input/output list items and implied DO loops. The items in an input/output list must be separated by commas.

Input/output list items and implied DO loops are described in the following paragraphs.

Input/Output List Items

An input list item is an entity whose value is assigned by an input statement. An input list item can be any of the following:

Variable (except a control variable of an implied DO)

Array

Array element

Substring

Descriptor

Vector

If an input list item is a descriptor, data is input to the vector that is associated with that descriptor.

An output list item is an entity whose value is copied to a file by an output statement. An output list item can be any of the following:

Variable.

Array.

Array element.

Substring.

Descriptor.

Vector.

Any expression. A character expression that involves the concatentation of an operand whose length specification is unknown (specified by an asterisk in the CHARACTER statement) is not permitted unless the operand is a symbolic constant.

A descriptor, descriptor array, or descriptor array element.

A descriptor, descriptor array, or descriptor array element preceded by an ampersand.

If an output list item is a descriptor and you omit the ampersand, data is transferred from the vector associated with the descriptor. If you specify the ampersand, the descriptor value is transferred.

Implied DO Loops in Input/Output Statements

An implied DO loop is an input list that is assigned values repeatedly by an input statement, or an output list whose values are copied repeatedly to a file by an output statement. See figure 6-24 for the format of an implied DO loop.

When an implied DO loop that appears in an input/output statement is executed, the following steps are taken:

1. The expressions in the implied DO loop are defaulted or evaluated and, if necessary, their results are converted to the type of the control variable.
2. The control variable is initialized with the value specified in the first expression.
3. The iteration count, K, for the implied DO is established according to the following relation:

   K = MAX(0,INT((aexp2 + aexp3 - aexp1)/aexp3))

   Note that the iteration count can be zero.
4. The input/output list, iolist, is processed K times. After each time, cvar is incremented by aexp3.
5. Processing continues with the next input/output list item or implied DO.

When an implied DO loop appears in an input statement, the input list items in the input list of the implied DO loop are assigned a value each time the implied DO loop is iterated. The value of the control variable must not be affected by the data input.

When an implied DO loop appears in an output statement, the output list items in the output list of the implied DO loop are copied to the file each time the implied DO loop is iterated.

See figure 6-25 for an example of an implied DO loop list item. The values input and output by the input/output statements in the example are shown.

```
(iolist, cvar = aexp1,aexp2,aexp3)
```

iolist	An input/output list.
cvar	A variable of any arithmetic type except complex; cvar is used as the control variable for the implied DO loop. The control variable cvar must not also be the control variable of a containing implied DO loop.
$aexp_1$	A scalar arithmetic expression of any type except complex; $aexp_1$ is used as the initial value for the control variable.
$aexp_2$	A scalar arithmetic expression of any type except complex; $aexp_2$ is used as the terminal value for the control variable.
$aexp_3$	A scalar arithmetic expression of any type except complex; optional; $aexp_3$ is used as the incrementation value for the control variable. If $aexp_3$ is not specified, the incrementation value for the control variable is 1. The result of $aexp_3$ must not be zero.

Figure 6-24. Implied DO Loop Format For Input/Output Statements

```
            .
            .
            .
            REAL A(5)
            .
            .
            .
            READ(1,100) (A(I),I=1,5), B
      100   FORMAT(6(F5.2,1X))
            .
            .
            .
            WRITE(2,200) (I,A(I),I=1,5)
      200   FORMAT('Δ',I2,2X,F5.2)
            .
            .
            .

   Input:

      00.01Δ00.02Δ00.03Δ00.04Δ00.05Δ00.06Δ

   Output:

      ΔΔ1ΔΔ0.01
      ΔΔ2ΔΔ0.02
      ΔΔ3ΔΔ0.03
      ΔΔ4ΔΔ0.04
      ΔΔ5ΔΔ0.05
```

Figure 6-25. Implied DO Loop in Input/Output Statement Example

CARRIAGE CONTROL

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

See table 6-2 for a summary of the standard FORTRAN carriage control characters and their functions. Other carriage control characters might be available at your particular site.

TABLE 6-2. CARRIAGE CONTROL CHARACTERS

Character	Function
Δ	Output record is printed on next line (single-spacing)
0	One line is skipped and output record is printed on the following line (double-spacing)
1	Output record is printed on the top of the next page
+	Output record is printed on the current line (overprinting)

You can generate a carriage control character for formatted output by using any exit descriptor (although the X, H, and apostrophe descriptors are the most commonly used). Namelist and list-directed output automatically generate appropriate carriage control characters.

If you do not specify a carriage control character as the first character of each record to be printed by a line printer, unexpected line spacing may result.

FORMATTED INPUT/OUTPUT STATEMENTS

A formatted input/output statement transfers data between a sequential access external file, direct access external file, or internal file, and internal storage in a format that you specify. Format specification is described later in this section.

Those aspects of formatted input/output which are unique to direct access external files are discussed separately under Direct Access Input/Output Statements. Those aspects unique to internal file formatted input/output are discussed under Internal File Input/Output Statements. The remainder of this discussion assumes the most common file type for formatted input/output: a sequential access external file.

The unit you specify in a formatted input/output statement must be preconnected to a file capable of formatted input/output, or must be connected for formatted input/output. (You can connect a unit by using the OPEN statement. Preconnection can be implicit or can be done explicitly with the PROGRAM statement or with the execution control statement.)

If the unit you specify is preconnected, the processor connects the unit to the file before the input/output statement is executed. See the description of the UNIT specifier for more information about processor-determined unit connection.

A FMT specifier must appear in a formatted input/output statement. The input/output list is optional in a formatted input/output statement.

The formatted input/output statements are:

Formatted READ statement

Formatted WRITE statement

Formatted PRINT statement

Formatted PUNCH statement

Each of these statements is described in the following paragraphs.

FORMATTED READ STATEMENT

The formatted READ statement transfers data from a sequential access external file to internal storage in the format you specify. See figure 6-26 for the format of the formatted READ statement.

```
READ (cilist) ilist
    or
READ fid, ilist


cilist    A control information list. The
          following specifiers must appear in
          cilist:

              UNIT
              FMT

          The following specifiers can also
          appear in cilist:

              END
              ERR
              IOSTAT
              REC

ilist     An input list; optional.

fid       A format identifier. See the
          description of the FMT specifier for
          the items that can be specified for
          fid.

If the second form of the formatted READ
statement is used and ilist is not specified,
the comma separating fid from ilist must not
appear.
```

Figure 6-26. Formatted READ Statement Format

If the second form of the formatted READ statement shown in figure 6-26 is used, data is transferred from the unit 5HINPUT.

The number of words in the input list and the edit descriptors you specify in the associated format specification must correspond to the format of the input record. If the input list is omitted from the formatted READ statement, at least one record is skipped. (The actual number of records skipped is determined by the FORMAT statement.)

If a formatted READ statement attempts to read beyond the end of a file, an execution-time error occurs. You can avoid this error by specifying the END or IOSTAT specifier in the formatted READ statement.

See figure 6-27 for an example of the formatted READ statement. The values input by the formatted READ statements in the example are shown. The values input by the first formatted READ statement are input from the file connected to unit 1. When an end-of-file condition is detected during execution of the first formatted READ statement, control transfers to the statement labeled 10 and N is assigned the value -1. If an input error occurs during execution of the first formatted READ statement, control transfers to the statement labeled 20 and the variable N is assigned the number of the execution-time error.

The values input by the second formatted READ statement are input from the file called INPUT.

```
          .
          .
          .
          I = 0
1         I = I + 1
          READ(1,100,END=10,ERR=20,IOSTAT=N) A,B
100       FORMAT(2(F5.2,1X))
          READ 100, C,D
          AVG(I) = (A+B+C+D)/4
          GO TO 1
10        CALL PLOT(AVG,I-1)
          .
          .
          .
          STOP
20        CALL IOERR(N)
          .
          .
          .

Input:

    10.00Δ20.00Δ
    50.00Δ70.00Δ
```

Figure 6-27. Formatted READ Statement Example

FORMATTED WRITE STATEMENT

The formatted WRITE statement transfers data from internal storage to a sequential access external file in the specified format. See figure 6-28 for the format of the formatted WRITE statement.

If the output list is omitted from the formatted WRITE statement and if an empty format specification of the form () is used, one output line is skipped.

See figure 6-29 for an example of the formatted WRITE statement. The values output by the formatted WRITE statements in the example are shown. The values are output to the file connected to unit 2. If an output error occurs during execution of one of the formatted WRITE statements, control transfers to the statement labeled 20 and the variable N is assigned the number of the execution-time error.

```
WRITE (cilist) olist

cilist    A control information list. The
          following specifiers must appear in
          cilist:

              UNIT
              FMT

          The following specifiers can also
          appear in cilist:

              ERR
              IOSTAT
              REC

olist     An output list; optional.
```

Figure 6-28. Formatted WRITE Statement Format

```
                .
                .
                .
                B = 5.0
                DO 10 I = 1,3
                A = 3.0 + I
                IF (A.LT.B) THEN
                WRITE(2,200,ERR=20,IOSTAT=N) A,B
          200   FORMAT('Δ',F5.2,'ΔISΔLESSΔTHANΔ',F5.2)
                ELSEIF (A .GT. B) THEN
                WRITE(2,201,ERR=20,IOSTAT=N) A,B
          201   FORMAT('Δ',F5.2,'ΔISΔGREATERΔTHANΔ',F5.2)
                ELSE
                WRITE(2,202,ERR=20,IOSTAT=N) A,B
          202   FORMAT('Δ',F5.2,'ΔISΔEQUALΔTOΔ',F5.2)
                ENDIF
          10    CONTINUE
                .
                .
                .
          20    CALL IOERR(N)
                .
                .
                .

Output:

    ΔΔ4.00ΔISΔLESSΔTHANΔΔ5.00
    ΔΔ5.00ΔISΔEQUALΔTOΔΔ5.00
    ΔΔ6.00ΔISΔGREATERΔTHANΔΔ5.00
```

Figure 6-29. Formatted Write Statement Example

FORMATTED PRINT STATEMENT

The formatted PRINT statement transfers data from internal storage to the unit 6HOUTPUT. See figure 6-30 for the format of the formatted PRINT statement.

If the output list does not appear in the formatted PRINT statement, format control continues until the format is exhausted or until the first repeatable or colon edit descriptor is encountered. Thus, at least one line is printed.

See figure 6-31 for an example of the formatted PRINT statement. The values output by the formatted PRINT statements in the example are shown. The values are written to the unit 6HOUTPUT.

```
PRINT fid, olist


fid       A format identifier. See the
          description of the FMT specifier for
          the items that can be specified for
          fid.

olist     An output list; optional


If olist is not specified, the comma separating
fid from olist must not appear.
```

Figure 6-30. PRINT Statement Format

```
                .
                .
                .
                REAL A(5)/5*30.0/
                .
                .
                .
                PRINT 200
          200   FORMAT('ΔFILEΔOUTPUTΔCONTAINS:')
                PRINT 201,(A(I),I=1,5)
          201   FORMAT('Δ',5(F4.1,1X))
                .
                .
                .

Output:

    ΔFILE OUTPUT CONTAINS:
    Δ30.0Δ30.0Δ30.0Δ30.0Δ30.0Δ
```

Figure 6-31. PRINT Statement Example

FORMATTED PUNCH STATEMENT

The formatted PUNCH statement transfers data from internal storage to the unit 5HPUNCH in the format you specify. See figure 6-32 for the format of the formatted PUNCH statement.

If the output list does not appear in the formatted PUNCH statement, format control continues until the format is exhausted or until the first repeatable or colon edit descriptor is encountered. Thus, at least one line is written.

See figure 6-33 for an example of the formatted PUNCH statement. The values output by the formatted PUNCH statements in the example are shown. The values are output to the unit 5HPUNCH.

```
PUNCH fid, olist

fid      A format identifier. See the
         description of the FMT specifier for
         the items that can be specified for
         fid.

olist    An output list; optional.

If olist is not specified, the comma separating
fid from olist must not appear.
```

Figure 6-32. PUNCH Statement Format

```
              .
              .
              .
              REAL A(5)/5*20.0/
              .
              .
              .
              PUNCH 200
      200     FORMAT('FILE PUNCH CONTAINS:')
              PUNCH 201,(A(I),I=1,5)
      201     FORMAT(5(F5.2,1X))
              .
              .
              .

   Output:

      FILE PUNCH CONTAINS:
      20.0 20.0 20.0 20.0 20.0
```

Figure 6-33. PUNCH Statement Example

FORMAT SPECIFICATION

A format specification is a list of edit descriptors that specifies how data is to be converted during execution of a formatted input/output statement. Each item that appears in the input/output list of a formatted input/output statement must correspond to an edit descriptor that appears in the format specification. You must use a format specification with each formatted input/output statement. A format specification can appear in one of three places:

In a FORMAT statement

In a character expression, character variable, character array, character array element, or substring that appears as a format specification in the formatted input/output statement

In a noncharacter array that appears as a format specification in the formatted input/output statement

Each of these methods of format specification is described in the following paragraphs. The edit descriptors are described later in this section.

FORMAT STATEMENT

The FORMAT statement is a nonexecutable statement that provides a format specification for a formatted input/output statement. See figure 6-34 for the format of the FORMAT statement.

```
sl FORMAT (fspec)

sl       A statement label

fspec    A format specification consisting of
         zero or more edit descriptors sepa-
         rated by commas. The comma can be
         omitted:

           Between a P edit descriptor and an
           immediately following F, E, D, or
           G edit descriptor

           Before or after a slash edit
           descriptor

           Before or after a colon edit
           descriptor
```

Figure 6-34. FORMAT Statement Format

If you supply a FORMAT statement, it must appear in the same program unit as the formatted input/output statements that use it.

See figure 6-35 for an example of the FORMAT statement. The first FORMAT statement in the example provides a format specification for the formatted READ statement. The second FORMAT statement in the example provides a format specification for the formatted WRITE statement.

CHARACTER FORMAT SPECIFICATION

A character format specification is a character expression whose result is a format specification, or a character variable, character array, character array element, or substring that contains a format specification. The format specification must be enclosed in parentheses. One or more blanks can precede the left parenthesis. Any characters following the right parenthesis are disregarded.

If you supply a character format specification, it must be specified by the FMT specifier in the formatted input/output statement that uses it.

If an apostrophe appears in a character constant or in an H output field that is part of a character format specification, two consecutive apostrophes must be written for each apostrophe. The two consecutive apostrophes represent only one character of the H output field and they are counted as one character in specifying the length of the field.

If an apostrophe edit descriptor is used in a character format specification, two consecutive apostrophes must be used for each of the delimiting apostrophes of the apostrophe descriptor. In order to represent one apostrophe in an apostrophe descriptor that appears in a character constant format specification, eight consecutive apostrophes must be specified. The eight consecutive apostrophes represent only one character of the apostrophe output field.

See figure 6-36 for an example of a character format specification. The character array CHARFMT in the example contains a format specification that is used by the formatted READ statement. The formatted WRITE statement uses a character expression as a format specification.

```
      .
      .
      .
      READ(1,100) A,B,C
100   FORMAT(3X,F5.2,T15,2(F3.1,2X))
      .
      .
      .
      WRITE(2,200)
200   FORMAT('1SUMMARY OF'/' OUTSTANDING ORDERS:')
      .
      .
      .
```

Input:

ΔΔΔ31.49ΔΔ2.0Δ2.1ΔΔ2.2ΔΔ

31.49	2.1	2.2
A	B	C

Output:

```
1SUMMARYΔOF
ΔOUTSTANDING ORDERS:
```

Figure 6-35. FORMAT Statement Example

```
      .
      .
      .
      INTEGER INPUT1(2), INPUT2(2)
      CHARACTER*10 CHARFMT(2)
      DATA CHARFMT /'(I5,5X,I5)',(5X,I5,I5)'/,MONEY/2525/
      .
      .
      .
      DO 10 I = 1,2
      READ(1,CHARFMT(I)) INPUT1(I),INPUT2(I)
10    CONTINUE
      .
      .
      .
      WRITE(2,'(''ΔTHE ANSWER ISΔ'',I4)')MONEY
      .
      .
      .
```

Input:

1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7

1 2 3 4 5	1 2 3 4 5
INPUT1(1)	INPUT2(1)

1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7

6 7 8 9 0	1 2 3 4 5
INPUT1(2)	INPUT2(2)

Output:

MONEY

2 5 2 5

THEΔANSWERΔISΔ2525

Figure 6-36. Character Format Specification Example

NONCHARACTER FORMAT SPECIFICATION

Format specifications can be contained in a noncharacter array. The rules for noncharacter format specification are the same as for character format specification.

EDIT DESCRIPTORS

Edit descriptors describe the fields of an input record and the fields of an output record, and specify how an input/output operation is to convert data.

The two types of edit descriptors are repeatable edit descriptors and nonrepeatable edit descriptors. The repeatable edit descriptors are: A, B, D, E, F, G, I, L, R, and Z. A repeatable edit descriptor can be preceded by a repeat specification. Repeatable edit descriptors correspond to the items in the input/output list of the formatted input/output statement.

The nonrepeatable edit descriptors are: BN, BZ, H, P, T, TL, TR, S, SP, SS, X, the apostrophe, the slash, and the colon. A nonrepeatable edit descriptor must not be preceded by a repeat specification. Nonrepeatable edit descriptors do not correspond to the items in the input/output list of the formatted input/output statement.

When a formatted input/output statement is executed, format control is initiated. The actions performed under format control depend on the edit descriptors that appear in the format specification and the items in the input/output list.

If no items appear in the input/output list, format control continues until the format is exhausted or until the first repeatable or colon edit descriptor is encountered. Thus, as least one record is read or written.

If one or more items appear in the input/output list, at least one repeatable edit descriptor must appear in the format specification. Each input/output list item corresponds to a repeatable edit descriptor; however, an input/output list item of type complex corresponds to two edit descriptors.

Edit descriptors are evaluated from left to right during execution of a formatted input/output statement. After each repeatable edit descriptor, H edit descriptor, or apostrophe edit descriptor is processed, the file is positioned after the last field read or written. You can use the T, TL, TR, X, and slash descriptors to change the position in a record. You can repeat groups of edit descriptors by enclosing them in parentheses and preceding the group with a repeat specification. A repeat specification, if given, must be greater than 0 and less than 256.

Format control terminates when all of the items in the input/output list have been input or output and another repeatable edit descriptor, colon edit descriptor, or the end of the format specification is encountered. There can be more repeatable edit descriptors than input/output list items; the excess edit descriptors are not interpreted.

If the rightmost parenthesis of the format specification is reached before all of the items in the input/output list have been input or output, the input file or the output file is positioned at the beginning of the next record, and format control is continued with the edit descriptor that follows the left parenthesis which corresponds to the preceding right parenthesis. If there is no preceding right parenthesis, format control continues with the edit descriptor that follows the first left parenthesis of the format specification.

If a format specification is reused in this manner, the reused portion of the format specification must contain at least one repeatable edit descriptor. If format control reverts to a parenthesis that is preceded by a repeat specification, the repeat specification is reused. Reuse of a format specification has no effect on the scale factor, sign control, or blank interpretation control that is in effect.

The edit descriptors and their functions are summarized in table 6-3. Each of the edit descriptors is described in the following paragraphs.

A Descriptor

The A descriptor formats character data during input/output operations. See figure 6-37 for the format of the A descriptor.

```
rA[w]

r   An unsigned integer constant (0 < r < 256)
    that is used as a repeat specification;
    optional. The default is 1.

w   An unsigned integer constant (0 < w < 256)
    that specifies the width in characters of
    the A input field or A output field. If w
    is omitted, the length of the list item
    determines the length of the external field.
```

Figure 6-37. A Descriptor Format

TABLE 6-3. EDIT DESCRIPTORS

Descriptor	Purpose
A	Formats data of any type but bit as though it were character data
B	Formats bit data
BN	Causes blanks in numeric input fields to be ignored
BZ	Causes blanks in numeric input fields to be interpreted as zeros
D	Formats real, double-precision, half-precision, and complex data
E	Formats real, double-precision, half-precision, and complex data
F	Formats real, double-precision, half-precision, and complex data
G	Formats real, double-precision, half-precision, and complex data
H	Places a string of characters in an output record
I	Formats integer data
L	Formats logical data
P	Establishes a scale factor for data formatted by D, E, F, and G descriptors
R	Formats data of any type but bit as though it were character data
S	Suppresses printing of the plus sign during output of numeric data
SP	Suppresses printing of the plus sign during output of numeric data
SS	Suppresses printing of the plus sign during output of numeric data
T	Specifies the column from which the next character is to be input or to which the next character is to be output
TL	Moves the input/output record pointer to the left
TR	Moves the input/output record pointer to the right
X	Moves the input/output record pointer to the right
Z	Formats data of any type but bit as though it were hexadecimal data
'	Delimits a string of characters and places them in an output record
/	Indicates that no more data is to be input from the current record or output to the current record during execution of the current input/output statement
:	Terminates format control if there are no more items in the input/output list

Input

When the A descriptor formats data during an input operation, the value in the input field is assigned to the input list item. The number of characters input is the number of characters in the A input field. The input list item can be of any data type.

The data that appears in an A input field can be a string of characters. Any of the characters listed in appendix A can appear in an A input field. Blanks are significant characters in an A input field.

If the length of the input list item is less than the width of the A input field, the rightmost characters in the A input field are assigned to the input list item.

If the length of the input list item is greater than the width of the A input field, the character value input is left-justified and blank-filled in the input list item.

Output

When the A descriptor formats data during an output operation, the value of the output list item is placed in the output field. The number of characters output is the number of characters in the A output field. The output list item can be of any data type.

Any of the characters listed in appendix A can appear in an A output field.

If the length of the output list item is less than the width of the A output field, the character value that is output is right-justified and blank-filled in the A output field.

If the length of the output list item is greater than the width of the A output field, the leftmost characters of the output list item are placed in the A output field.

Examples

See figure 6-38 for an example of the A descriptor.

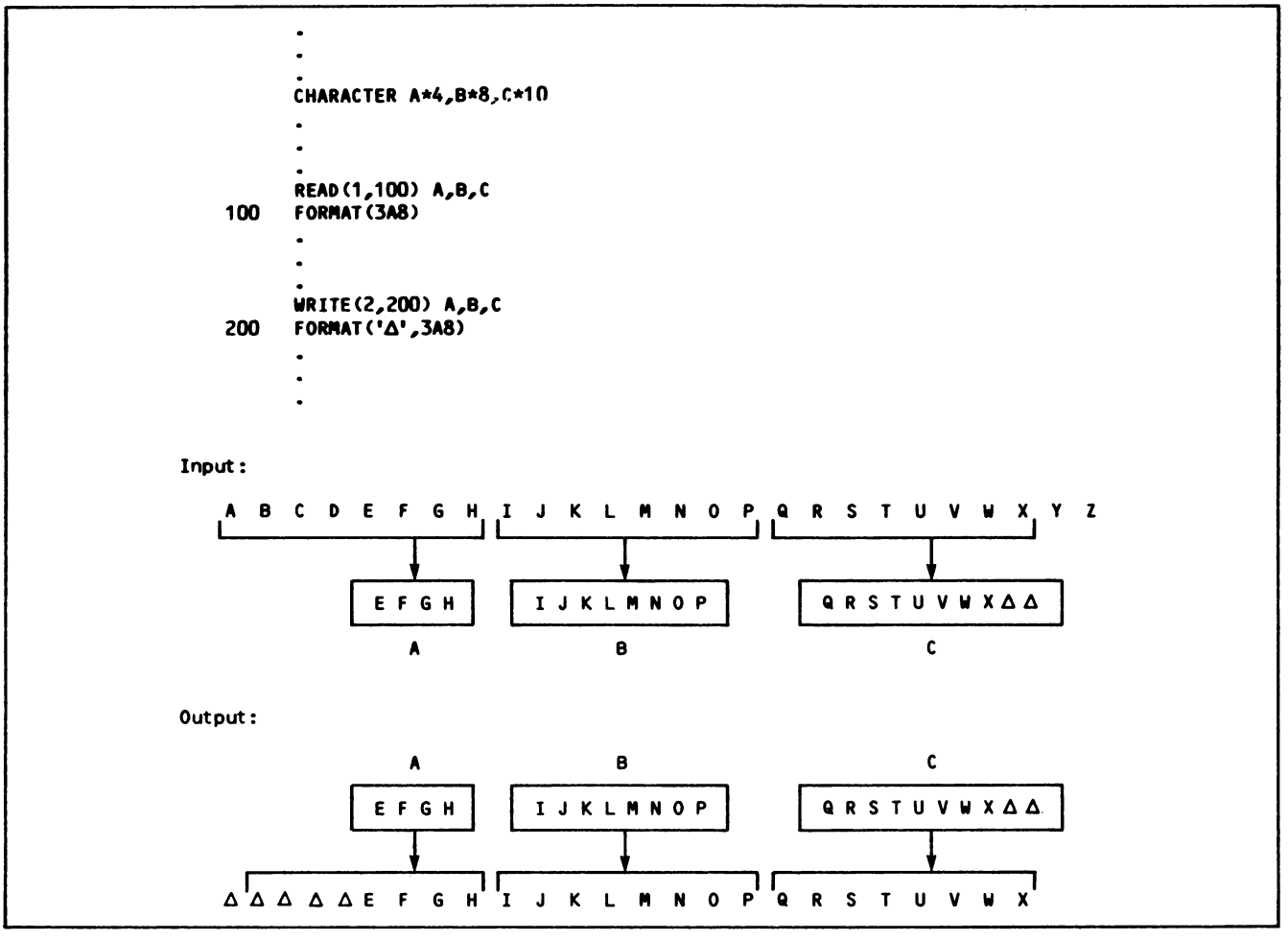


Figure 6-38. A Descriptor Example

B Descriptor

The B descriptor formats bit data during input/output operations. See figure 6-39 for the format of the B descriptor.

```
rBw

r   An unsigned integer constant (0 < r < 256)
    that is used as a repeat specification;
    optional. The default is 1.

w   An unsigned integer constant (0 < w < 256)
    that specifies the width in characters of
    the B input field or B output field.
```

Figure 6-39. B Descriptor Format

Input

When the B descriptor formats data during an input operation, the value in the input field is converted to type bit and is assigned to the input list item. One bit is input. The input list item must be of type bit.

The data that appears in a B input field must be in the proper format. The B input field must contain a 0 or a 1 in the rightmost column; all other columns must be blank.

Output

When the B descriptor formats data during an output operation, the value of the output list item is converted to a string of characters. These characters are placed in the output field. One bit is output. The output list item must be of type bit.

The B output field contains a 0 or a 1 in the rightmost column; all other columns are blank.

Examples

See figure 6-40 for an example of the B descriptor.

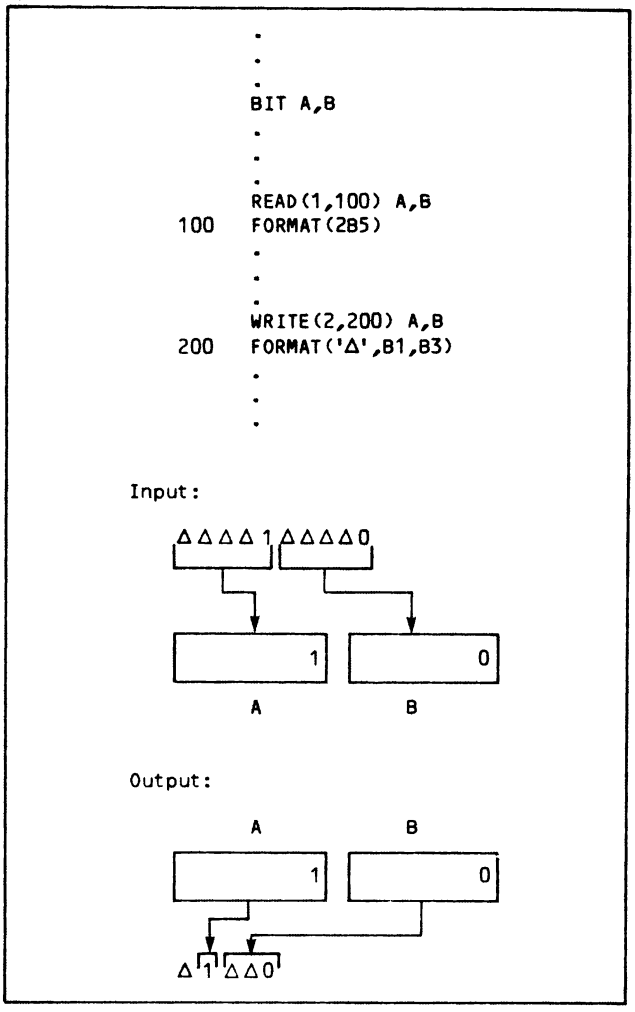


Figure 6-40. B Descriptor Example

```
BN
```

Figure 6-41. BN Descriptor Format

BN Descriptor

The BN descriptor causes blanks that appear within subsequent numeric fields of an input record to be ignored. See figure 6-41 for the format of the BN descriptor.

The BN descriptor cannot be associated with an input/output list item.

The BN descriptor affects only blanks that appear within numeric fields of an input record. However, if a numeric field of an input record contains all blanks, the numeric field has the value 0.

Normally, blanks that appear within numeric fields of an input record are either treated as zeros or are ignored, depending on the BLANK specifier in effect for the file. The BN descriptor overrides any previous specification; it is effective only for the input list items whose edit descriptors are processed after the BN descriptor in the format specification.

The BN descriptor affects only input fields described by the I, F, E, D, G, and Z edit descriptors. The BN descriptor has no effect during execution of an output statement.

See figure 6-42 for an example of the BN descriptor.

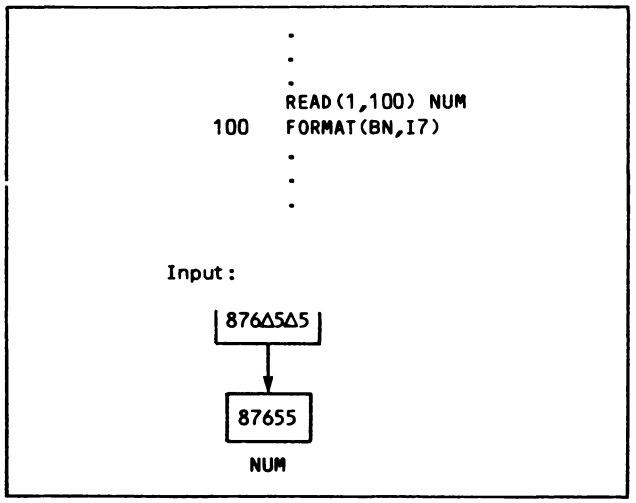


Figure 6-42. BN Descriptor Example

BZ Descriptor

The BZ descriptor causes blanks that appear within subsequent numeric fields of an input record to be treated as zeros. See figure 6-43 for the format of the BZ descriptor.

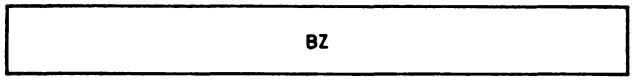


Figure 6-43. BZ Descriptor Format

The BZ descriptor cannot be associated with an input/output list item.

The BZ descriptor affects only blanks that appear within numeric fields of an input record; leading blanks are not affected.

Normally, blanks that appear within numeric fields of an input record are either treated as zeros or are ignored, depending on the BLANK specifier that appears in the OPEN statement for the unit. The BZ descriptor overrides any previous specification; it is effective only for the input list items whose edit descriptors appear to the right of the BZ descriptor in the format specification.

The BZ descriptor affects only input fields described by the I, F, E, D, G, and Z edit descriptors. The BZ descriptor has no effect during execution of an output statement.

See figure 6-44 for an example of the BZ descriptor.

D Descriptor

The D descriptor formats double-precision data during input/output operations. The D descriptor can also format half-precision and real data during input/output operations. Two consecutive D, E, F, or G descriptors can format complex data during input/output operations. See figure 6-45 for the format of the D descriptor.

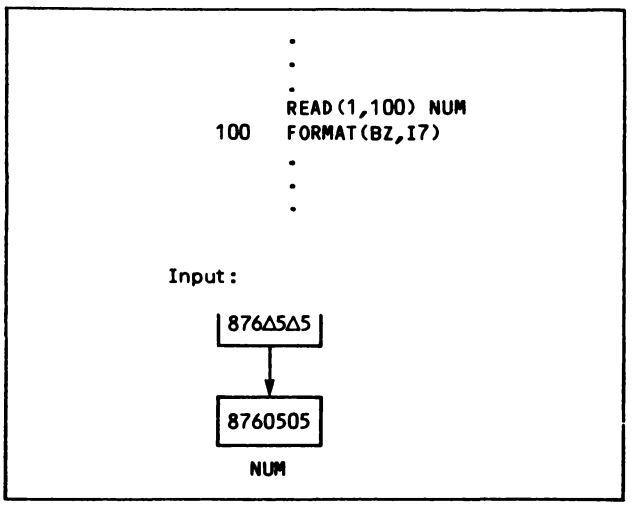


Figure 6-44. BZ Descriptor Example

rDw.d

r An unsigned integer constant (0 < r < 256) that is used as a repeat specification; optional. The default is 1.

w An unsigned integer constant (0 < w < 256) that specifies the width in characters of the D input field or D output field (including the exponent).

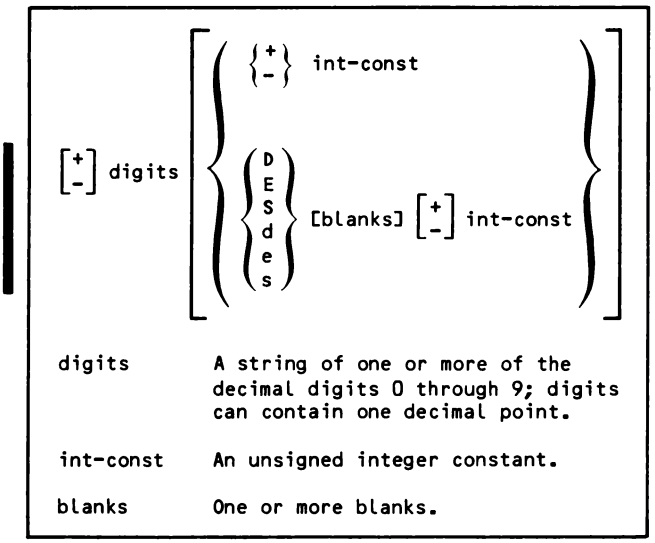
d An unsigned integer constant less than or equal to w that specifies the number of digits to the right of the decimal point in the mantissa.

Figure 6-45. D Descriptor Format

Input

When the D descriptor formats data during an input operation, the value in the input field is converted to type half-precision, real, or double-precision and is assigned to the input list item. The data type to which the value is converted is the data type of the input list item to which the value is assigned. The input list item must be of type half-precision, real, double-precision, or complex. If the input list item is of type complex, two consecutive D, E, F, or G descriptors are required: the first is for the real part of the complex value, the second is for the imaginary part of the complex value. (The two descriptors may be different. Note also that nonrepeatable edit descriptors can appear between the two successive descriptors.)

The data that appears in a D input field must be in the proper format. See figure 6-46 for the format of a D input field.



digits	A string of one or more of the decimal digits 0 through 9; digits can contain one decimal point.
int-const	An unsigned integer constant.
blanks	One or more blanks.

Figure 6-46. D, E, F, and G Input Field Format

Blanks that appear in a D input field are interpreted according to any BN or BZ edit descriptor, or according to any BLANK specifier in effect for the file. If you do not specify a BN or BZ edit descriptor or a BLANK specifier, blanks that appear in a D input field are ignored.

A decimal point that appears in a D input field overrides the decimal point position specified in the D descriptor.

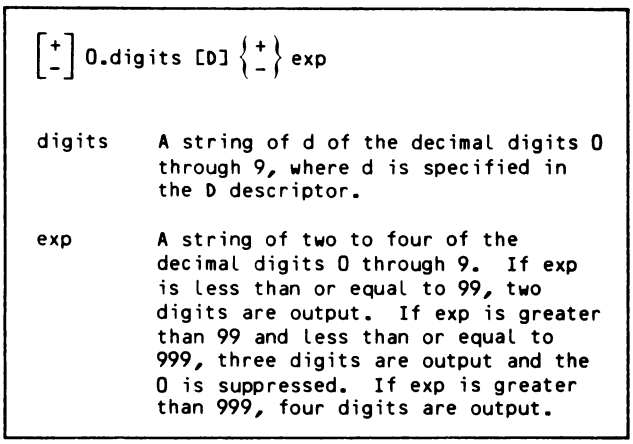
Output

When the D descriptor formats data during an output operation, the value of the output list item is converted to a string of characters. These characters are placed in the output field. The output list item must be of type half-precision, real, double-precision, or complex. If the output list item is of type complex, two consecutive D, E, F, or G descriptors are required: the first is for the real part of the complex value, the second is for the imaginary part of the complex value. (The two descriptors may be different. Note also that nonrepeatable edit descriptors can appear between the two descriptors.)

See figure 6-47 for the format of a D output field. The scale factor k controls the decimal normalization. The output field contains k leading zeros and (d-k) significant digits after the decimal point. If k is greater than zero and less than (d+2), the output field contains k significant digits to the left of the decimal point and (d-k+1) significant digits to the right of the decimal point. Other values of k are not permitted.

If the length of the output list item is less than the width of the D output field specified, the value of the output list item is right-justified and blank-filled in the D output field.

If the length of the output list item is greater than the width of the D output field, the D output field is filled with asterisks. However, asterisks are not printed if the width of the field is not exceeded when optional characters are omitted.



digits	A string of d of the decimal digits 0 through 9, where d is specified in the D descriptor.
exp	A string of two to four of the decimal digits 0 through 9. If exp is less than or equal to 99, two digits are output. If exp is greater than 99 and less than or equal to 999, three digits are output and the 0 is suppressed. If exp is greater than 999, four digits are output.

Figure 6-47. D Output Field Format

Output of the sign of a positive mantissa is controlled by the S, SP, and SS descriptors.

Examples

See figure 6-48 for an example of the D descriptor.

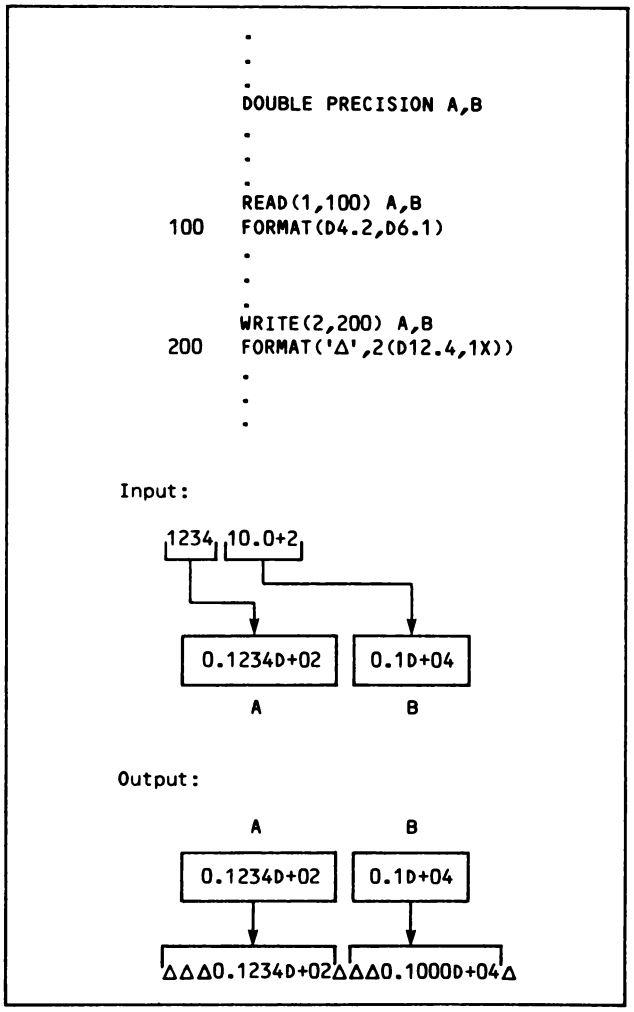


Figure 6-48. D Descriptor Example

E Descriptor

The E descriptor formats real data during input/output operations. The E descriptor can also format half-precision and double-precision data during input/output operations. Two consecutive D, E, F, or G descriptors can format complex data during input/output operations. See figure 6-49 for the format of the E descriptor.

```
rEw.d
   or
rEw.dEe
```

r An unsigned integer constant (0 < r < 256) that is used as a repeat specification; optional. The default is 1.

w An unsigned integer constant (0 < w < 256) that specifies the width in characters of the E input field or E output field (including the exponent).

d An unsigned integer constant less than or equal to w that specifies the number of digits to the right of the decimal point in the mantissa.

e An unsigned integer constant (0 < e < 256) that specifies the number of digits in the exponent field; e has no effect on input.

Figure 6-49. E Descriptor Format

Input

When the E descriptor formats data during an input operation, the value in the input field is converted to type half-precision, real, or double-precision and is assigned to the input list item. The data type to which the value is converted is the data type of the input list item to which the value is assigned. The input list item must be of type half-precision, real, double-precision, or complex. If the input list item is of type complex, two consecutive D, E, F, or G descriptors are required: the first is for the real part of the complex value; the second is for the imaginary part of the complex value. (The two descriptors can be different. Note also that nonrepeatable descriptors can appear between the the two descriptors.)

The data that appears in an E input field must be in the proper format. See figure 6-46 for the format of an E input field.

Leading blanks in an E input field are ignored. Other blanks that appear in an E input field are interpreted according to any BN or BZ edit descriptor, or according to any BLANK specifier in the OPEN statement that connected the file to the unit. If you do not specify a BN or BZ edit descriptor or a BLANK specifier, blanks that appear in an E input field are ignored.

A decimal point that appears in an E input field overrides the decimal point position specified in the E descriptor.

Output

When the E descriptor formats data during an output operation, the value of the output list item is converted to a string of characters. These characters are placed in the output field. The output list item must be of type half-precision, real, double-precision, or complex. If the output list item is of type complex, two consecutive D, E, F, or G descriptors are required: the first is for the real part of the complex value; the second is for the imaginary part of the complex value. (The two descriptors may be different. Note also that nonrepeatable edit descriptors can appear between the two descriptors.)

See figure 6-50 for the format of an E output field. The scale factor k controls the decimal normalization. If -d<k<= 0, the output field after the decimal point contains |k| leading zeros and (d-|k|) significant digits. A zero is output to the left of the decimal point if space permits. If k is greater than zero and less than (d+2), the output field contains k significant digits to the left of the decimal point and (d-k+1) significant digits to the right of the decimal point. Other values of k are not permitted.

```
[+] 0.digits [E] {+} exp
[-]              {-}
```

digits A string of d of the decimal digits 0 through 9, where d is specified in the E descriptor.

exp A string of two to four of the decimal digits 0 through 9. If exp is less than or equal to 99, two digits are output. If exp is greater than 99 and less than or equal to 999, three digits are output and the E is suppressed. If exp is greater than 999, four digits are output and the E is suppressed.

If an E descriptor of the form rEw.dEe is specified e digits are output.

Figure 6-50. E Output Field Format

If the length of the output list item is less than the width of the E output field specified, the value of the output list item is right-justified and blank-filled in the E output field.

If the length of the output list item is greater than the width of the E output field, the E output field is filled with asterisks. However, asterisks are not printed if the width of the field is not exceeded when optional characters are omitted.

Output of the sign of a positive mantissa is controlled by the S, SP, and SS descriptors.

Examples

See figure 6-51 for an example of the E descriptor.

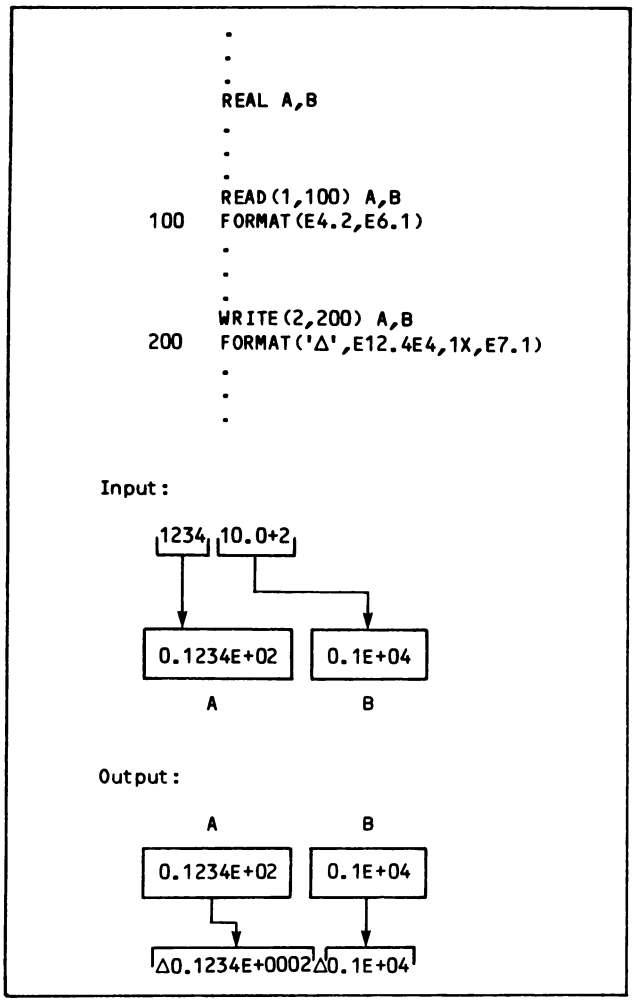


Figure 6-51. E Descriptor Example

F Descriptor

The F descriptor formats real data during input/output operations. The F descriptor can also format half-precision and double-precision data during input/output operations. Two consecutive D, E, F, or G descriptors can format complex data during input/output operations. See figure 6-52 for the format of the F descriptor.

Input

When the F descriptor formats data during an input operation, the value in the input field is converted to type half-precision, real, or double-precision and is assigned to the input list item. The data type to which the value is converted is the data type of the input list item to which the value is assigned. The input list item must be of type half-precision, real, double-precision, or complex. When paired with a D, E, G, or another F descriptor, the F descriptor can format complex data during input/output operations; the first descriptor is for the real part of the complex value; the second is for the imaginary part of the complex value. (The two descriptors may be different. Note also that nonrepeatable edit descriptors can appear between the two descriptors.)

```
rFw
  or
rFw.d

r   An unsigned integer constant (0 < r < 256)
    that is used as a repeat specification;
    optional. The default is 1.

w   An unsigned integer constant (0 < w < 256)
    that specifies the width in characters of
    the F input field or F output field.

d   An unsigned integer constant less than or
    equal to w that specifies the number of
    digits to the right of the decimal point.
```

Figure 6-52. F Descriptor Format

The data that appears in an F input field must be in the proper format. See figure 6-46 for the format of an F input field.

Blanks that appear in an F input field are interpreted according to any BN or BZ edit descriptor, or according to any BLANK specifier in the OPEN statement in effect for the file. If you do not specify a BN or BZ edit descriptor or a BLANK specifier, blanks that appear in an F input field are ignored.

A decimal point that appears in an F input field overrides the decimal point position specified in the F descriptor.

Output

When the F descriptor formats data during an output operation, the value of the output list item is converted to a string of characters. These characters are placed in the output field. The output list item must be of type half-precision, real, double-precision, or complex. If the output list item is of type complex, two consecutive D, E, F, or G descriptors are required: the first is for the real part of the complex value; the second is for the imaginary part of the complex value. (The two descriptors can be different. Note also that nonrepeatable edit descriptors can appear between the two descriptors.)

See figure 6-53 for the format of an F output field.

```
[+]
[-] digits

digits   A string of the decimal digits 0
         through 9; digits contains one
         decimal point.
```

Figure 6-53. F Output Field Format

If the length of the output list item is less than the width of the F output field specified, the value of the output list item is right-justified and blank-filled in the F output field.

If the length of the output list item is greater than the width of the F output field, the F output field is filled with asterisks. However, asterisks are not printed if the width of the field is not exceeded when optional characters are omitted.

Output of the sign of a positive value is controlled by the S, SP, and SS descriptors.

Examples

See figure 6-54 for an example of the F descriptor.

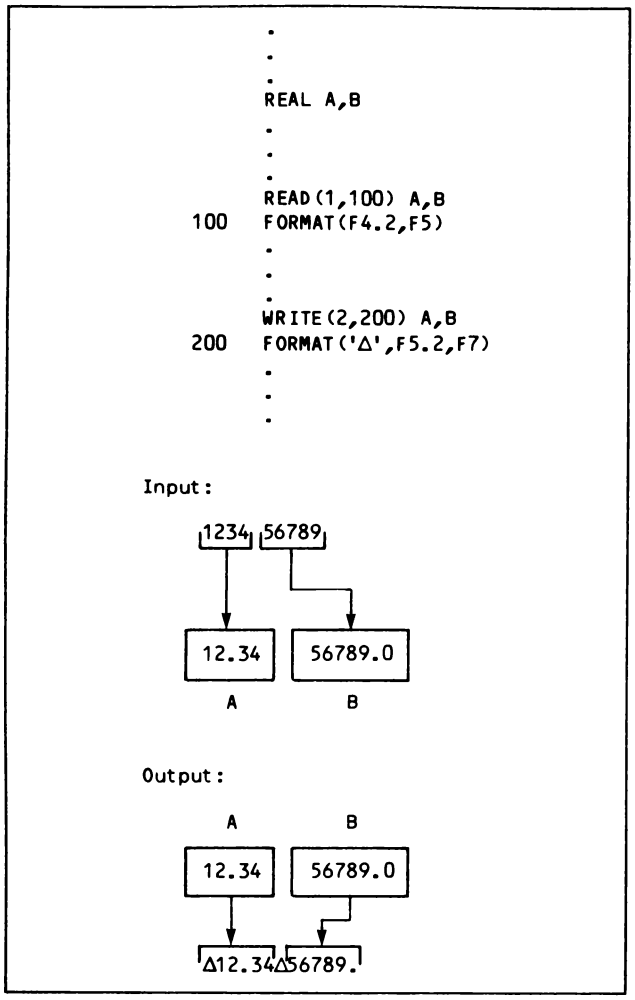


Figure 6-54. F Descriptor Example

G Descriptor

The G descriptor formats half-precision, real, and double-precision data during input/output operations. A G descriptor can be paired with a D, E, F, or another G descriptor to format complex data during input/output operations. See figure 6-55 for the format of the G descriptor.

Input

When the G descriptor formats data during an input operation, the value in the input field is converted to type half-precision, real, or double-precision

```
rGw
    or
rGw.d
    or
rGw.dEe
```

r An unsigned integer constant (0 < r < 256) that is used as a repeat specification; optional. The default is 1.

w An unsigned integer constant (0 < w < 256) that specifies the width in characters of the G input field or G output field (including the exponent).

d An unsigned integer constant less than w that specifies the number of digits to the right of the decimal point in the mantissa.

e An unsigned integer constant (0 < e < 256) that specifies the number of digits in the exponent field; e has no effect on input.

Figure 6-55. G Descriptor Format

and is assigned to the input list item. The data type to which the value is converted is the data type of the input list item to which the value is assigned. The input list item must be of type half-precision, real, double-precision, or complex. If the input list item is of type complex, two consecutive D, E, F, or G descriptors are required: the first is for the real part of the complex value; the second is for the imaginary part of the complex value. (The two descriptors can be different. Note also that nonrepeatable edit descriptors can appear between the two descriptors.)

The data that appears in a G input field must be in the proper format. See figure 6-46 for the format of a G input field.

Blanks that appear within a G input field are interpreted according to any BN or BZ edit descriptor, or according to any blank specifier in effect for the file. If you do not specify a BN or BZ edit descriptor or a BLANK specifier, blanks that appear within a G input field are ignored.

A decimal point that appears in a G input field overrides the decimal point position specified in the G descriptor.

Output

When the G descriptor formats data during an output operation, the value of the output list item is converted to a string of characters. These characters are placed in the output field. The output list item must be of type half-precision, real, double-precision, or complex. If the output list item is of type complex, two consecutive D, E, F, or G descriptors are required: the first is for the real part of the complex value; the second is for the imaginary part of the complex value. (The two descriptors can be different. Note also that nonrepeatable edit descriptors can appear between the two descriptors.)

The format of a G output field depends on the magnitude of the data being output. If the data being output is no less than 0.1 and no greater than 10**d, the format of the G output field is the same as the format of the F output field; however, the scale factor, if any, is ignored and n blanks are inserted to the right of the G output field when the magnitude of the output data is in this range. (The value of n is 4 for the Gw.d form and e+2 for the Gw.dEe form.) See figure 6-53 for the format of an F output field.

If the data being output is less than 0.1 or greater than 10**d, the format of the G output field is the same as the format of the E output field and the scale factor is effective. See figure 6-50 for the format of the E output field.

If the length of the output list item is less than the width of the G output field specified, the value of the output list item is right-justified and blank-filled in the G output field.

If the length of the output list item is greater than the width of the G output field, the G output field is filled with asterisks. However, asterisks are not printed if the width of the field is not exceeded when optional characters are omitted.

Output of the sign of a positive value is controlled by the S, SP, and SS descriptors.

Examples

See figure 6-56 for an example of the G descriptor.

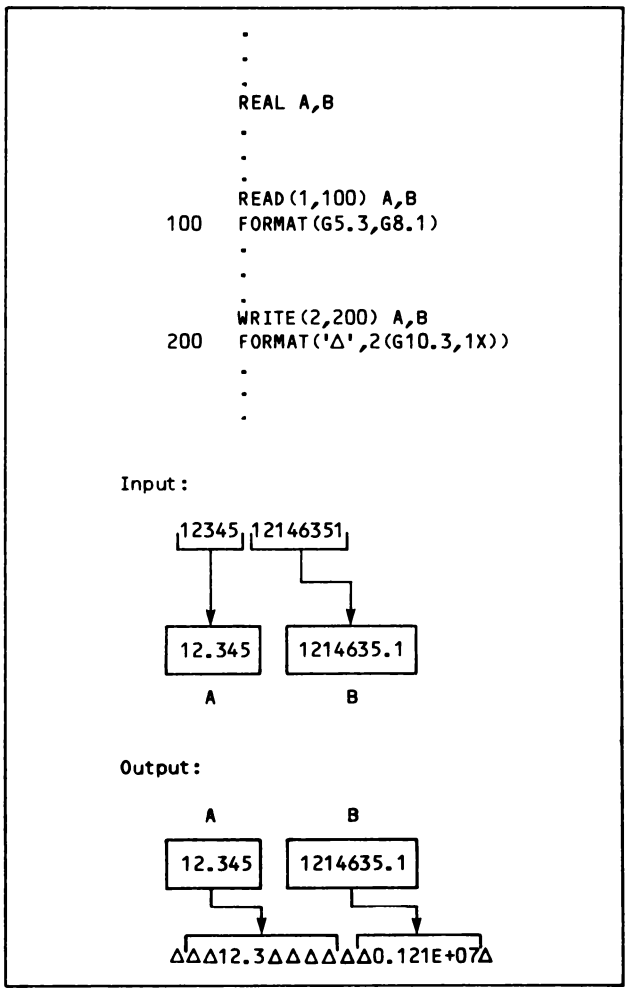


Figure 6-56. G Descriptor Example

H Descriptor

The H descriptor causes a string of characters to be placed in an output record. The H descriptor can be used only for output operations. See figure 6-57 for the format of the H descriptor.

The H descriptor cannot be associated with an input/output list item.

When the H descriptor is used for an output operation, the string of characters specified in the H descriptor is placed in the output field. The number of characters output is the number of characters in the H output field.

Any of the characters listed in appendix A can appear in an H output field. Blanks are significant characters in an H output field.

See figure 6-58 for an example of the H descriptor.

nHstring

n	An unsigned integer constant (0 < n < 256) that specifies the exact number of characters in string.
string	A string of characters from the CYBER 200 character set. This string begins in the next character position after the H and must contain exactly the number of characters specified in n.

Figure 6-57. H Descriptor Format

I Descriptor

The I descriptor formats integer data during input/output operations. See figure 6-59 for the format of the I descriptor.

```
      .
      .
      .
      WRITE(2,'(18HΔPROGRAM''SΔOUTPUT:)')
      .
      .
      .

 Output:

     PROGRAM'SΔOUTPUT:
```

Figure 6-58. H Descriptor Example

```
rIw
   or
rIw.m

r   An unsigned integer constant (0 < r < 256)
    that is used as a repeat specification;
    optional. The default is 1.

w   An unsigned integer constant (0 < w < 256)
    that specifies the width in characters of
    the I input field or I output field.

m   An unsigned integer constant less than or
    equal to w that specifies the minimum
    number of digits to be output to the I
    output field; m has no effect on input.
```

Figure 6-59. I Descriptor Format

Input

When the I descriptor formats data during an input operation, the value in the input field is converted to type integer and is assigned to the input list item. The input list item must be of type integer.

The data that appears in an I input field must be of the same format as an integer constant.

Blanks that appear within an I input field are interpreted according to any BN or BZ edit descriptor, or according to any blank specifier in effect for the file. If you do not specify a BN or BZ edit descriptor or a BLANK specifier, blanks that appear within an I input field are ignored.

Output

When the I descriptor formats data during an output operation, the value of the output list item is converted to a string of characters. These characters are placed in the output field. The output list item must be of type integer.

The data that is output to an I output field consists of one or more of the decimal digits 0 through 9. If a descriptor of the form rIw.m is used, at least m digits are output; leading zeros are output if necessary. If m is 0 and if the value of the output list item is 0, the output field consists of blanks.

If the length of the output list item is less than the width of the I output field specified, the value of the output list item is right-justified and blank-filled in the I output field.

If the length of the output list item is greater than the width of the I output field, the I output field is filled with asterisks. However, asterisks are not printed if the width of the field is not exceeded when optional characters are omitted. (Leading zeros produced as a result of a nonzero m value are not considered optional.)

Output of the sign of a value is controlled by the S, SP, and SS descriptors.

Examples

See figure 6-60 for an example of the I descriptor.

```
      .
      .
      .
      INTEGER NUM1,NUM2
      .
      .
      .
      READ(1,100) NUM1,NUM2
100   FORMAT(2I1)
      .
      .
      .
      WRITE(2,200) NUM1,NUM2
200   FORMAT('Δ',I7,I7.5)
      .
      .
      .

Input:

 5 5
 |  \
 5   5
NUM1 NUM2

Output:

NUM1 NUM2
 5    5
 |    |
ΔΔΔΔΔΔ5ΔΔ00005
```

Figure 6-60. I Descriptor Example

L Descriptor

The L descriptor formats logical data during input/output operations. See figure 6-61 for the format of the L descriptor.

rLw

r An unsigned integer constant (0 < r < 256) that is used as a repeat specification; optional. The default is 1.

w An unsigned integer constant (0 < w < 256) that specifies the width in characters of the L input field or L output field.

Figure 6-61. L Descriptor Format

Input

When the L descriptor formats data during an input operation, the value in the input field is converted to type logical and is assigned to the input list item. The input list item must be of type logical.

The L input field must contain a T or an F. The T or the F can be preceded only by a decimal point or by any number of blanks. The T or the F can be followed by any other characters. For example, the logical constants .TRUE. and .FALSE. can appear in an L input field.

Output

When the L descriptor formats data during an output operation, the value of the output list item is converted to a string of characters. These characters are placed in the output field. The number of characters output is the number of characters in the L output field. The output list item must be of type logical.

The L output field contains a T or an F in the rightmost column; all other columns are blank.

Examples

See figure 6-62 for an example of the L descriptor.

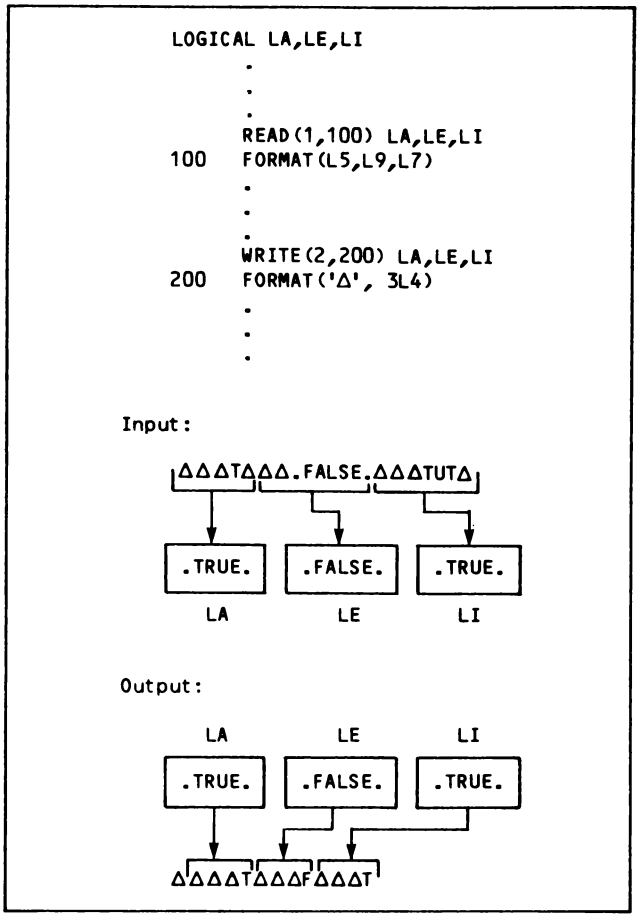


Figure 6-62. L Descriptor Example

kP

k An optionally signed integer constant used as the scale factor (k < 256). If the scale factor is to apply to a field width (d) specified on a D, E, or G edit descriptor, the value must be within the range -d<k<d+2.

Figure 6-63. P Descriptor Format

P Descriptor

The P descriptor causes a scale factor to be applied to data that is input or output using the D, E, F, or G edit descriptors. A scale factor is a number that increases or decreases a value by a power of 10. See figure 6-63 for the format of the P descriptor.

The P descriptor cannot be associated with an input/output list item.

At the beginning of execution of each input/output statement, the value of the scale factor is zero. If a P descriptor is specified, the scale factor it produces applies to all data that is formatted by D, E, F, and G descriptors that are processed after the P descriptor in the format specification.

Input

Values that are formatted by the D, E, F, and G descriptors are divided by the result of the value 10 raised to the scale factor currently in effect. However, if an exponent is specified in a value that appears in a D, E, F, or G input field, the scale factor has no effect.

Output

During an output operation, the effect of the scale factor depends on the descriptor that formats the output values:

If a D or an E descriptor formats an output value, the mantissa of the value is multiplied by the result of the value 10 raised to the scale factor. The exponent of the value is reduced by the scale factor.

If an F descriptor formats an output value, value is multiplied by the result of the value 10 raised to the scale factor.

If a G descriptor formats an output value, the effect depends on the magnitude of the data being output. If the magnitude of the data is no less than 0.1 and less than 10**d, there is no effect. If the magnitude of the data is not in this range, the effect is the same as if an E descriptor had been used to format the output value.

Examples

See figure 6-64 for an example of the P descriptor.

R Descriptor

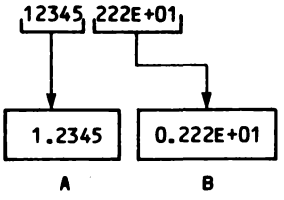
The R descriptor formats character data during input/output operations. See figure 6-65 for the format of the R descriptor.

```
rRw

r   An unsigned integer constant (0 < r < 256)
    that is used as a repeat specification;
    optional. The default is 1.

w   An unsigned integer constant (0 < w < 256)
    that specifies the width in characters of
    the R input field or R output field.
```

Figure 6-65. R Descriptor Format

```
      .
      .
      .
      REAL A,B
      .
      .
      .
      READ(1,100) A,B
100   FORMAT(2P,F5.2,E7.3)
      .
      .
      .
      WRITE(2,200) A,B
200   FORMAT(' ',-1P,F10.5,E12.2)
      .
      .
      .
```

Input:



Output:

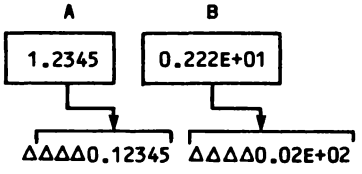


Figure 6-64. P Descriptor Example

Input

When the R descriptor formats data during an input operation, the value in the input field is assigned to the input list item. The number of characters input is the number of characters in the R input field. The input list item can be of any data type.

The data that appears in an R input field can be a string of any of the characters listed in appendix A.

If the length of the input list item is less than the width of the R input field, the rightmost characters in the R input field are assigned to the input list item.

If the length of the input list item is greater than the width of the R input field, the character value input is right-justified and binary-zero-filled in the input list item.

Output

When the R descriptor formats data during an output operation, the value of the output list item is placed in the output field. The number of characters output is the number of characters in the R output field. The output list item can be of any data type.

Any of the characters listed in appendix A can appear in an R output field. Blanks are significant characters in an R output field.

If the length of the output list item is less than the width of the R output field, the character value that is output is right-justified and character-zero-filled in the R output field.

If the length of the output list item is greater than the width of the R output field, the rightmost characters of the output list item are placed in the R output field.

Examples

See figure 6-66 for an example of the R descriptor.

S Descriptor

The S descriptor controls the printing of the plus sign during output of numeric data. The S descriptor specifies that a plus sign not be printed. The

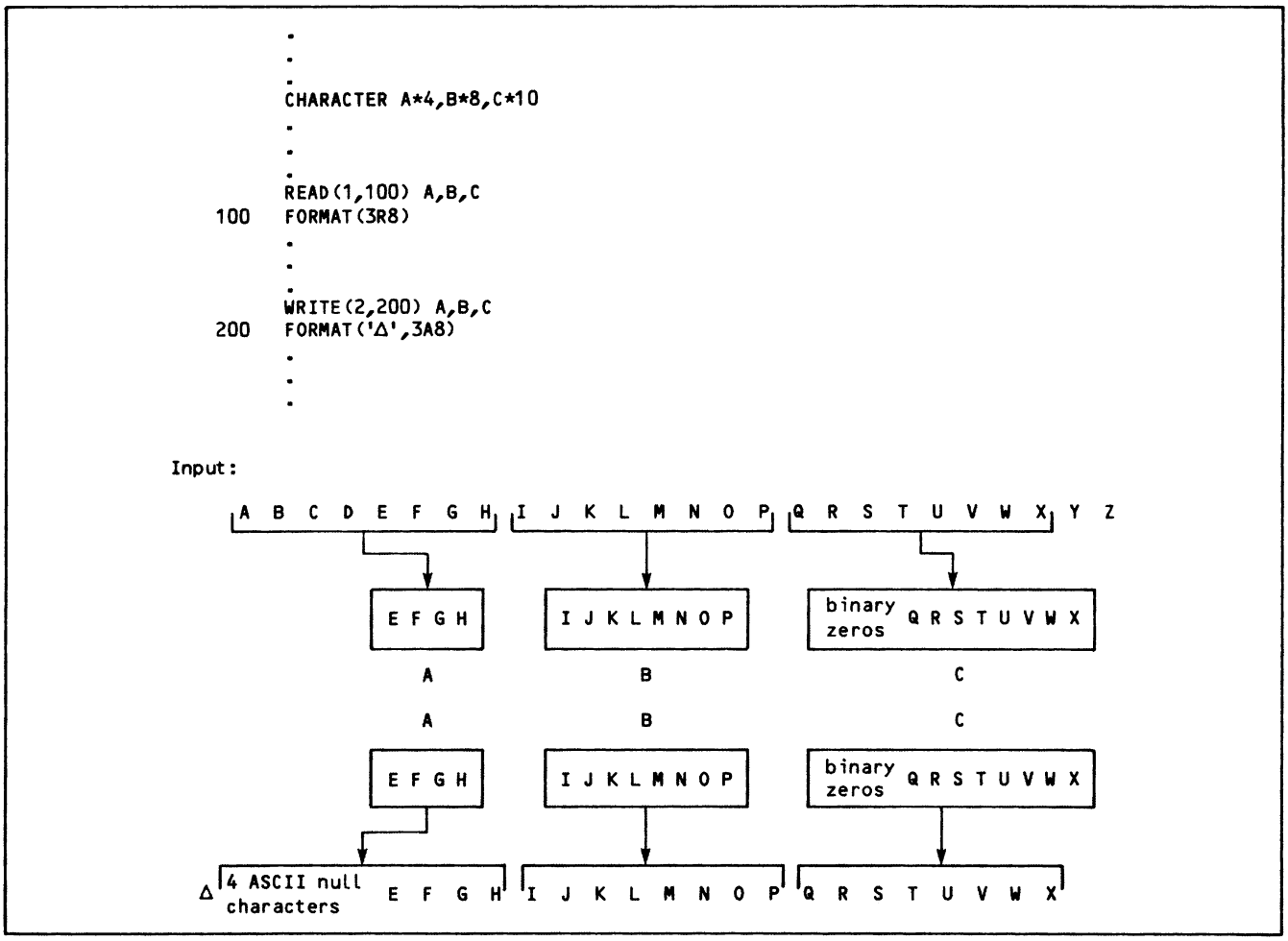


Figure 6-66. R Descriptor Example

effect of the S descriptor is identical to that of the SS descriptor. See figure 6-67 for the format of the S descriptor.

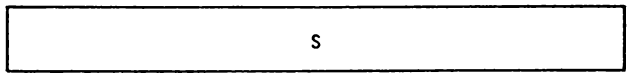


Figure 6-67. S Descriptor Format

The S descriptor cannot be associated with an input/output list item.

The S descriptor affects the sign of data that is output using the I, D, E, F, and G descriptors only. The S descriptor has no effect during the execution of an input statement.

If no S descriptor, SP descriptor, or SS descriptor is specified in a format specification, an optional plus sign is not printed. The S descriptor does not affect the sign of an exponent.

See figure 6-68 for an example of the S descriptor.

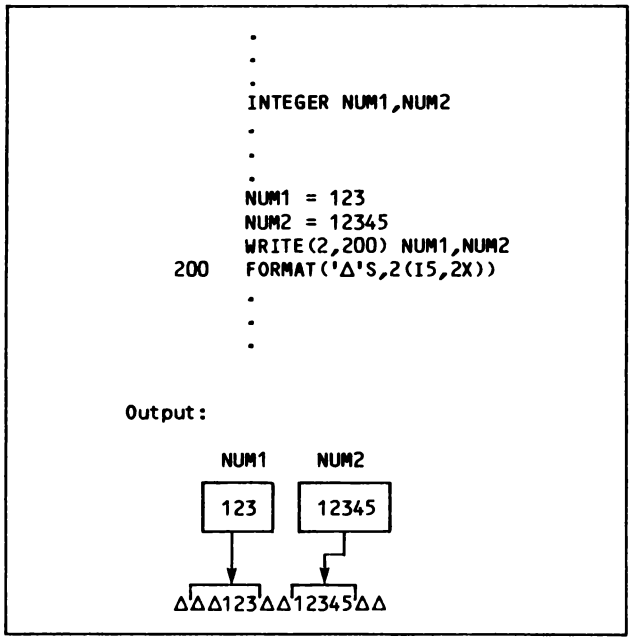


Figure 6-68. S Descriptor Example

SP Descriptor

The SP descriptor controls the printing of the plus sign during output of numeric data. The SP descriptor specifies that a plus sign must always be printed in front of positive numeric values. See figure 6-69 for the format of the SP descriptor.

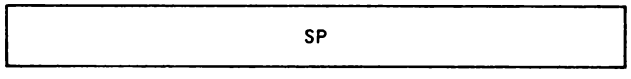


Figure 6-69. SP Descriptor Format

The SP descriptor cannot be associated with an input/output list item.

The SP descriptor affects the sign of data that is output using the I, D, E, F, and G descriptors only. The SP descriptor has no effect during the execution of an input statement.

If no S descriptor, SP descriptor, or SS descriptor is specified in a format specification, an optional plus sign is not printed. The SP descriptor does not affect the sign of an exponent.

See figure 6-70 for an example of the SP descriptor.

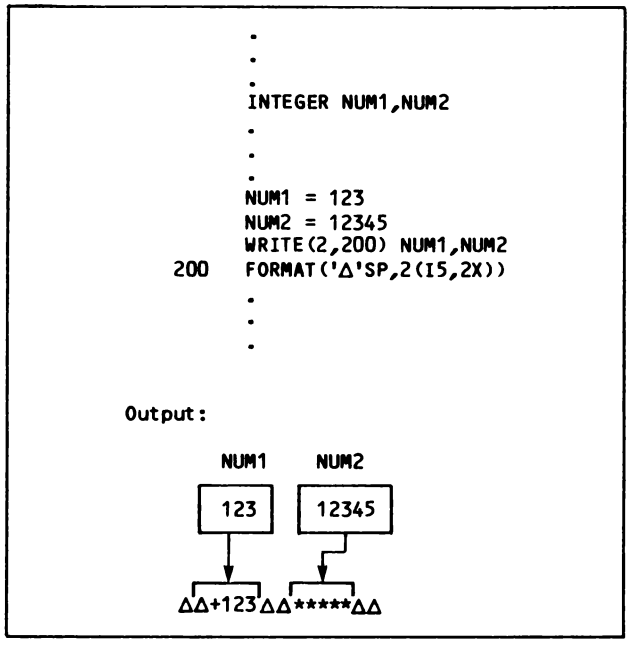


Figure 6-70. SP Descriptor Example

SS Descriptor

The SS descriptor controls the printing of the plus sign during output of numeric data. The SS descriptor specifies that a plus sign not be printed. The effect of the SS descriptor is identical to that of the S descriptor. See figure 6-71 for the format of the SS descriptor.

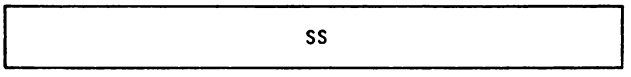


Figure 6-71. SS Descriptor Format

The SS descriptor cannot be associated with an input/output list item.

The SS descriptor affects the sign of data that is output using the I, D, E, F, and G descriptors only. The SS descriptor has no effect during the execution of an input statement.

If no S descriptor, SP descriptor, or SS descriptor is specified in a format specification, an optional plus sign is not printed. The SS descriptor does not affect the sign of an exponent.

See figure 6-72 for an example of the SS descriptor.

```
                .
                .
                .
                INTEGER NUM1,NUM2
                .
                .
                .
                NUM1 = 123
                NUM2 = 12345
                WRITE(2,200) NUM1,NUM2
200             FORMAT('Δ'SS,2(I5,2X))
                .
                .
                .

Output:

                NUM1    NUM2
               ┌─────┐ ┌─────┐
               │ 123 │ │12345│
               └──┬──┘ └──┬──┘
                  ▼       ▼
               ΔΔΔ123ΔΔ12345ΔΔ
```

Figure 6-72. SS Descriptor Example

T Descriptor

The T descriptor specifies the column from which the next character is to be input, or the column to which the next character is to be output. See figure 6-73 for the format of the T descriptor.

```
Tc

c   An unsigned integer constant (0 < c < 256)
    that specifies the column of the input
    record from which the next character is
    to be input, or the column of the output
    record to which the next character is to be
    output.
```

Figure 6-73. T Descriptor Format

The T descriptor cannot be associated with an input/output list item.

Input

When the T descriptor is used during an input operation, the next character to be read from the input record is the character that is in the column specified in the T descriptor.

Output

When the T descriptor is used during an output operation, the next character to be written to the output record is written to the column of the output record specified in the T descriptor. When the next character is written, any undefined characters to the left are set to blank; however, the T descriptor does not affect the length of the output record.

Examples

See figure 6-74 for an example of the T descriptor.

```
                .
                .
                .
                INTEGER NUM1,NUM2
                .
                .
                .
                READ(1,100) NUM1,NUM2
100             FORMAT(I3,T2,I3)
                .
                .
                .
                WRITE(2,200) NUM1,NUM2
200             FORMAT('Δ',T5,I3,T10,I3)
                .
                .
                .

Input:

                1 2 3 4
               ┌─┴─┴─┴─┐
               ▼       ▼
             ┌─────┐ ┌─────┐
             │ 123 │ │ 234 │
             └─────┘ └─────┘
              NUM1    NUM2

Output:

                NUM1    NUM2
               ┌─────┐ ┌─────┐
               │ 123 │ │ 234 │
               └──┬──┘ └──┬──┘
                  ▼       ▼
               ΔΔΔΔ123ΔΔ234
```

Figure 6-74. T Descriptor Example

TL Descriptor

The TL descriptor moves the input/output record pointer to the left. The input/output record pointer indicates the column from which the next character is input, or the column to which the next character is output. See figure 6-75 for the format of the TL descriptor.

The TL descriptor cannot be associated with an input/output list item.

TLn

n An unsigned integer constant (0 < n < 256) that specifies how many columns to the left of the current column to move the input/output record pointer.

Figure 6-75. TL Descriptor Format

Input

When the TL descriptor is used during an input operation, the input record pointer is repositioned to a column that appears to the left of the current column. The TL descriptor specifies how many columns to the left of the current column to move the pointer. The character that appears in that column is the next character to be read by the input statement.

If the TL descriptor moves the input record pointer to the left of the first column of the input record, the input record pointer is repositioned to the first column of the input record.

Output

When the TL descriptor is used during an output operation, the output record pointer is repositioned to a column that appears to the left of the current column. The TL descriptor specifies how many columns to the left of the current column to move the pointer. The next character to be output is output in that column. When the next character is written, any undefined characters to the left are set to blank; however, the TL descriptor does not affect the length of the output record.

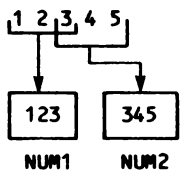
If the TL descriptor moves the output record pointer to the left of the first column of the output record, the output record pointer is repositioned to the first column of the output record.

Examples

See figure 6-76 for an example of the TL descriptor.

```
      .
      .
      .
      INTEGER NUM1,NUM2
      .
      .
      .
      READ(1,100) NUM1,NUM2
100   FORMAT(I3,TL1,I3)
      .
      .
      .
      WRITE(2,200) NUM1,NUM2
200   FORMAT('Δ',I3,'ΔOFΔTHEΔΔΔΔΔARE Δ DEFECTIVE',TL17,I3)
      .
      .
      .
```

Input:



Output:

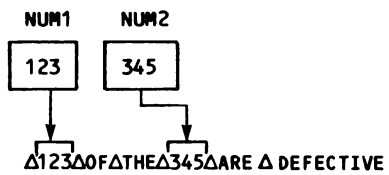


Figure 6-76. TL Descriptor Example

TR Descriptor

The TR descriptor moves the input/output record pointer to the right. The input/output record pointer indicates the column from which the next character is to be input, or the column to which the next character is to be output. See figure 6-77 for the format of the TR descriptor.

TRn

n An unsigned integer constant (0 < n < 256) that specifies how many columns to the right of the current column to move the input/output record pointer.

Figure 6-77. TR Descriptor Format

The TR descriptor cannot be associated with an input/output list item.

Input

When the TR descriptor is used during an input operation, the input record pointer is repositioned to a column that appears to the right of the current column. The TR descriptor specifies how many columns to the right of the current column to move the pointer. The character that appears in that column is the next character to be read by the input statement.

Output

When the TR descriptor is used during an output operation, the output record pointer is repositioned to a column that appears to the right of the current column. The TR descriptor specifies how many columns to the right of the current column to move the pointer. The next character to be output is output in that column. When the next character is written, any undefined characters to the left are set to blank; however, the TR descriptor does not affect the length of the output record.

Examples

See figure 6-78 for an example of the TR descriptor.

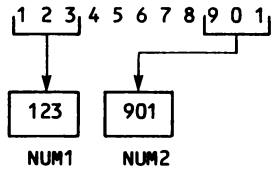
X Descriptor

The X descriptor moves the input/output record pointer to the right. The input/output record pointer indicates the column from which the next character is input, or the column to which the next character is output. See figure 6-79 for the format of the X descriptor.

The X descriptor cannot be associated with an input/output list item.

```
                .
                .
                .
                INTEGER NUM1,NUM2
                .
                .
                .
                READ(1,100) NUM1,NUM2
100             FORMAT(I3,TR5,I3)
                .
                .
                .
                WRITE(2,200) NUM1,NUM2
200             FORMAT('Δ',I3,TR7,I3)
                .
                .
                .
```

Input:



Output:

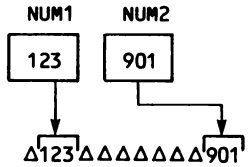


Figure 6-78. TR Descriptor Example

nX

n An unsigned integer constant (0 < n < 256) that specifies how many columns to the right of the current column to move the input/output record pointer.

Figure 6-79. X Descriptor Format

Input

When the X descriptor is used during an input operation, the input record pointer is repositioned to a column that appears to the right of the current column. The X descriptor specifies how many columns to the right of the current column to move the pointer. The character that appears in that column is the next character to be read by the input statement.

Output

When the X descriptor is used during an output operation, the output record pointer is repositioned to a column that appears to the right of the current column. The X descriptor specifies how many columns to the right of the current column to move the pointer. The next character to be output is output in that column. When the next character is written, any undefined characters to the left are set to blank; however, the X descriptor does not affect the length of the output record.

Examples

See figure 6-80 for an example of the X descriptor.

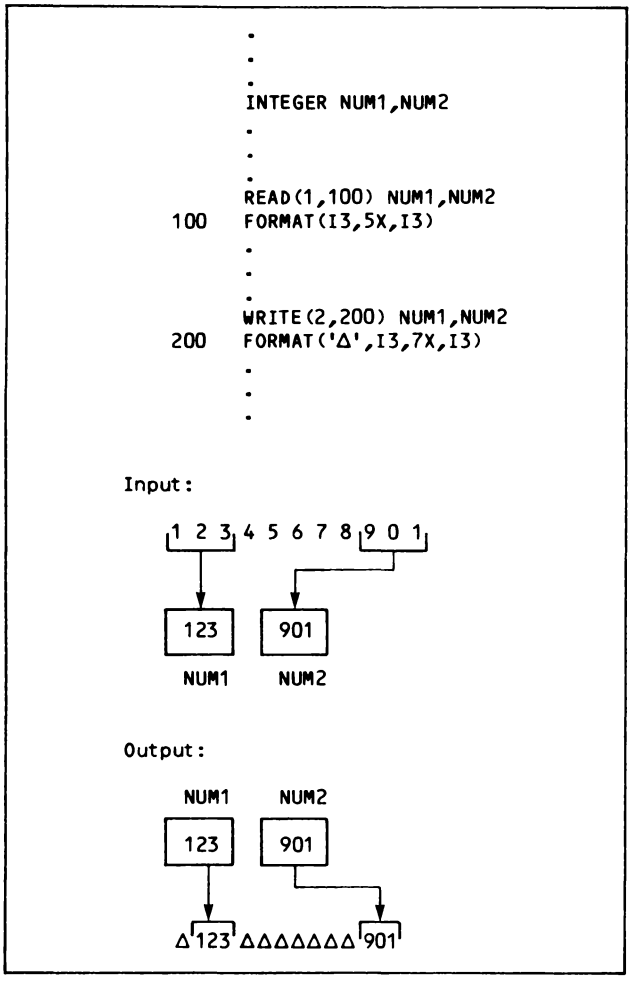


Figure 6-80. X Descriptor Example

Z Descriptor

The Z descriptor formats hexadecimal data during input/output operations. See figure 6-81 for the format of the Z descriptor.

```
rZw
   or
rZw.m
```

r An unsigned integer constant (0 < r < 256) that is used as a repeat specification; optional. The default is 1.

w An unsigned integer constant (0 < w < 256) that specifies the width in characters of the Z input field or Z output field.

m An unsigned integer constant less than or equal to w that specifies the number of digits to be output to the Z output field; m has no effect on input.

Figure 6-81. Z Descriptor Format

Input

When the Z descriptor formats data during an input operation, the value in the input field is treated as a sequence of hexadecimal digits and is assigned to the input list item. The input list item can be of any data type.

The data that appears in a Z input field must be a string of the hexadecimal digits 0 through 9 and uppercase or lowercase A through F. The hexadecimal digits correspond to the decimal values 0 through 15. Each hexadecimal digit occupies 4 bits.

Blanks that appear within a Z input field are interpreted according to any BN or BZ edit descriptors, or according to any blank specifier in effect for the file. If you do not specify a BN or BZ edit descriptor or a BLANK specifier, blanks that appear within a Z input field are ignored.

Output

When the Z descriptor formats data during an output operation, the value of the output list item is converted to a string of the hexadecimal digits 0 through 9 and A through F. The hexadecimal digits correspond to the decimal values 0 through 15. These characters are placed in the output field. The output list item can be of any data type.

If a descriptor of the form rZw.m is used, at least m digits are output; leading zeros are output if necessary. If m is 0 and if the value of the output list item is 0, the output field consists of blanks.

If the length of the output list item is less than the width of the Z output field specified, the value of the output list item is right-justified and blank-filled in the Z output field.

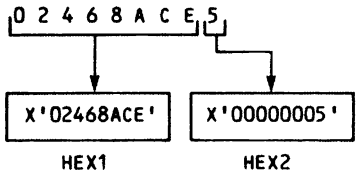
If the length of the output list item is greater than the width of the Z output field, the rightmost hexadecimal digits are placed in the Z output field.

Examples

See figure 6-82 for an example of the Z descriptor.

```
      .
      .
      .
      CHARACTER*4 HEX1,HEX2
      .
      .
      .
      READ(1,100) HEX1,HEX2
100   FORMAT(Z8,Z1)
      .
      .
      .
      WRITE(2,200) HEX1,HEX2
200   FORMAT('Δ',Z4,Z8.4)
      .
      .
      .
```

Input:



Output:

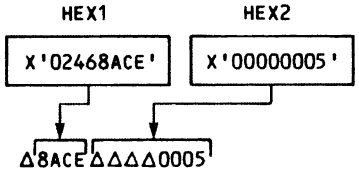


Figure 6-82. Z Descriptor Example

Apostrophe Descriptor

The apostrophe descriptor causes a string of characters to be placed in an output record. The apostrophe descriptor can be used only for output operations. See figure 6-83 for the format of the apostrophe descriptor.

```
'string'

string   A string of from 1 through 255 characters from the CYBER 200 character set.
```

Figure 6-83. Apostrophe Descriptor Format

The apostrophe descriptor cannot be associated with an input/output list item.

When the apostrophe descriptor is used for an output operation, the string of characters specified in the apostrophe descriptor is placed in the output field. The number of characters output is the number of characters in the apostrophe output field.

Any of the characters listed in appendix A can appear in an apostrophe output field. Blanks are significant characters in an apostrophe output field. An apostrophe can be represented in an apostrophe output field by specifying two consecutive apostrophes.

See figure 6-84 for an example of the apostrophe descriptor.

```
      .
      .
      .
      WRITE(2,'(''ΔPROGRAM''''S OUTPUT:'')')
      .
      .
      .

Output:

    ΔPROGRAM'SΔOUTPUT:
```

Figure 6-84. Apostrophe Descriptor Example

Slash Descriptor

The slash descriptor indicates that no more data is to be input from the current record or output to the current record during the execution of the current input/output statement. See figure 6-85 for the format of the slash descriptor.

```
/
```

Figure 6-85. Slash Descriptor Format

Input

When the slash descriptor is used during an input operation that involves a record of a sequential file, the remaining portion of the current record is skipped and the file is positioned at the beginning of the next record.

When the slash descriptor is used during an input operation that involves a record of a direct access file, the remaining portion of the current record is skipped and the record number is increased by one. The file is positioned at the beginning of that record.

Output

When a slash descriptor is used during an output operation that involves a sequential file, a new record is created. The new record is the last record of the file and the new record is the current record. Consecutive slash descriptors can cause empty records to be output to a sequential file.

When the slash descriptor is used during an output operation that involves a record of a direct access file or a record of an internal file, the record number is increased by one and the file is positioned at the beginning of that record. Consecutive slash descriptors cause records of a direct access file or records of an internal file to be filled with blanks.

Examples

See figure 6-86 for an example of the slash descriptor.

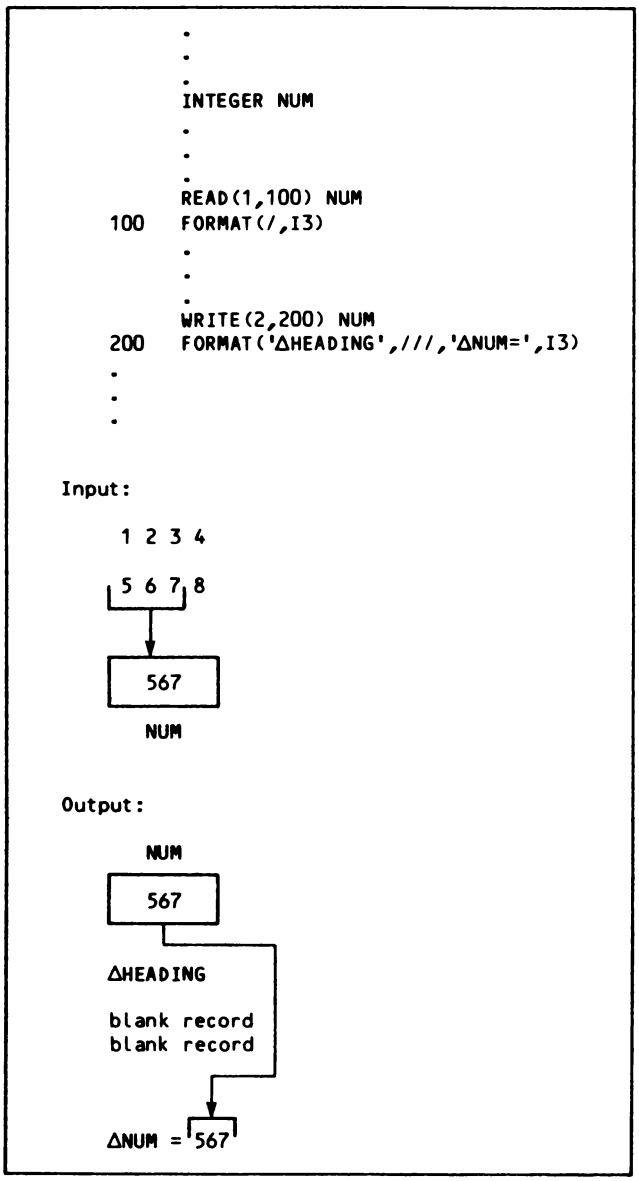


Figure 6-86. Slash Descriptor Example

Colon Descriptor

The colon descriptor terminates format control if there are no more items in the input/output list. The colon descriptor has no effect if there are more items in the input/output list. See figure 6-87 for the format of the colon descriptor.

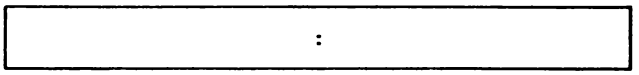


Figure 6-87. Colon Descriptor Format

See figure 6-88 for an example of the colon descriptor.

UNFORMATTED INPUT/OUTPUT STATEMENTS

An unformatted input/output statement transfers data between a sequential access external file or a direct access external file and internal storage. No formatting is performed; the data is transferred as it exists on the external file or in internal storage.

Those aspects of unformatted input/output which are unique to direct access external files are discussed separately under Direct Access Input/Output Statements. The remainder of this discussion assumes the most common file type for unformatted input/output: a sequential access external file.

The unit specified in an unformatted input/output statement must be connected for unformatted input/output; a unit can be connected by using the OPEN statement or the PROGRAM statement.

If the unit specified is not connected, the processor connects the unit to a file before the input/output statement is executed. See the description of the UNIT specifier for more information about processor-determined unit connection.

A FMT specifier must not appear in an unformatted input/output statement. The input/output list is optional in an unformatted input/output statement.

The unformatted input/output statements are:

Unformatted READ statement

Unformatted WRITE statement

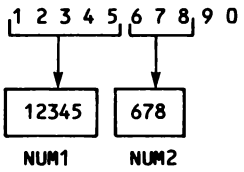
Each of these statements is described in the following paragraphs.

UNFORMATTED READ STATEMENT

The unformatted READ statement transfers data from a sequential access external file to internal storage in the format in which it exists on the file. See figure 6-89 for the format of the unformatted READ statement.

```
      .
      .
      .
      INTEGER NUM1,NUM2
      .
      .
      .
      READ(1,100) NUM1,NUM2
100   FORMAT(I5,I3,:,2I5,4I2)
      .
      .
      .
      WRITE(2,200) NUM1,NUM2
200   FORMAT('Δ',I5,:,2X,I3,:,'ΔAREΔTHEΔANSWERS.')
      .
      .
      .
```

Input:



Output:

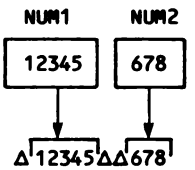


Figure 6-88. Colon Descriptor Example

```
READ (cilist) ilist

cilist   A control information list. The UNIT
         specifier must appear in cilist and
         must not be an asterisk.

         The following specifiers can also
         appear in cilist:

             END
             ERR
             IOSTAT
             REC

ilist    An input list; optional.
```

Figure 6-89. Unformatted READ Statement Format

When an unformatted READ statement is executed, one record is transferred from the file to internal storage. No formatting is performed. The file must be positioned at the beginning of an unformatted record or at the beginning of an endfile record when execution of an unformatted READ statement begins.

The number of items in the input list must be less than or equal to the number of values in the input record. The type of each value in the input record must be the same as the type of the input list item to which it corresponds. However, a complex input list item can correspond to two real values in the input record, and two real input list items can correspond to one complex value in the input record.

If the input list does not appear in the unformatted READ statement, one input record is skipped.

If an unformatted READ statement attempts to read beyond the end of a file, an execution-time error occurs. You can avoid this occurrence by specifying the END specifier or the IOSTAT specifier on the unformatted READ statement.

See figure 6-90 for an example of the unformatted READ statement. The values input by the unformatted READ statement in the example are shown. The values input by the unformatted READ statement are input from the file connected to unit 1. When an end-of-file condition is detected during execution of the unformatted READ statement, N is assigned the value -1, and control transfers to the statement labeled 10. If an input error occurs during execution of the unformatted READ statement, control transfers to the statement labeled 20 and the variable N is assigned the number of the execution-time error.

```
      .
      .
      .
      CHARACTER*8 CHARS(10)
      .
      .
      .
      DO 1 I=1,10
      READ(1,END=10,ERR=20,IOSTAT=N) CHARS(I)
1     CONTINUE
10    CALL SORT(CHARS,I-1)
      .
      .
      .
      STOP
20    CALL IOERR(N)
      .
      .
      .
```

Input is the binary representation of the characters:

```
FIRSTΔΔΔ          CHARS(1)
SECONDΔΔ          CHARS(2)
THIRDΔΔΔ          CHARS(3)
```

Figure 6-90. Unformatted READ Statement Example

UNFORMATTED WRITE STATEMENT

The unformatted WRITE statement transfers data from internal storage to a sequential access external file in the format in which it exists in internal storage. See figure 6-91 for the format of the unformatted WRITE statement.

```
WRITE (cilist) olist

cilist    A control information list. The UNIT
          specifier must appear in cilist and
          must not be an asterisk.

          The following specifiers can also
          appear in cilist:

              ERR
              IOSTAT
              REC

olist     An output list; optional.
```

Figure 6-91. Unformatted WRITE Statement Format

When an unformatted WRITE statement is executed, one record is transferred from internal storage to the file. No formatting is performed.

If the output list does not appear in the unformatted WRITE statement, a zero length record is output.

See figure 6-92 for an example of the unformatted WRITE statement. The values output by the unformatted WRITE statement in the example are shown. The values are output to the file connected to unit 2. If an output error occurs during execution of the unformatted WRITE statement, control transfers to the statement labeled 20 and the variable N is assigned the number of the execution-time error.

```
      .
      .
      .
      CHARACTER*8 CHARS/'ABCDEFGH'/
      .
      .
      .
      WRITE(2,ERR=20,IOSTAT=N) CHARS
      .
      .
      .
20    CALL IOERR (N)
      .
      .
      .
```

Output is the binary representation of the characters:

```
ABCDEFGH
```

Figure 6-92. Unformatted WRITE Statement Example

LIST-DIRECTED INPUT/OUTPUT STATEMENTS

A list-directed input/output statement transfers data between a sequential access external file and internal storage in list-directed format. List-directed formatting is described later in this section.

The unit specified in a list-directed input/output statement must be capable of formatted sequential input/output, or must be connected for formatted sequential input/output. (A unit can be connected by using the OPEN statement. Preconnection can be implicit or can be done explicitly with the PROGRAM statement or the execution control statement.)

If the unit specified is preconnected, the processor connects the unit to the file before the input/output statement is executed. See the description of the UNIT specifier for more information about processor-determined unit connection.

An asterisk in the input/output statement specifies list-directed input/output; the input/output list is optional in the statement.

The list-directed input/output statements are:

List-directed READ statement

List-directed WRITE statement

List-directed PRINT statement

List-directed PUNCH statement

Each of these statements is described in the following paragraphs.

LIST-DIRECTED READ STATEMENT

The list-directed READ statement transfers data from a sequential access external file to internal storage in list-directed format. See figure 6-93 for the format of the list-directed READ statement.

If a control information list is not specified in a list-directed READ statement, data is transferred from the unit 5HINPUT. If the input list does not appear in the list-directed READ statement, one input record is skipped.

If a list-directed READ statement attempts to read beyond the end of a file, an execution-time error occurs. You can prevent this error by specifying the END specifier or the IOSTAT specifier on the list-directed READ statement.

See figure 6-94 for examples of the list-directed READ statement. The values input by the list-directed READ statements in the example are shown. The values input by the first list-directed READ statement are input from the file connected to unit 1. When an end-of-file condition is detected during execution of the first list-directed READ statement, control transfers to the statement labeled 10, and N is assigned the value -1. If an input error occurs during execution of the first list-directed READ statement, control transfers to the statement labeled 20 and the variable N is assigned the number of the execution-time error.

```
READ(cilist) ilist
    or
READ *,ilist


cilist    A control information list. The
          following specifiers must appear in
          cilist:

              UNIT
              FMT (must be an asterisk)

          The following specifiers can also
          appear in cilist:

              END
              ERR
              IOSTAT

ilist     An input list; optional.


If the second form of the list-directed READ
statement is used and ilist is not specified,
the comma separating the asterisk from ilist
must not appear.
```

Figure 6-93. List-Directed READ Statement Format

```
          .
          .
          .
          I = 0
    1     I = I + 1
          READ(1,*,END=10,ERR=20,IOSTAT=N) A,B
          READ*, C,D
          AVG(I) = (A+B+C+D)/4
          GOTO 1
    10    CALL PLOT(AVG,I-1)
          .
          .
          .
          STOP
    20    CALL IOERR(N)
          .
          .
          .


Input:

    10.00ΔΔΔ20.0          (On unit 1)
    70.0ΔΔ50.0            (On file INPUT)
```

Figure 6-94. List-Directed READ Statement Examples

The values input by the second list-directed READ statement are input from the unit 5HINPUT.

LIST-DIRECTED WRITE STATEMENT

The list-directed WRITE statement transfers data from internal storage to a sequential access external file in list-directed format. See figure 6-95 for the format of the list-directed WRITE statement.

```
WRITE(cilist) olist

cilist    A control information list.  The
          following specifiers must appear in
          cilist:

              UNIT
              FMT (must be an asterisk)

          The following specifiers can also
          appear in cilist:

              ERR
              IOSTAT

olist     An output list; optional.
```

Figure 6-95. List-Directed WRITE Statement Format

If the output list does not appear in the list-directed WRITE statement, a record consisting of a single blank character is written.

See figure 6-96 for examples of the list-directed WRITE statement. The value output by the list-directed WRITE statements in the example are shown. The values are output to the file connected to unit 2. If an output error occurs during execution of the first list-directed WRITE statement, control transfers to the statement labeled 20 and the variable N is assigned the number of the execution-time error.

LIST-DIRECTED PRINT STATEMENT

The list-directed PRINT statement transfers data from internal storage to the unit 6HOUTPUT in list-directed format. See figure 6-97 for the format of the list-directed PRINT statement.

```
PRINT *, olist

olist     An output list; optional

If olist is not specified, the comma separating
the asterisk from olist must not appear.
```

Figure 6-97. List-Directed PRINT Statement Format

If the output list does not appear in the list-directed PRINT statement, a record consisting of a single blank is written.

See figure 6-98 for an example of the list-directed PRINT statement. The values output by the list-directed PRINT statement in the example are shown. The values are output to the unit 6HOUTPUT.

```
              .
              .
              .
              I=5
              PRINT *, 'THE Δ ANSWER Δ IS Δ ',I
              .
              .
              .


     Output:

          ΔTHE Δ ANSWER Δ IS Δ 5
```

Figure 6-98. List-Directed PRINT Statement Example

```
                    .
                    .
                    .
                    INTEGER J(4)
                    COMPLEX Z(2)
                    DOUBLE PRECISION Q
                    DATA J,Z,Q /1,-2,3,-4,(7.,-1.),(-3.,2.),1.D-5/
                    WRITE(2,*,ERR=20,IOSTAT=n) J
                    WRITE(2,*) J
                    WRITE(2,*) Z(1),Q
                    .
                    .
                    .
                    STOP
               20   CALL IOERR (N)
                    .
                    .
                    .


               Output:

                    Δ1 Δ-2 Δ3 Δ-4

                      (7.0000000000000,-1.0000000000000)  1.0000000000000000000000000000E-05
```

Figure 6-96. List-Directed WRITE Statement Examples

LIST-DIRECTED PUNCH STATEMENT

The list-directed PUNCH statement transfers data from internal storage to the unit 5HPUNCH in list-directed format. See figure 6-99 for the format of the list-directed PUNCH statement.

```
PUNCH *, olist

olist    An output list; optional

If olist is not specified, the comma separating
the asterisk from olist must not appear.
```

Figure 6-99. List-Directed PUNCH Statement Format

If the output list does not appear in the list-directed PUNCH statement, a record consisting of a single blank character is punched.

See figure 6-100 for an example of the list-directed PUNCH statement. The values output by the list-directed PUNCH statement in the example are shown. The values are output to the file called PUNCH.

```
          .
          .
          .
          I = 5
          PUNCH *, 'THE Δ ANSWER Δ IS Δ ',I
          .
          .
          .

Output:

     ΔTHE Δ ANSWER Δ IS Δ 5
```

Figure 6-100. List-Directed PUNCH Statement Example

LIST-DIRECTED FORMATTING

List-directed input/output statements transfer data between sequential access external files and internal storage in list-directed format. List-directed format is a predefined format specification.

List-directed formatting for input and output is described in the following paragraphs.

LIST-DIRECTED INPUT FORMATTING

A list-directed input statement transfers data from a sequential access external file to internal storage. The data read must be in list-directed input format. A list-directed input record consists of zero or more blanks followed by a list of input fields separated by any of the following separators:

One or more contiguous blanks

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

A slash optionally preceded and optionally followed by one or more contiguous blanks (the slash separator terminates the input operation)

When a list-directed input statement is executed, the value in the first input field of a record is assigned to the first input list item in the input statement, the value in the next field is assigned to the second input list item, and so on.

If a list-directed input statement follows a list-directed input statement that terminated in the middle of a record, the second input statement begins inputting the values from the first input field of the next record.

Execution of a list-directed input statement terminates when all of the items in the input list have been assigned values. You can also terminate execution of a list-directed input statement by using the slash as a separator in the input data.

If all of the input fields of a record are input before execution of the list-directed input statement is terminated, the input statement continues inputting values from the input fields of subsequent records.

The value that appears in an input field is converted to the type of the input list item to which the value is assigned. The format of a value that appears in an input field depends on the type of the input list item to which it is assigned:

When the input list item is of type integer, the value in the corresponding input field must have the same format as an I input field.

When the input list item is of type real, double-precision, or half-precision, the value in the corresponding input field must have the same format as an F input field.

When the input list item is of type complex, the value in the corresponding input field must have the same format as a complex constant. Both the real part and imaginary part of the complex constant can be preceded or followed by blanks. The end of the record can occur between the real part and the comma, or between the comma and the imaginary part.

When the input list item is of type logical, the value in the corresponding input field must have the same format as an L input field; however, slashes and commas are not permitted as optional characters in the input field.

When the input list item is of type character, the value in the corresponding input field must be a string of one or more characters enclosed in apostrophes. The characters must be from the CYBER 200 character set. An apostrophe that appears in the character string must be represented as two consecutive apostrophes. The end of the record can occur in a character string without affecting the characters in the string.

When the input list item is of type bit, the value in the corresponding input field must have the same format as a B input field.

If several adjacent input fields contain the same value, you can use a repeat specification rather than explicitly specifying each input field. A repeat specification is an unsigned integer constant greater than or equal to 1 followed by an asterisk that precedes the input field to be repeated.

For input list items of type character, if the length of the input list item is less than the length of the character value in the input field, the leftmost characters in the input field are assigned to the input list item. If the length of the input list item is greater than the width of the character value in the input field, the character value input is left-justified and blank-filled in the input list item.

A null value can be assigned to an input list item by specifying two consecutive separators in the input data. When a null value is assigned to an input list item, the value of the input list item is not changed. A null value can be assigned to an input list item of type complex; however, a null value must be assigned to both the real part and the imaginary part of the complex input list item.

If an input operation is terminated by using the slash as a separator in the input data, the values of any input list items that have not been assigned values by the input operation are not changed.

LIST-DIRECTED OUTPUT FORMATTING

A list-directed output statement transfers data from internal storage to a sequential access external file. The data is output in list-directed output format. A list-directed output record consists of a blank followed by a list of output fields separated by blanks.

When a list-directed output statement is executed, the values of the output list items are converted to character strings and placed in the output file. A blank is inserted between each output field, except before and after character values. Each list-directed output statement outputs a new record.

Execution of a list-directed output statement terminates when the values of all of the output list items have been output.

If a list-directed output statement outputs a line that is longer than 137 characters, the line is continued on subsequent output lines. Lines are broken at separators; however, a line can be broken between the real part and the imaginary part of a complex output value, and a line can be broken in a character output value.

A blank is always inserted at the beginning of each output line; the blank is provided for carriage control in case the file is printed on a line printer.

The format of a value that is output to an output field depends on the type of the output list item being output:

When the output list item is of type integer, the value written to the corresponding output field has the same format as an I16 output field; however, leading blanks are removed.

When the output list item is of type real, double-precision, or half-precision, the value written to the corresponding output field has the same format as an F output field or an E output field, depending on the magnitude of the value. If the magnitude of the value is greater than or equal to 10**-3 and less than or equal to 10**d1, the output field has the same format as a 0PF output field: d1+1 digits are output.

If the magnitude of the value is less than 10**-3 or greater than or equal to 10**d1, the output field has the same format as a 1PEw.dEe output field.

The values of d1, w, d, and e depend on the data type of the output list item. See table 6-4 for the values.

When the output list item is of type complex, the value written to the corresponding output field has the same format as a complex constant. No blanks appear in the constant unless the end of the record occurs in the constant. The end of the record can occur between the real part and the comma, or between the comma and the imaginary part.

When the output list item is of type logical, the value written to the corresponding output field has the same format as an L output field.

When the output list item is of type character, the value written to the corresponding output field is a string of one or more characters. The string is not enclosed in apostrophes. The characters are from the CYBER 200 character set. An apostrophe that appears in the character string is represented as one apostrophe. The end of the record can occur in a character string without affecting the characters in the string; however, a blank is always output as the first character in a record in order to provide carriage control.

When the output list item is of type bit, the value written to the corresponding output field has the same format as a B output field.

Null values cannot be output. The slash cannot be output as a separator.

TABLE 6-4. VALUES FOR d1, w, d, AND e

Output List Item Type	d1	w	d	e
Real	13	22	13	4
Double-Precision	27	36	27	4
Half-Precision	6	13	6	2

NAMELIST INPUT/OUTPUT STATEMENTS

A namelist input/output statement transfers data between a sequential access external file and internal storage in namelist format. Namelist formatting is described later in this section.

The unit specified in a namelist input/output statement must be preconnected to a file capable of formatted sequential input/output, or must be connected for formatted input/output. (A unit can be connected by using the OPEN statement. Preconnection can be implicit or can be done explicitly with the PROGRAM statement or the execution control statement.

If the unit specified is preconnected, the processor connects the unit to the file before the input/output statement is executed. See the description of the UNIT specifier for more information about processor-determined unit connection.

A FMT specifier in a namelist input/output statement must specify a namelist group name. A namelist input/output statement must not contain an input/output list.

The NAMELIST statement defines a namelist group. The namelist input/output statements are:

Namelist READ statement

Namelist WRITE statement

Namelist PRINT statement

Namelist PUNCH statement

The NAMELIST statement and each of the namelist input/output statements are described in the following paragraphs.

NAMELIST STATEMENT

The NAMELIST statement is a nonexecutable statement that defines one or more namelist groups. A namelist group is an input/output list that is identified by a symbolic name; the symbolic name is called the namelist group name. See figure 6-101 for the format of the NAMELIST statement.

NAMELIST /$grpname_1$/ $niolist_1$.../$grpname_n$/$niolist_n$

$grpname_1$	A symbolic name that is used as the namelist group name
$niolist_i$	A list of one or more variable names and array names separated by commas that are used as the namelist group

Figure 6-101. NAMELIST Statement Format

A NAMELIST statement that defines a namelist group must appear in each program unit that performs namelist input/output using that namelist group.

See figure 6-102 for an example of the NAMELIST statement. The NAMELIST statement in the example defines two namelist groups: the first is called GROUP1 and consists of the items A and B, and the second is called GROUP2 and consists of the item D.

```
      .
      .
      .
      REAL A
      COMPLEX B
      DOUBLE PRECISION D
      NAMELIST /GROUP1/A,B/GROUP2/D
      .
      .
      .
      READ(1, GROUP1)
      READ(1, GROUP2)
      .
      .
      .
```

Figure 6-102. NAMELIST Statement Example

NAMELIST READ STATEMENT

The namelist READ statement transfers data from a sequential access external file to internal storage in namelist format. See figure 6-103 for the format of the namelist READ statement.

```
READ (cilist)
    or
READ grpname
```

cilist — A control information list. The following specifiers must appear in cilist:

- UNIT
- FMT (must be a namelist group name)

The following specifiers can also appear in cilist:

- END
- ERR
- IOSTAT

grpname — A namelist group name.

Figure 6-103. Namelist READ Statement Format

When a namelist READ statement is executed, one namelist group is transferred from the file to the items in the namelist group. The values read are converted to the type of the items to which they are assigned.

If a control information list is not specified in a namelist READ statement, data is transferred from the unit 5HINPUT.

If a namelist READ statement attempts to read beyond the end of a file, an execution-time error occurs. You can prevent this error by specifying the END specifier or the IOSTAT specifier on the namelist READ statement.

See figure 6-104 for an example of the namelist READ statement. The values input by the namelist READ statements in the example are shown. The values input by the first namelist READ statement are input from the file connected to unit 1. When an end-of-file condition is detected during execution of the first namelist READ statement, control transfers to the statement labeled 10, and N is assigned the value -1. If an input error occurs during execution of the first namelist READ statement, control transfers to the statement labeled 20 and the variable N is assigned the number of the execution-time error.

The values input by the second namelist READ statement are input from the unit 5HINPUT.

NAMELIST WRITE STATEMENT

The namelist WRITE statement transfers data from internal storage to a sequential access external file in namelist format. See figure 6-105 for the format of the namelist WRITE statement.

See figure 6-106 for an example of the namelist WRITE statement. The values output by the namelist WRITE statement in the example are shown. The values are output to the file connected to unit 2. If an output error occurs during execution of the namelist WRITE statement, control transfers to the statement labeled 20 and the variable N is assigned the number of the execution-time error.

NAMELIST PRINT STATEMENT

The namelist PRINT statement transfers data from internal storage to the unit 6HOUTPUT in namelist format. See figure 6-107 for the format of the namelist PRINT statement.

See figure 6-108 for an example of the namelist PRINT statement. The values output by the namelist PRINT statement in the example are shown. The values are output to the unit 6HOUTPUT.

```
          .
          .
          .
          NAMELIST /GROUP1/A,B/GROUP2/C,D
          .
          .
          .
          I = 0
    1     I = I + 1
          READ(1,GROUP1,END=10,ERR=20,IOSTAT=N)
          READ GROUP2
          AVG(I) = (A+B+C+D)/4
          GOTO 1
    10    CALL PLOT(AVG,I-1)
          .
          .
          .
          STOP
    20    CALL IOERR(N)
          .
          .
          .

Input:

    Δ&GROUP1
    ΔA=10.0,B=20.0
    Δ&END
    Δ&GROUP2
    ΔC=50.0,D=70.0
    Δ&END
```

Figure 6-104. Namelist READ Statement Example

```
WRITE (cilist)

cilist    A control information list. The
          following specifiers must appear in
          cilist:

              UNIT
              FMT (must be a namelist
                   group name)

          The following specifiers can also
          appear in cilist:

              ERR
              IOSTAT
```

Figure 6-105. Namelist WRITE Statement Format

```
                .
                .
                .
                REAL A
                INTEGER I
                COMPLEX C
                NAMELIST /GROUP/A,I,C
                .
                .
                .
                A = 5.7
                I = 12
                C = (1.0,0.0)
                WRITE(2,GROUP,ERR=20,IOSTAT=N)
                .
                .
                .
           20   CALL IOERR(N)
                .
                .
                .

      Output:

          Δ&GROUP
          ΔA=5.7000000000000,I=12,C=(1.0000000000000,0.0000000000000E+00)
          Δ&END
```

Figure 6-106. Namelist WRITE Statement Example

```
                PRINT grpname

                grpname   A namelist group name
```

Figure 6-107. Namelist PRINT Statement Format

```
                .
                .
                .
                REAL A
                INTEGER I
                COMPLEX C
                NAMELIST /GROUP/A,I,C
                .
                .
                .
                A = 5.7
                I = 12
                C = (1.0,0.0)
                PRINT GROUP
                .
                .
                .

      Output:

          Δ&GROUP
          ΔA=5.7000000000000,I=12,C=(1.0000000000000,0.0000000000000E+00)
          Δ&END
```

Figure 6-108. Namelist PRINT Statement Example

NAMELIST PUNCH STATEMENT

The namelist PUNCH statement transfers data from internal storage to the unit 5HPUNCH in namelist format. See figure 6-109 for the format of the namelist PUNCH statement.

```
PUNCH grpname

grpname   A namelist group name
```

Figure 6-109. Namelist PUNCH Statement Format

See figure 6-110 for an example of the namelist PUNCH statement. The values output by the namelist PUNCH statement in the example are shown. The values are output to the unit 5HPUNCH.

NAMELIST FORMATTING

Namelist input/output statements transfer data between sequential access external files and internal storage in namelist format. Namelist format is a predefined format specification.

Namelist formatting for input and output is described in the following paragraphs.

NAMELIST INPUT FORMATTING

A namelist input statement transfers data from a sequential access external file to internal storage. The data read must be in namelist input format. See figure 6-111 for the format of namelist input.

Δ&grpname sep d_1 sep...sepd_n&END

sep	A separator consisting of a comma, blank, or record boundary, surrounded by zero or more blanks or record boundaries.
grpname	A namelist group name
d_i	A data item of one of the following forms: var=const aname=const sep...sep const aname=r*const sep...sep r*const aname(sub)=const

where:

var	A simple variable
aname	An array name
sub	An integer constant expression
const	A constant
r	An integer constant repeat specification

Each record must have a blank in the first column.

Figure 6-111. Namelist Input Format

```
                    .
                    .
                    .
                    REAL A
                    INTEGER I
                    COMPLEX C
                    NAMELIST /GROUP/A,I,C
                    .
                    .
                    .
                    A = 5.7
                    I = 12
                    C = (1.0,0.0)
                    PUNCH GROUP
                    .
                    .
                    .

Output:

    Δ&GROUP
    ΔA=5.7000000000000,I=12,C=(1.0000000000000,0.0000000000000E+00)
    Δ&END
```

Figure 6-110. Namelist PUNCH Statement Example

When a namelist input statement is executed, the value specified for each item of the namelist input group in the input data is assigned to the item in the namelist input group.

Execution of a namelist input statement terminates when all of the values in the namelist input data are assigned to the corresponding item in the namelist group.

The value specified for each item in the namelist input group is converted to the type of the input list item to which the value is assigned. A bit, logical, character, or complex constant must be of the same type as the corresponding input list item.

For input list items of type character, if the length of the namelist group item is less than the length of the value specified for that item in the input data, the leftmost characters in the input value are assigned to the namelist group item. If the length of the namelist group item is greater than the length of the value specified for that item in the input data, the character value input is left-justified and blank-filled in the namelist group item.

An integer, half-precision, real, or double-precision constant can be used for an integer, half-precision, real, or double-precision input list item. The forms for integer, half-precision, real, and double-precision constants are described for list-directed input earlier in this section under the heading, List-Directed Input Formatting.

Use of the BLANK specifier in an OPEN statement has no effect on namelist editing.

If a value is not specified for an item of the namelist group, the value of that item is not changed.

NAMELIST OUTPUT FORMATTING

A namelist output statement transfers data from internal storage to a sequential access external file. The data is output in namelist output format. See figure 6-112 for the format of namelist output.

When a namelist output statement is executed, the value of each namelist group item is converted to a character string and transferred to the output file in namelist format.

Execution of a namelist output statement terminates when the values of all namelist group items are output to the file.

BUFFER INPUT/OUTPUT STATEMENTS

A buffer input/output statement transfers data between a sequential access external file and a buffer area in internal storage. The buffer input/output statements are provided for compatibility with other FORTRAN compilers and are not intended for use with new programs. See appendix E for a description of the buffer input/output statements.

```
Δ&grpname
Δdata1
   .
   .
   .
Δdatan
Δ&END
```

grpname — A namelist group name

$data_i$ — A data item of one of the following forms:

```
var=const
aname=const1 Δ ... Δ constn
```

where:

var — A simple variable

aname — An array name

const — A constant

No output record can be longer than 137 characters. If necessary, the output record is split into multiple records, each no longer than 137 characters. The split occurs at the end of a constant, after the comma within a complex constant, or within a character constant. A blank is inserted in the first column of each record.

Figure 6-112. Namelist Output Format

DIRECT ACCESS INPUT/OUTPUT

To perform input and output on a direct access file, you must satisfy four conditions: open the file as a direct access file; declare the proper record length; include the REC specifier in the input/output statement; and leave out the END specification in the input/output statement.

The OPEN statement must be used to connect a direct access file; the OPEN statement is described later in this section.

When the slash descriptor is used during a formatted input operation that involves a record of a direct access file, the remaining portion of the current record is skipped and the record number is increased by one. The file is positioned at the beginning of that record.

When the slash descriptor is used during a formatted output operation that involves a record of a direct access file, the remainder of the record is blank-filled, the record number is increased by one, and the file is positioned at the beginning of that record. Consecutive slash descriptors cause records of a direct access file to be filled with blanks.

When a formatted output statement causes data to be output to a direct access file, the number of characters output must not exceed the record length of the file. If the number of characters output to a direct access file is less than the length of a record of the file, the remaining portion of the record is filled with blanks.

When an unformatted output statement causes data to be output to a direct access file, the number of words output must not exceed the length of a record of the file. If the number of words output to a direct access file is less than the length of a record of the file, the remaining portion of the record is undefined.

See figure 6-113 for an example of a formatted direct access READ statement that reads a direct access file. The values input by the READ statement in the example are shown. The values input by the READ statement are input from the file connected to unit 1. If an input error occurs during execution of the READ statement, control transfers to the statement labeled 20 and the variable N is assigned the number of the execution-time error.

The slash descriptor in the format specification causes the record number to be incremented by 1.

See figure 6-114 for an example of an unformatted READ statement that reads a direct access file. The values input by the READ statement in the example are shown. The values input by the READ statement are input from the file connected to unit 1. If an input error occurs during execution of the READ statement, control transfers to the statement labeled 20 and the variable N is assigned the number of the execution-time error.

```
      .
      .
      .
      READ(1,100,REC=2,ERR=20,IOSTAT=N) I,J
100   FORMAT(I4,/I4)
      .
      .
      .
20    CALL IOERR(N)
      .
      .
      .
```

Input:

Record Number	Record		
1	1884		
2	1885	→ 1885	I
3	1886	→ 1886	J
4	1887		

Figure 6-113. Formatted Direct Access Input/Output Example

INTERNAL FILE INPUT/OUTPUT

Formatted input/output statements can perform input and output on internal files. In order to perform input and output on an internal file, you must use a character variable, a character array, or a substring as the UNIT specifier in the control information list of the formatted input/output statement.

When an input operation is performed on an internal file, data is transferred from consecutive locations of the internal file beginning at the first character position; the input values are stored in the items in the input list. Formatting is performed according to the format specification you provide.

When the slash descriptor is used during an input operation that involves a record of an internal file, the remaining portion of the current record

```
      .
      .
      .
      CHARACTER CHAR*8
      .
      .
      .
      READ(1,REC=2,ERR=20,IOSTAT=N) CHAR
      .
      .
      .
20    CALL IOERR(N)
      .
      .
      .
```

Input:

Record Number	Record (Binary Representation)		
1	FIRSTΔΔΔ		
2	SECONDΔΔ	→ SECONDΔΔ	CHAR
3	THIRDΔΔΔ		

Figure 6-114. Unformatted Direct Access Input/Output Example

is skipped and the record number is increased by one. The file is positioned at the beginning of that record.

When an output operation is performed on an internal file, the values of the output list items are converted to character strings and transferred to the internal file. The character string is formatted according to the format specification you provide.

When the slash descriptor is used during an output operation that involves a record of an internal file, the remainder of the current record is blank-filled, the record number is increased by one, and the file is positioned at the beginning of that record. Consecutive slash descriptors cause records of an internal file to be filled with blanks.

The length of a character string output to a record of an internal file must not be greater than the length of the file. If the length of the character string output to a record of an internal file is less than the length of the record, the remaining portion of the record is filled with blanks.

See figure 6-115 for an example of a formatted READ statement that reads an internal file. The values input by the formatted internal file READ statement in the example are shown. The values input by the formatted internal file READ statement are input from the internal file CHAR.

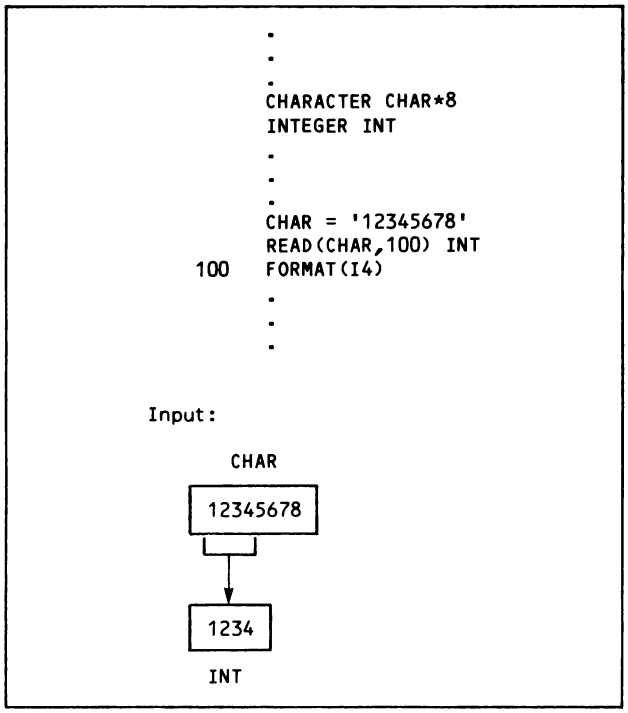


Figure 6-115. Internal File Input/Output Example

EXTENDED INTERNAL FILE INPUT/OUTPUT STATEMENTS

An extended internal file input/output statement transfers data between an extended internal file and internal storage in a format that you specify. Format specification for extended internal file input/output statements is the same as for formatted input/output statements.

Extended internal file input/output statements can also perform input and output operations on internal files.

A FMT specifier must appear in an extended internal file input/output statement. The input/output list is optional.

The extended internal file input/output statements are:

DECODE statement

ENCODE statement

Each of these statements is described in the following paragraphs.

DECODE STATEMENT

The DECODE statement transfers data from an extended internal file or from an internal file to internal storage in the format you specify. See figure 6-116 for the format of the DECODE statement.

```
DECODE (no-chars,fid,uid) iolist
```

no-chars	An unsigned integer constant or an integer variable greater than or equal to 1; no-chars specifies the number of characters in each record of the extended internal file or the internal file.
fid	A format identifier; fid must not specify list-directed or namelist formatting.
uid	A unit identifier; uid must be an extended internal file identifier or an internal file identifier.
iolist	An input/output list; optional.

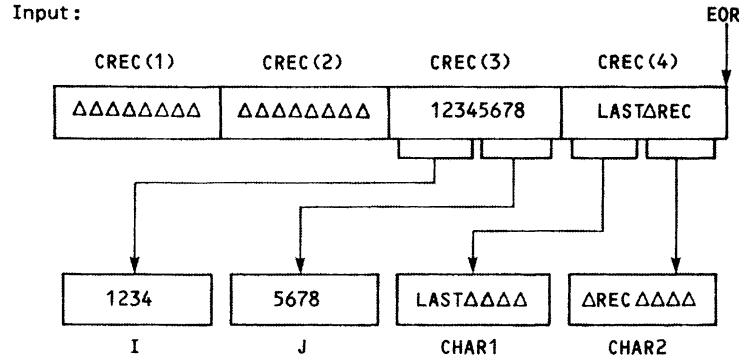
Figure 6-116. DECODE Statement Format

The DECODE statement is analogous to the formatted READ statement.

The number of words in the input list and the edit descriptors specified in the associated format specification must correspond to the format of the input record. An input record is skipped for each slash descriptor that appears in the associated format specification.

See figure 6-117 for an example of the DECODE statement. The values input by the DECODE statement in the example are shown. The values input by the DECODE statement are input from the internal file CREC.

```
      .
      .
      .
      CHARACTER*8 CREC(4),CHAR1,CHAR2
      INTEGER I,J
      DATA CREC/2*'          ','12345678','LAST REC'/
      .
      .
      .
      DECODE (32,100,CREC) I,J,CHAR1,CHAR2
100   FORMAT(16X,2I4,2A4)
      .
      .
      .
```



I and J contain internal integer values.

Figure 6-117. DECODE Statement Example

ENCODE STATEMENT

The ENCODE statement transfers data from internal storage to an extended internal file or to an internal file in the format you specify. See figure 6-118 for the format of the ENCODE statement.

The ENCODE statement is analogous to the formatted WRITE statement.

See figure 6-119 for an example of the ENCODE statement. The values output by the ENCODE statement are output to the internal file CREC.

```
ENCODE (no-chars,fid,uid) iolist
```

no-chars	An unsigned integer constant or an integer variable greater than or equal to 1; no-chars specifies the number of characters in each record of the extended internal file or the internal file.
fid	A format identifier; fid must not specify list-directed or namelist formatting.
uid	A unit identifier; uid must be an extended internal file identifier or an internal file identifier.
iolist	An input/output list; optional.

Figure 6-118. ENCODE Statement Format

CONCURRENT INPUT/OUTPUT STATEMENTS

The concurrent input/output statements cause input/output operations to be initiated, then return control to the program. The concurrent input/output statements are written as calls to predefined subroutines. See section 11 for a description of the concurrent input/output subroutines.

```
      .
      .
      .
      CHARACTER*8 CREC(4),CHAR1,CHAR2
      INTEGER I,J
      DATA CHAR1,CHAR2/'THEΔBEGI','NNINGΔΔΔ'/
      DATA I,J/2*37/
      .
      .
      .
      ENCODE (32,200,CREC) CHAR1,CHAR2,I,J
200   FORMAT('THISΔISΔ',2A8,2I2)
      .
      .
      .
```

Output:

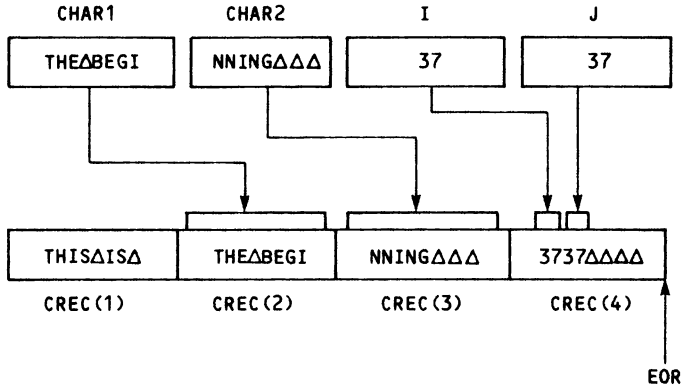


Figure 6-119. ENCODE Statement Example

DIRECT CALLS TO SIL ROUTINES

You can cause control to transfer to a System Interface Language (SIL) subroutine by using a direct call to the subroutine. See section 13 for a description of direct calls to SIL routines.

AUXILIARY INPUT/OUTPUT STATEMENTS

Auxiliary input/output statements connect files to units, disconnect files from units, and inquire about the properties of a file or unit.

The auxiliary input/output statements are:

OPEN statement

CLOSE statement

INQUIRE statement

Each of these statements is described in the following paragraphs.

OPEN STATEMENT

The OPEN statement connects an existing file to a unit, creates a file that is preconnected to a unit, creates and connects a new file to a unit, or changes certain of the properties of the connection of a file and unit. See figure 6-120 for the format of the OPEN statement.

```
OPEN (cilist)

cilist    A control information list. The UNIT
          specifier must appear in cilist and
          must not be an asterisk.

          The following specifiers can also
          appear in cilist:

              ACCESS
              BLANK
              BUFS
              ERR
              FILE
              FORM
              IOSTAT
              RECL (must be specified if
                    ACCESS='DIRECT')
              STATUS
```

Figure 6-120. OPEN Statement Format

An OPEN statement can appear in any program unit. The file connected by an OPEN statement can be referenced in any program unit.

If the file already exists, the record structure already defined for the file is used; if the FORTRAN program creates a sequential access file, the control word delimited (W) record type is used; if the FORTRAN program creates a direct access file, the fixed-length (F) record type is used.

If a unit is connected to a file that exists, execution of an OPEN statement for that unit is permitted. If the FILE specifier does not appear in the OPEN statement, the file connected to the unit by the OPEN statement is the same as the file that is already connected to the unit.

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

If the file to be connected to the unit is not the same as the file to which the unit is connected, the file that is currently connected to the unit is disconnected from that unit before the OPEN statement is executed. The effect is the same as if a CLOSE statement (without a STATUS specifier) had been executed before the OPEN statement.

If the file to be connected to the unit is the same as the file to which the unit is connected, the specifiers that appear in the OPEN statement must have the same values as those that are currently in effect; however, the BLANK specifier can have a value different from the value currently in effect. In that case, the new value of the BLANK specifier becomes effective. The position of the file is not affected.

If a file is connected to a unit, an OPEN statement can be used to connect that file to another unit. A file can be connected to more than one unit at the same time. The position of the file is not affected.

See figure 6-121 for examples of the OPEN statement. The first OPEN statement in the example connects the file MYFILE to unit 1. The file MYFILE is a direct access file with a record length of 10 characters.

The second OPEN statement in the example connects the file HERFILE to unit 1; however, the file MYFILE is already connected to unit 1. Therefore, the file MYFILE is disconnected from unit 1 before file HERFILE is connected.

If an input/output error occurs during execution of either of the OPEN statements, control transfers to the statement labeled 20 and the variable N is assigned the number of the execution-time error.

CLOSE STATEMENT

The CLOSE statement disconnects a file from a unit. See figure 6-122 for the format of the CLOSE statement.

```
CLOSE (cilist)

cilist    A control information list. The UNIT
          specifier must appear in cilist and
          must not be an asterisk.

          The following specifiers can also
          appear in cilist:

              ERR
              IOSTAT
              STATUS
```

Figure 6-122. CLOSE Statement Format

A CLOSE statement can appear in any program unit. The CLOSE statement need not appear in the same program unit as the OPEN statement that connects the file to the unit.

```
                .
                .
                .
                OPEN(1,FILE='MYFILE',ACCESS='DIRECT',RECL=10,ERR=20,
               +IOSTAT=N)
                READ(1,100,REC=2,ERR=20,IOSTAT=N) NPUT
100             FORMAT(I5)
                .
                .
                .
                OPEN(1,FILE='HERFILE',ERR=20,IOSTAT=N)
                READ(1,100,END=10,ERR=20,IOSTAT=N)
                .
                .
                .
                CLOSE(1,ERR=20,IOSTAT=N)
                .
                .
                .
10              CALL OUT (NPUT)
                .
                .
                .
20              CALL IOERR (N)
                .
                .
                .
```

Figure 6-121. OPEN and CLOSE Statement Examples

If a CLOSE statement specifies a unit that does not exist or has no file connected to it, the CLOSE statement has no effect.

After a file has been disconnected from a unit by a CLOSE statement, the file can be reconnected to a unit as long as the file still exists, and the unit can be reconnected to a file.

After normal termination of program execution, all files are automatically disconnected from their respective units. The effect is the same as if a CLOSE statement (with the STATUS specifier value KEEP) had been executed for each connected unit. However, if a particular file was connected by an OPEN statement with the STATUS specifier value SCRATCH, the effect is the same as if a CLOSE statement (with the STATUS specifier value DELETE) had been executed for that unit.

See figure 6-121 for an example of the CLOSE statement. The CLOSE statement in the example disconnects the file HERFILE from unit 1.

If an input/output error occurs during execution of the CLOSE statement, control transfers to the statement labeled 20 and the variable N is assigned the number of the execution-time error.

INQUIRE STATEMENT

The two types of INQUIRE statements are the INQUIRE by file statement and the INQUIRE by unit statement. The INQUIRE by file statement inquires about the properties of a particular file. The INQUIRE by unit statement inquires about the properties of a particular unit. See figure 6-123 for the format of the INQUIRE statement.

Following execution of an INQUIRE statement, the specified parameters contain values that are current at the time the statement is executed.

If a unit number is specified and the unit is opened, the ACCESS, BLANK, DIRECT, EXIST, FORM, FORMATTED, NEXTREC, NAME, NAMED, NUMBER, OPENED, RECL, SEQUENTIAL, and UNFORMATTED variables will contain information about the file associated with the unit. EXIST returns a TRUE value only if the unit has been opened by a reference on the PROGRAM statement or the OPEN statement; it does not indicate whether a file by this name is local or not.

If a file name is specified, the ACCESS, BLANK, DIRECT, EXIST, FORM, FORMATTED, NEXTREC, NAME, NAMED, NUMBER, OPENED, RECL, SEQUENTIAL, and UNFORMATTED variables will contain information about the file and the unit it is associated with.

If the file name specified in an INQUIRE by file statement is not valid or if the file does not exist, the values returned for the specifiers DIRECT, FORMATTED, NAME, NAMED, SEQUENTIAL, and UNFORMATTED are undefined.

```
INQUIRE (cilist)

cilist    A control information list. The FILE
          specifier must appear in cilist for
          an INQUIRE by file statement. The
          UNIT specifier must appear in cilist
          for an INQUIRE by unit statement, and
          the UNIT specifier must not be an
          asterisk.

          The following specifiers can also
          appear in cilist:

              ACCESS
              BLANK
              BUFS
              DIRECT
              ERR
              EXIST
              FORM
              FORMATTED
              IOSTAT
              NAME
              NAMED
              NEXTREC
              NUMBER
              OPENED
              RECL
              SEQUENTIAL
              UNFORMATTED
```

Figure 6-123. INQUIRE Statement Format

If the unit specified in an INQUIRE by unit statement is not valid or if the unit is not connected, the values returned for the specifiers ACCESS, BLANK, DIRECT, FORM, FORMATTED, NAME, NAMED, NEXTREC, NUMBER, RECL, SEQUENTIAL, and UNFORMATTED are undefined.

Values are always returned for the specifiers EXIST and OPENED unless an error occurs.

If an error occurs during the execution of an INQUIRE statement, the values returned for all specifiers in the INQUIRE statement except the IOSTAT specifier are undefined.

The INQUIRE statement can be executed before, during, or after a file is connected to a unit. The values returned for the specifiers are those that are current at the time the INQUIRE statement is executed.

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

See figure 6-124 for examples of the INQUIRE statement. The values returned for the specifiers in the INQUIRE statements are shown.

```
      .
      .
      .
      CHARACTER C1*8
      LOGICAL L1,L2,L3
      .
      .
      .
      OPEN(1,FILE='HISFILE',ACCESS='DIRECT',RECL=20)
      .
      .
      .
      INQUIRE(1,DIRECT=C1,OPENED=L1)
      .
      .
      .
      CLOSE(1)
      INQUIRE(FILE='HISFILE',EXIST=L2,OPENED=L3)
      .
      .
      .

Values returned by INQUIRE statements:

      C1 = YES
      L1 = .TRUE.
      L2 = .TRUE.
      L3 = .FALSE.
```

Figure 6-124. INQUIRE Statement Examples

FILE POSITIONING STATEMENTS

The file positioning statements change the position of a file that is connected to a unit.

The file positioning statements are:

REWIND statement

BACKSPACE statement

ENDFILE statement

Each of these statements is described in the following paragraphs.

REWIND STATEMENT

The REWIND statement positions a file at its initial point. See figure 6-125 for the format of the REWIND statement.

If the file is already positioned at its initial point, the REWIND statement has no effect. A REWIND statement for a file that is connected but does not exist has no effect.

The REWIND statement cannot rewind a file connected for direct access.

See figure 6-126 for an example of the REWIND statement. The REWIND statement in the example positions the file connected to unit 1 to its initial point.

```
REWIND (cilist)
    or
REWIND uid


cilist     A control information list. The UNIT
           specifier must appear in cilist and
           must not be an asterisk.

           The following specifiers can also
           appear in cilist:

               ERR
               IOSTAT

uid        A unit identifier.
```

Figure 6-125. REWIND Statement Format

```
            .
            .
            .
            DO 5 I=1,10
            WRITE(1,100) I
      100   FORMAT('ΔRECORDΔ',I2)
      5     CONTINUE
            ENDFILE 1
            .
            .
            .
            REWIND 1
            READ(1,105) I
      105   FORMAT(8X,I2)
            BACKSPACE 1
            .
            .
            .
```

Figure 6-126. REWIND, BACKSPACE, and ENDFILE Statement Example

BACKSPACE STATEMENT

The BACKSPACE statement positions a file before the preceding record. See figure 6-127 for the format of the BACKSPACE statement.

If there is no preceding record, the BACKSPACE statement has no effect. If the preceding record is an endfile record, the BACKSPACE statement positions the file before the endfile record.

A BACKSPACE statement for a file that is connected but does not exist is not permitted.

Backspace can only be done on units that have been used by the OPEN statement or some data transfer I/O statement.

The BACKSPACE statement cannot position a file connected for direct access.

See figure 6-126 for an example of the BACKSPACE statement. The BACKSPACE statement in the example positions the file connected to unit 1 to the preceding record, which is the first record.

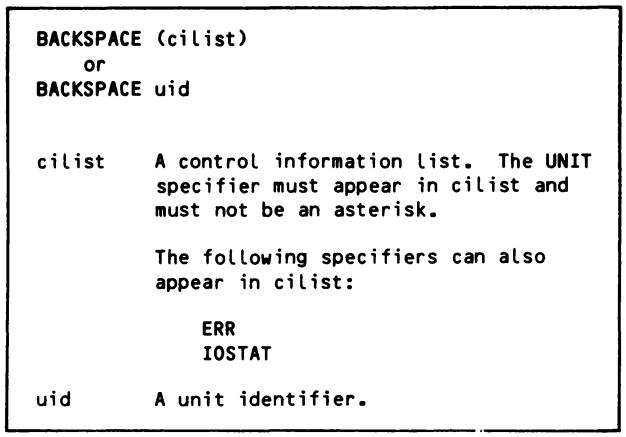


Figure 6-127. BACKSPACE Statement Format

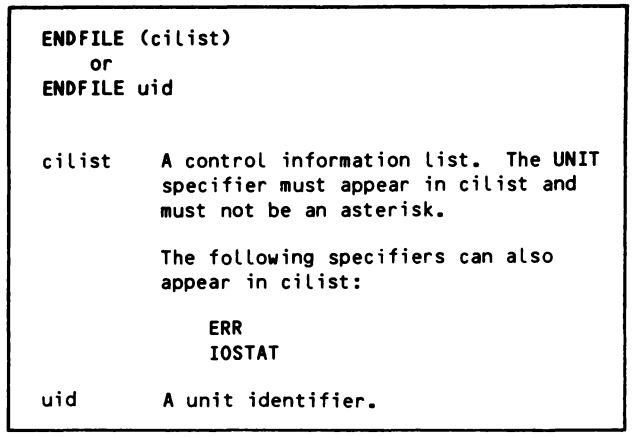


Figure 6-128. ENDFILE Statement Format

ENDFILE STATEMENT

The ENDFILE statement outputs an endfile record as the next record of the file. The file is then positioned after the endfile record. See figure 6-128 for the format of the ENDFILE statement.

The ENDFILE statement cannot write an endfile record on a file connected for direct access; it can only write an endfile record on a file connected for sequential access. If a file containing an endfile record is connected for direct access, only the records that precede the endfile record can be read.

After execution of an ENDFILE statement, a REWIND or BACKSPACE statement must be used to reposition the file before data can be input or output to the file.

An ENDFILE statement for a file that is connected but does not exist creates the file.

See figure 6-126 for an example of the ENDFILE statement. The ENDFILE statement in the example writes an endfile record on the file connected to unit 1.

A FORTRAN program consists of one or more program units. A program unit is a main program, a function subprogram, a subroutine subprogram, or a block data subprogram.

This section describes main programs, function subprograms, subroutine subprograms, block data subprograms, and statement functions. Intrinsic functions and predefined subroutines are described in sections 10 and 11.

MAIN PROGRAMS

A main program is a group of statements that begins with an optional PROGRAM statement and ends with an END statement. See figure 7-1 for the structure of a main program. A FORTRAN program must have one main program.

```
PROGRAM statement (optional)

    Main program body

END statement
```

Figure 7-1. Main Program Structure

PROGRAM STATEMENT

The PROGRAM statement is the first statement in a main program; however, the PROGRAM statement can be omitted. The PROGRAM statement assigns a name to the program and optionally declares files, preconnects files to units for input/output operations performed by the program, and requests the mapping of dynamic space into large pages. See figure 7-2 for the format of the PROGRAM statement.

Using a PROGRAM statement to preconnect files to units eliminates the need to connect files to units using the OPEN statement. See table 7-1 for examples of file preconnection using the PROGRAM statement. For each PROGRAM statement example, the table shows the OPEN statement that would result in the equivalent unit and file connection.

The attributes of a preconnected file are the attributes already assigned to the file if the file exists. If the file is a new file, the attributes of the file are assigned when the first input/output statement that references the file is executed.

Certain units and files are automatically preconnected for formatted sequential input/output. These units and files are used for input/output statements that do not contain control information lists or whose UNIT specifiers are asterisks. See table 7-2 for the units and files that are automatically preconnected and the statements that can use the files and units.

TABLE 7-2. AUTOMATICALLY PRECONNECTED FILES AND UNITS

File	Unit	Statements that can use the preconnection
INPUT	5HINPUT	Formatted READ statement List-directed READ statement Namelist READ statement
OUTPUT	6HOUTPUT	Formatted WRITE statement Formatted PRINT statement List-directed PRINT statement Namelist PRINT statement
PUNCH	5HPUNCH	Formatted PUNCH statement List-directed PUNCH statement Namelist PUNCH statement

MAIN PROGRAM BODY

The body of a main program can contain nonexecutable and executable statements. The statements that must not appear in the body of a main program are: BLOCK DATA, PROGRAM, FUNCTION, SUBROUTINE, RETURN, END, and ENTRY. The appearance of a SAVE statement in a main program has no effect.

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

Program execution begins with the first executable statement in the main program.

TABLE 7-1. FILE CONNECTION EXAMPLES

PROGRAM Statement Preconnection	Equivalent OPEN Statement Connection
PROGRAM NAME(MYFILE)	OPEN(UNIT=6HMYFILE,FILE='MYFILE', ...)
PROGRAM NAME(TAPE1)	OPEN(UNIT=1,FILE='TAPE1', ...)
PROGRAM NAME(UNIT1)	OPEN(UNIT=1,FILE='UNIT1', ...)
PROGRAM NAME(MYFILE,UN=MYFILE)	OPEN(UNIT=2HUN,FILE='MYFILE', ...)
PROGRAM NAME(MYFILE,TAPE1=MYFILE)	OPEN(UNIT=1,FILE='MYFILE', ...)
PROGRAM NAME(MYFILE,UNIT1=MYFILE)	OPEN(UNIT=1,FILE='MYFILE', ...)

PROGRAM pname (ps_1 ... ,ps_n,[RLP=nlp])

pname — A symbolic name that is used as the name of the main entry point of the program and the name of the object module. The symbolic name must not be the same as the name of an external subprogram, a block data subprogram, or a common block that is used in the program. The symbolic name must not be the same as any symbolic name that is used in the main program unit, but it can be the same as a file name or an alternate unit name. If the PROGRAM statement is omitted, pname is M_A_I_N.

ps_i — A preconnection specifier; optional; ps_i can be a file declaration specifier or an alternate unit specifier.

A file declaration specifier has the following format:

fn or fn=bl

fn — A symbolic file name.

bl — An unsigned integer constant greater than or equal to 1 and less than or equal to 24; bl is the buffer length in 512 word blocks for the file. The default is 8. If a file name appears in the PROGRAM statement, the buffer length of the file must not be changed by any OPEN statements.

An alternate unit specifier has the following format:

an=fn

an — A symbolic name that is used as an alternate unit name; the value specified for an must be such that nHan is an external unit identifier (n is the number of characters in an, and an is a valid system file name). Usually an is of the form TAPEk or UNITk, where k is an integer that is no less than 0 and no greater than 999 and has no leading zeros.

fn — A symbolic file name; fn must appear previously in a file declaration specifier; fn can appear in more than one alternate unit specifier.

[RLP=nlp]† — A dynamic space mapping parameter of the form [RLP=nlp]; optional. It requests that a certain number of large pages of dynamic space be mapped in at the start of program execution. This map parameter can appear anywhere in the PROGRAM statement parameter list; however, it must not appear more than once. Its omission means that dynamic space is mapped in small pages.

The map parameter has the following format:

[RLP] or [RLP=nlp]

nlp — Optional unsigned integer constant; nlp gives the number of large pages of dynamic space that are mapped in at the start of program execution. A large page is 65536 full words. The default for nlp is 1. Thus, [RLP] has the same meaning as [RLP=1].

If the PROGRAM statement has neither ps_i parameters nor an RLP parameter, the parentheses must not appear.

---

†The brackets [ ] are part of the map parameter. If you use the map parameter, the brackets must appear.

Figure 7-2. PROGRAM Statement Format

END STATEMENT FOR MAIN PROGRAMS

A main program must end with one END statement. See figure 7-3 for the format of the END statement. An END statement can contain a statement label. The END statement must not be continued. If an END statement in a main program is executed before a STOP statement, the END statement has the same effect as a STOP statement.

```
END
```

Figure 7-3. END Statement Format

MAIN PROGRAM EXAMPLE

See figure 7-4 for an example of a main program. The PROGRAM statement in the main program specifies that the name of the program is AVG, and declares two files, DATA and OUT. The file DATA is preconnected to unit 1 and the file OUT is preconnected to unit 2. The file DATA must exist before program execution and must be a formatted sequential external file. The file OUT need not exist before program execution, but if it does exist, it must be a formatted sequential external file. If the file OUT does not exist, it becomes a formatted sequential external file when the WRITE statement is executed.

```
      PROGRAM AVG(DATA,OUT,TAPE1=DATA,TAPE2=OUT)
      INTEGER NUMBER(10), NSUM
      REAL RESULT
      READ(1,100)(NUMBER(I),I=1,10)
      NSUM = NADD(NUMBER)
      CALL DIVIDE(NSUM,10,RESULT)
      WRITE(2,200)(NUMBER(I),I=1,10),RESULT
      STOP
100   FORMAT(10I2)
200   FORMAT('1THE AVERAGE OF'/10(' ',I2)/' IS ',F7.3)
      END

      FUNCTION NADD(IARRAY)
      INTEGER IARRAY(10)
      N = 0
      DO 10 I=1,10
      N = IARRAY(I) + N
10    CONTINUE
      NADD = N
      RETURN
      END

      SUBROUTINE DIVIDE(N,I,Q)
      INTEGER N,I
      REAL Q
      Q = REAL(N)/REAL(I)
      RETURN
      END
```

Figure 7-4. Main Program, Function, and Subroutine Example

FUNCTION SUBPROGRAMS

A function subprogram is a group of statements that begins with a FUNCTION statement and ends with an END statement. See figure 7-5 for the structure of a function subprogram.

A FORTRAN program can have any number of function subprograms. A function subprogram is executed when a function reference is encountered in a statement that appears in a program unit other than the function subprogram being referenced.

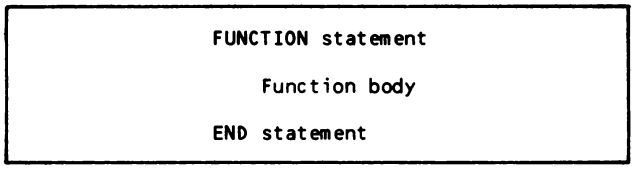


Figure 7-5. Function Subprogram Structure

A function subprogram differs from a subroutine subprogram in a number of ways. See table 7-3 for a summary of the differences between function subprograms and subroutine subprograms.

TABLE 7-3. DIFFERENCES BETWEEN FUNCTIONS AND SUBROUTINES

	Functions	Subroutines
How Referenced	The function name appears in an expression. Parentheses must follow the name even if there are no arguments.	The subroutine name appears in a CALL statement. Parentheses after the name can be omitted if there are no arguments.
Results	A function must return a value through the function name. It can also return values through common blocks and by modifying its arguments; certain restrictions apply and are described later in this section.	A subroutine can return any number of values through arguments and common blocks.
Type and Length	A function name has a data type and length. The type and length of a function name are the type and length of the function result.	A subroutine name does not have a data type or length.
Alternate Return	Alternate return specifiers must not occur as arguments.	Alternate return specifiers can occur as arguments.

Functions that you define are external functions. You can also reference function subprograms that are predefined; these functions are called intrinsic functions. Some intrinsic functions are external functions, some intrinsic functions are inline functions, and some intrinsic functions can be either external or inline. See section 10 for a description of the external and inline intrinsic functions.

FUNCTION STATEMENT

The FUNCTION statement is the first statement in a function subprogram. The FUNCTION statement assigns a name and a type to the function. The FUNCTION statement can also specify dummy arguments used in the function subprogram. Dummy arguments are described later in this section. See figure 7-6 for the format of the FUNCTION statement.

```
typ FUNCTION fname (darg1, ... ,dargn)
    or
CHARACTER FUNCTION fname*len (darg1, ... ,dargn)
```

typ — A type specification for fname; optional; typ can be any of the following:

- INTEGER
- HALF PRECISION
- REAL
- DOUBLE PRECISION
- COMPLEX
- LOGICAL
- CHARACTER
- CHARACTER*len

fname — A symbolic name that is used as the name of the main entry point of the function.

len — An integer constant expression whose result is greater than 0; optional; len specifies the length in characters of fname.

An asterisk enclosed in parentheses can be specified for len, which indicates that the length of fname is the same as the length of fname in the referencing program unit.

If len is omitted, the preceding asterisk must not appear. The default is 1.

$darg_i$ — A dummy argument, which can be a variable, array, descriptor, descriptor array, dummy function name, or dummy subroutine name; optional. No two dummy arguments can have the same name. The parentheses are required even if no dummy arguments appear.

Figure 7-6. FUNCTION Statement Format

The type of the function name is determined by the FUNCTION statement, by a type specification statement that appears in the function body, or by the first letter of the function name. An IMPLICIT statement that appears in the function body can affect the type of the function name. The type of the function name must be the same in all program units that reference the function.

The name of a function subprogram is the name of the main entry point of the function. You can assign other names to a function subprogram by using the ENTRY statement; these other names are the names of secondary entry points of the function. Secondary entry points are described later in this section.

FUNCTION BODY

The body of a function subprogram can contain nonexecutable and executable statements. The statements that must not appear in the body of a function subprogram are: BLOCK DATA, PROGRAM, FUNCTION, SUBROUTINE, and END. The body of a function subprogram must not contain a statement that directly or indirectly references the function subprogram.

Execution of a function normally begins with the first executable statement in the function subprogram. Execution of a function subprogram that has secondary entry points can begin elsewhere in the body of the function subprogram. Secondary entry points are described later in this section.

The function name is considered to be a variable name in the function body. A value must be assigned to the function name before a RETURN statement is executed. The value of the function name can be referenced and changed in the function body. The value that the function name has when a RETURN statement is executed is the value that is returned to the program unit that referenced the function.

The function name must not be initialized in a type specification or DATA statement. The function name must not appear in any nonexecutable statements in the function body except in a type specification statement or in the input/output list of a NAMELIST statement.

If you use a function name that is the same as the name of an intrinsic function, you cannot reference the intrinsic function in the function body. See section 10 for a list of the intrinsic function names.

The statements in the function body can modify the arguments that are passed to the function in order to return additional values to the program unit that references the function; however, a function must not modify any arguments that are used elsewhere in the statement that contains the function reference. Also, a function must not modify any dummy arguments whose corresponding actual arguments are constants, symbolic constants, substrings, vectors, function references, or expressions that contain operators or are enclosed in parentheses.

See figure 7-7 for an example of a function that modifies its arguments. The function reference in the example passes the values 9.0 and 16.0 to the function PYTHAG. The function returns the value 5.0 through its name; furthermore, the function changes the value of A to 3.0 and changes the value of B to 4.0.

```
PROGRAM ARGMOD
REAL A,B,C
DATA A,B /9.0,16.0/
.
.
.
C = PYTHAG(A,B)
.
.
.
END

FUNCTION PYTHAG(A,B)
REAL A,B
PYTHAG = SQRT(A+B)
A = SQRT(A)
B = SQRT(B)
RETURN
END
```

Figure 7-7. Modification of Function Arguments Example

The statements in the function body can modify the values of common block elements in order to return additional values to the program unit that references the function; however, two restrictions apply:

A function must not modify any common block elements that are used elsewhere in the statement that contains the function reference.

A function must not modify a common block element if the value of the common block element affects another function reference that is in the same statement.

RETURN STATEMENT FOR FUNCTION SUBPROGRAMS

The RETURN statement is an executable statement that returns control from a function subprogram to the program unit that called the function subprogram. See figure 7-8 for the format of the RETURN statement for function subprograms.

```
RETURN
```

Figure 7-8. RETURN Statement for Function Subprograms Format

In a function subprogram, execution of a RETURN statement causes the value computed by the function to replace the function reference. Evaluation of the statement that contains the function reference continues.

END STATEMENT FOR FUNCTION SUBPROGRAMS

A function must end with one END statement. See figure 7-3 for the format of the END statement. An END statement can contain a statement label. The END statement must not be continued. If an END statement in a function is executed before a RETURN statement, the END statement has the same effect as a RETURN statement.

FUNCTION REFERENCES

A function is referenced by using a function reference. See figure 7-9 for the format of a function reference.

fname($aarg_1$, ... ,$aarg_n$)

fname	The name of an entry point of a function subprogram.
$aarg_i$	An actual argument, which can be a constant, symbolic constant, expression (except concatentation of an operand whose length is specified as (*)), substring, variable, array, array element, vector reference, descriptor, descriptor array, descriptor array element, actual function name, actual subroutine name, dummy function name, or dummy subroutine name; optional. The parentheses are required even if no actual arguments appear.

Figure 7-9. Function Reference Format

A function can be referenced from a main program, statement function, another function subprogram, or a subroutine subprogram. A function reference can appear in an arithmetic, logical, or character expression.

Recursive references are not permitted. A recursive reference is a reference that directly or indirectly references the function in which it appears.

A function reference causes control to transfer to the function subprogram. The statements in the function are executed during the evaluation of the expression in which the function reference appears. The values that the actual arguments have at the time the function is referenced are the values that are used during execution of the function.

When a RETURN statement is executed in a function subprogram, control returns to the program unit that referenced the function. Evaluation of the statement that contains the function reference then continues.

A reference to a function that has the same name as an intrinsic function references the intrinsic function rather than your function. See section 10 for a list of the intrinsic function names. In

order to reference a function that has the same name as an intrinsic function, you must declare the function name in an EXTERNAL statement in all program units that reference that function. See section 3 for a description of the EXTERNAL statement.

See figure 7-10 for an example of a reference to a function that has the same name as an intrinsic function. The function reference in the example references the function TIME that is shown, rather than the intrinsic function TIME.

```
      PROGRAM EXTFUN
      REAL IN,OUT,HOURS
      EXTERNAL TIME
      .
      .
      .
      READ 100, IN,OUT
100   FORMAT(2F4.1)
      HOURS = TIME(IN,OUT)
      .
      .
      .
      END

      FUNCTION TIME(IN,OUT)
      REAL IN,OUT
      IF(IN.LE.OUT) THEN
         TIME = OUT - IN
      ELSE IF(IN.GT.12) THEN
         TIME = OUT + (24 - IN)
      ELSE
         TIME = OUT + (12 - IN)
      END IF
      RETURN
      END
```

Figure 7-10. Function With Same Name as an Intrinsic Function Example

FUNCTION SUBPROGRAM EXAMPLE

See figure 7-4 for an example of a function subprogram. The name of the function in the example is NADD. NADD is an integer function that has one dummy argument, IARRAY, which is an array of 10 elements. The function is referenced in the main program. The actual argument in the function reference is NUMBER, which is an array of 10 elements.

SUBROUTINE SUBPROGRAMS

A subroutine subprogram is a group of statements that begins with a SUBROUTINE statement and ends with an END statement. See figure 7-11 for the structure of a subroutine subprogram.

```
SUBROUTINE statement

    Subroutine body

END statement
```

Figure 7-11. Subroutine Subprogram Structure

A FORTRAN program can have any number of subroutine subprograms. A subroutine subprogram is executed when a CALL statement is executed in a program unit other than the subroutine subprogram being called.

A subroutine subprogram differs from a function subprogram in a number of ways. See table 7-3 for a summary of the differences between subroutine subprograms and function subprograms.

You can define and reference your own subroutine subprograms. You can also reference subroutine subprograms that are predefined. See section 11 for a description of the predefined subroutines. All subroutine subprograms are external subprograms.

SUBROUTINE STATEMENT

The SUBROUTINE statement is the first statement in a subroutine subprogram. The SUBROUTINE statement assigns a name to the subroutine and can also specify dummy arguments that are used in the subroutine subprogram. Dummy arguments are described later in this section. See figure 7-12 for the format of the SUBROUTINE statement.

SUBROUTINE sname ($darg_1$, ... ,$darg_n$)

sname	A symbolic name that is used as the name of the main entry point of the subroutine.
$darg_i$	A dummy argument, which can be a variable, array, descriptor, descriptor array, dummy function name, dummy subroutine name, or asterisk; optional. No two dummy arguments can have the same name. If no dummy arguments appear, the parentheses are optional.

Figure 7-12. SUBROUTINE Statement Format

The name of a subroutine subprogram cannot be associated with a data type; the name of a subroutine subprogram is the name of the main entry point of the subroutine. You can assign other names to a subroutine subprogram by using the ENTRY statement; these other names are the names of secondary entry points of the subroutine. Secondary entry points are described later in this section.

SUBROUTINE BODY

The body of a subroutine subprogram can contain nonexecutable and executable statements. The statements that must not appear in the body of a subroutine subprogram are: BLOCK DATA, PROGRAM, FUNCTION, SUBROUTINE, and END. The body of a subroutine subprogram must not contain a statement that directly or indirectly calls the subroutine subprogram.

Execution of a subroutine normally begins with the first executable statement in the subroutine subprogram. Execution of a subroutine subprogram that has secondary entry points can begin elsewhere in the body of the subroutine subprogram. Secondary entry points are described later in this section.

The subroutine name must not appear in any statements in the subroutine body.

The statements in the subroutine body can modify the arguments that are passed to the subroutine; however, a subroutine must not modify dummy arguments whose corresponding actual arguments are constants, symbolic constants, substrings, vectors, function references, or expressions that contain operators or are enclosed in parentheses. The statements in the subroutine body can modify any common block elements without restriction.

RETURN STATEMENT FOR SUBROUTINE SUBPROGRAMS

The RETURN statement is an executable statement that returns control from a subroutine subprogram to the program unit that called the subroutine subprogram. See figure 7-13 for the format of the RETURN statement for subroutine subprograms.

```
RETURN alt

alt     An integer constant or a simple
        integer variable that indicates to
        which of the statement labels in the
        CALL statement control is to
        transfer when the RETURN statement
        is executed; optional. If alt is
        not specified, control transfers to
        the statement that follows the CALL
        statement.
```

Figure 7-13. RETURN Statement for Subroutine Subprograms Format

In a subroutine subprogram, execution of a RETURN statement transfers control to the first executable statement after the CALL statement that called the subroutine subprogram. You can specify that control be returned to another statement in the program unit that called the subroutine by using alternate returns.

In order to use an alternate return, you must supply a list of statement labels in the actual argument list of the CALL statement that calls the subroutine. See the description of the CALL statement for the syntax of the statement label list.

You must also place asterisks in the dummy argument lists of the SUBROUTINE statement or ENTRY statement in the subroutine subprogram. The asterisks must correspond to the statement labels specified in the actual argument list of the CALL statement. See the description of the SUBROUTINE statement and the description of the ENTRY statement for the syntax of the dummy argument list.

The RETURN statement parameter can then be used to indicate to which of the statement labels in the CALL statement control transfers when the RETURN statement is executed. A RETURN statement parameter of 1 indicates that control returns to the first statement label in the actual argument list, a RETURN statement parameter of 2 indicates that control returns to the second statement label in the actual argument list, and so on.

If the RETURN statement parameter is less than 1 or greater than the number of statement labels specified in the actual argument list, control returns to the first executable statement after the CALL statement that referenced the subprogram.

You can use a RETURN statement that has no RETURN statement parameter in a subroutine that also uses alternate returns.

Alternate returns must not appear in a function subprogram.

END STATEMENT FOR SUBROUTINE SUBPROGRAMS

A subroutine must end with one END statement. See figure 7-3 for the format of the END statement. An END statement can contain a statement label. The END statement must not be continued. If an END statement in a subroutine is executed before a RETURN statement, the END statement has the same effect as a RETURN statement with no alternate return specifier.

SUBROUTINE CALLS

A subroutine is called by using the CALL statement. The CALL statement is an executable statement that transfers control to a subroutine subprogram or a predefined subroutine. See figure 7-14 for the format of the CALL statement.

A subroutine can be called from a main program, function subprogram, or another subroutine subprogram.

Recursive calls are not permitted. A recursive call is a CALL statement that directly or indirectly calls the program unit in which it appears.

When a CALL statement is executed, control transfers to the subroutine subprogram or predefined subroutine specified in the CALL statement. Execution of the CALL statement is not complete until control returns from the subroutine subprogram or predefined subroutine specified in the CALL statement.

Control normally returns to the first executable statement after the CALL statement. However, you can specify that control return to some other statement in the calling program unit. See the description of the RETURN statement for subroutine subprograms.

SUBROUTINE SUBPROGRAM EXAMPLE

See figure 7-4 for an example of a subroutine subprogram. The name of the subroutine in the example is DIVIDE. It has three dummy arguments, N, I, and Q that are variables of type integer, integer, and real, respectively. The subroutine is called in the main program. The actual arguments in the CALL statement are NSUM, 10, and RESULT. NSUM is an integer variable, 10 is an integer constant, and RESULT is a real variable.

CALL sname (argument list)

sname	The name of an entry point of a subroutine subprogram.
argument list $aarg_1$, ... ,$aarg_n$	If no argument list appears, the parentheses are optional.
$aarg_i$	An actual argument, which can be a constant, symbolic constant, expression (except concatentation of an operand whose length is specified as (*)), substring, variable, array, array element, vector reference, descriptor, descriptor array, descriptor array element, actual function name, actual subroutine name, dummy function name, dummy subroutine name, or alternate return specifier. An alternate return specifier is a statement label prefixed by an asterisk or ampersand. A statement label that appears in the argument list of a CALL statement must appear in the label field of an executable statement in the program unit that contains the CALL statement.

Figure 7-14. CALL Statement Format

SUBPROGRAM COMMUNICATION

You can transfer data between main programs, function subprograms, and subroutine subprograms in two ways: by using common blocks and by using arguments.

COMMON BLOCKS

Common blocks are areas of storage that can be referenced by one or more program units. The two types of common blocks are unnamed common blocks and named common blocks. See section 3 for a description of common blocks.

ARGUMENTS

Arguments are individual language elements that can be referenced by one or more program units. The two types of arguments are dummy arguments and actual arguments.

Dummy Arguments

Dummy arguments are variables, arrays, descriptors, descriptor arrays, dummy function names, or dummy subroutine names that appear in the argument list of a FUNCTION, SUBROUTINE, or ENTRY statement. Asterisks used for alternate returns from a subroutine can also appear in the dummy argument list of a SUBROUTINE statement, or in the dummy argument list of an ENTRY statement that appears in a subroutine subprogram.

Dummy arguments must be assigned appropriate data types. Dummy arguments can be used in the body of a function or subroutine subprogram. A dummy argument must not appear in a COMMON, EQUIVALENCE, or DATA statement.

Actual Arguments

Actual arguments are constants, symbolic constants, expressions (except concatenation of an operand whose length is specified as (*)), substrings, variables, arrays, array elements, vectors, descriptors, descriptor arrays, descriptor array elements, actual function names, actual subroutine names, dummy function names, or dummy subroutine names that appear in the argument list of a function reference or CALL statement. Alternate return specifiers can appear in the actual argument list of a CALL statement.

ARGUMENT CORRESPONDENCE

Each actual argument corresponds to a dummy argument. The data type of an actual argument must be the same as the data type of the dummy argument to which it corresponds, except that an actual argument of type Hollerith can correspond to a dummy argument of any type other than character or bit.

The length of an actual argument must be the same as the length of the dummy argument to which it corresponds.

The number and order of the actual arguments must be the same as the number and order of the dummy arguments.

If an actual argument is a constant, symbolic constant, expression that contains operators or is enclosed in parentheses, substring, or vector, the value of the corresponding dummy argument must not be modified in the subprogram.

See table 7-4 for the legal correspondences of actual and dummy arguments.

RESTRICTIONS ON ASSOCIATION OF ARGUMENTS

If a subprogram reference causes a dummy argument in the referenced subprogram to become associated with another dummy argument in the referenced subprogram, neither dummy argument can be defined during execution of that subprogram.

For example, if a subroutine is headed by:

```
SUBROUTINE XYZ (A,B)
```

and is referenced by:

```
CALL XYZ (C,C)
```

then the dummy arguments A and B each become associated with the same actual argument C and therefore with each other. Neither A nor B can be defined during this execution of subroutine XYZ or by any procedures referenced by XYZ.

If a subprogram reference causes a dummy argument to become associated with an entity in a common block in the referenced subprogram or in a subprogram referenced by the referenced subprogram, neither the dummy argument nor the entity in the common block can be defined within the subprogram or within a subprogram referenced by the referenced subprogram.

For example, if a subroutine contains the statements:

```
SUBROUTINE XYZ (A)
COMMON C
```

and is referenced by a program unit that contains the statements:

```
COMMON B
CALL XYZ (B)
```

then the dummy argument A becomes associated with the actual argument B. Because B and C are allocated the same space in the blank common block, A and C then reference the same space. Neither A nor C can be defined during execution of the subroutine XYZ or by any procedures referenced by XYZ.

ARRAYS AS DUMMY ARGUMENTS

The size of an array that is a dummy argument must be declared in the subprogram like all other arrays.Dimension bound expressions for such an array can contain integer variables that are in a common block or that are dummy arguments. If an integer variable that is used in a dimension bound expres- sion is a dummy argument, it must appear in the dummy argument list of every FUNCTION, SUBROUTINE, and ENTRY statement that contains the array name.

The upper bound of the last dimension of a column-wise array can be an asterisk. The upper bound of the first dimension of a rowwise array can be an asterisk. Such an array is called an assumed-size array. An assumed-size array must not appear without subscripts in an input/output list or in an array assignment statement.

SUBPROGRAM NAMES AS ACTUAL ARGUMENTS

If an intrinsic function name is used as an actual argument, it must be declared in an INTRINSIC statement in the calling program unit. If a subroutine name or an external function name is used as an actual argument, it must be declared in an EXTERNAL statement in the calling program unit. The corresponding dummy argument in the referenced subprogram can be used either as an actual argument in subprogram references, or as a subprogram name in subprogram references.

TABLE 7-4. DUMMY AND ACTUAL ARGUMENT CORRESPONDENCE

Dummy Argument	Actual Argument
Variable	Constant Symbolic constant Scalar expression Substring Variable Array element
Array	Array element Array
Descriptor	Vector reference Descriptor Descriptor array element
Descriptor array	Descriptor array element Descriptor array
Dummy function name	Actual function name Dummy function name
Dummy subroutine name	Actual subroutine name Dummy subroutine name
Asterisk denoting alternate return for subroutine only	Alternate return specifier (a statement label prefixed by an asterisk or ampersand)
Asterisk denoting vector function result	Vector reference Descriptor array element Descriptor

ENTRY POINTS

An entry point is a place in a function or subroutine where execution begins when the function or subroutine is referenced. Each function and subroutine has one main entry point. The main entry point is the first executable statement after the FUNCTION or SUBROUTINE statement.

An entry point is identified by an entry point name. The name of the main entry point is the name of the function or subroutine, which is specified by the FUNCTION or SUBROUTINE statement.

Functions and subroutines can have entry points other than the main entry point. These entry points are called secondary entry points. Secondary entry points are identified by secondary entry point names. Secondary entry points are defined by using the ENTRY statement.

ENTRY STATEMENT

The ENTRY statement specifies that the first executable statement after the ENTRY statement is a secondary entry point. The ENTRY statement also specifies the name of the secondary entry point it defines. You can specify any number of secondary entry points in a function or subroutine. See figure 7-15 for the format of the ENTRY statement.

ENTRY sename ($darg_1$, ... ,$darg_n$)

sename — A symbolic name that is used as the name of a secondary entry point in the function or subroutine.

$darg_i$ — A dummy argument, which can be a variable, array, descriptor, descriptor array, dummy function name, dummy subroutine name, or asterisk; optional. An asterisk can be specified only if the ENTRY statement appears in a sub- routine. No two dummy arguments can have the same name. If no dummy arguments appear, the parentheses are optional.

Figure 7-15. ENTRY Statement Format

An ENTRY statement is nonexecutable. An ENTRY statement must not appear in the range of a DO statement or in an if-block, elseif-block, or else-block. An ENTRY statement must not appear in a main program or in a block data subprogram.

SECONDARY ENTRY POINTS IN FUNCTIONS

When an ENTRY statement appears in a function subprogram, a secondary entry point is defined for the function. The type of a secondary entry point name in a function is determined by a type specification statement that appears in the function body or by the first letter of the secondary entry point name. An IMPLICIT statement that appears in the function body can affect the type of the secondary entry point.

A secondary entry point name need not be associated with the same data type as the main entry point name or any other secondary entry point names in the function; however, a function reference that uses a secondary entry point name must be associated with the same data type as the secondary entry point name.

Scalar functions can have vector secondary entry points, and vector functions can have scalar secondary entry points. See section 9 for a description of vector functions.

The name of a secondary entry point in a function must not appear in any nonexecutable statements in the function body except in a type specification statement or in the input/output list of a NAMELIST statement.

At least one secondary entry point name must be assigned a value before a RETURN statement is executed. When one entry point name is assigned a value, all other entry point names that have the same data type and length are assigned the same value. The values of entry points that are not of the same data type or length are undefined.

SECONDARY ENTRY POINTS IN SUBROUTINES

When an ENTRY statement appears in a subroutine subprogram, a secondary entry point is defined for the subroutine. The name of a secondary entry point in a subroutine must not be associated with a data type.

The name of a secondary entry point in a subroutine must not appear in any statements in the subroutine body.

REFERENCING SECONDARY ENTRY POINTS

A secondary entry point name of a function is referenced in the same way as the main entry point name is referenced; however, the secondary entry point name is used in the function reference rather than the main entry point name. The secondary entry point reference must be of the same type as the secondary entry point. See the description of function references for the format of a function reference.

A secondary entry point name of a subroutine is called in the same way as the main entry point name; however, the secondary entry point name is used in the CALL statement rather than the main entry point name. See the description of subroutine calls for the format of the CALL statement.

Recursive function references and recursive subroutine calls are not permitted. A function or subroutine must not directly or indirectly reference any of its own entry points. In a function, the value of an entry point name can be referenced and changed because the entry point name is treated like a variable name in the function body; if an entry point name is not followed by an argument list, the use of the entry point name is not a function reference.

SECONDARY ENTRY POINT ARGUMENT LISTS

A secondary entry point of a function must have at least one dummy argument; a secondary entry point of a subroutine need not have any arguments.

The list of arguments in an ENTRY statement need not contain the same elements as the argument lists of other FUNCTION, SUBROUTINE, or ENTRY statements in the same program unit.

A dummy argument must not appear in an executable statement that precedes the first dummy argument list in which the dummy argument appears. Thus, a dummy argument in an ENTRY statement must not appear in an executable statement preceding that ENTRY statement unless the dummy argument also appears in a FUNCTION, SUBROUTINE, or ENTRY statement that precedes the executable statement.

A dummy argument in an ENTRY statement must not appear in the expression of a statement function definition unless one of the following is true:

- The dummy argument in the ENTRY statement is also a dummy argument in the statement function.
- The dummy argument in the ENTRY statement is also a dummy argument in the FUNCTION or SUBROUTINE statement.
- The ENTRY statement in which the dummy argument appears precedes the statement function definition.

An executable statement containing a dummy argument can be executed only if the dummy argument appears in the argument list of the entry point that was referenced.

SECONDARY ENTRY POINT EXAMPLE

See figure 7-16 for an example of a function that has a secondary entry point. The main entry point of the function in the example is ISOS; a secondary entry point is ANYTRI. Because ISOS and ANYTRI are of the same data type and length, assigning a value to ANYTRI causes the same value to be assigned to ISOS.

```
      PROGRAM ENT
      REAL ISOS,ANYTRI,BASE,HEIGHT
      .
      .
      .
      AREA = ISOS(BASE,BASE)
      .
      .
      .
      AREA = ANYTRI(BASE,HEIGHT)
      .
      .
      .
      END

      FUNCTION ISOS(B,H)
      REAL ISOS,ANYTRI,B,H
      H = SQRT((B**2) - ((BASE/2)**2))
      ENTRY ANYTRI(B,H)
      ANYTRI = (B*H)/2
      RETURN
      END
```

Figure 7-16. Secondary Entry Points Example

BLOCK DATA SUBPROGRAMS

A block data subprogram is a group of statements that begins with a BLOCK DATA statement and ends with an END statement. See figure 7-17 for the structure of a block data subprogram.

```
      BLOCK DATA statement

          Block data subprogram body

      END statement
```

Figure 7-17. Block Data Subprogram Structure

A FORTRAN program can have any number of block data subprograms. A block data subprogram is a nonexecutable subprogram.

BLOCK DATA STATEMENT

The BLOCK DATA statement is the first statement in a block data subprogram. The BLOCK DATA statement can assign a name to the block data subprogram. See figure 7-18 for the format of the BLOCK DATA statement.

```
BLOCK DATA bname

bname    A symbolic name; optional; bname is
         used as the name of the block data
         subprogram.
```

Figure 7-18. BLOCK DATA Statement Format

BLOCK DATA SUBPROGRAM BODY

The body of a block data subprogram can contain any of the following statements:

IMPLICIT statements

Type specification statements

EQUIVALENCE statements

DIMENSION statements

ROWWISE statements

COMMON statements

DESCRIPTOR statements

DATA statements

PARAMETER statements

SAVE statements

No other statements can appear in the body of a block data subprogram.

The purpose of a block data subprogram is to initialize the values of elements in named common blocks before program execution begins. Elements in the unnamed common block must not be initialized in a block data subprogram.

If any element in a particular common block is initialized in a block data subprogram, a complete set of specification statements for the entire common block must be present, including any type specification, EQUIVALENCE, and DIMENSION statements. Not all of the elements of a common block need be initialized.

A separate block data subprogram is not required for each common block. Different variables and array elements in a common block can be initialized in different program units, but no variable or array element can be initialized more than once.

END STATEMENT FOR BLOCK DATA SUBPROGRAMS

A block data subprogram must end with one END statement. See figure 7-3 for the format of the END statement. An END statement can contain a statement label. The END statement must not be continued.

BLOCK DATA SUBPROGRAM EXAMPLE

See figure 7-19 for an example of a block data subprogram. The block data subprogram CITIES initializes part of common block EAST and all of common block MIDWEST. The block data subprogram STATES initializes the remainder of common block EAST and all of common block US.

```
PROGRAM BLOCK
IMPLICIT REAL(A-Z)
COMMON /US/ CA,OH,MN
COMMON /EAST/ NY,WASH,BOSTON
COMMON /MIDWEST/ DETROIT,TOLEDO,CHICAGO
.
.
.
END

BLOCK DATA CITIES
IMPLICIT REAL(A-Z)
COMMON /EAST/ NY,WASH,BOSTON
COMMON /MIDWEST/ DETROIT,TOLEDO,CHICAGO
DATA BOSTON,DETROIT,TOLEDO,CHICAGO /4*0.0/
END

BLOCK DATA STATES
IMPLICIT REAL(A-Z)
COMMON /EAST/ NY,WASH,BOSTON
COMMON /US/ CA,OH,MN
DATA NY,WASH /2*5.0/
DATA CA,OH,MN /3*10.0/
END
```

Figure 7-19. BLOCK DATA Statement Examples

STATEMENT FUNCTIONS

A statement function is a single statement that defines the rules for computing a value. A statement function can be referenced anywhere in the program unit in which the statement function is defined. A statement function reference specifies the arguments that are passed to the statement function.

DEFINING STATEMENT FUNCTIONS

A statement function is defined by using a statement function definition. See figure 7-20 for the format of a statement function definition.

```
sfname (darg1, ... ,dargn) = expression

sfname     A symbolic name that is used as the
           name of the statement function.

dargi      A dummy argument, which can be a
           simple variable; optional. No two
           dummy arguments can have the same
           name. The parentheses are required
           even if no dummy argument appear.
```

Figure 7-20. Statement Function Definition Format

A statement function must be defined before the first executable statement in the program unit in which the statement function is referenced. A statement function must be defined after all non-executable statements except DATA, FORMAT, and ENTRY statements.

The names of dummy arguments can be the same as other symbolic names used in the program unit, such as other variable names. The expression that appears in a statement function definition can contain constants, symbolic constants, variables, array elements, substrings, and function references in addition to the dummy arguments of the statement function. If the expression contains other statement function references, the statement functions that are referenced must be defined before the statement function that contains the statement function reference.

Recursive statement function references are not permitted. A recursive statement function reference is a statement function reference that directly or indirectly references the statement function in which it appears.

The type of a statement function result depends on the type of the statement function name. The data type with which a statement function name is associated is specified by a type specification statement or by the first letter of the statement function name. See section 3 for a description of type specification.

The result of the expression in a statement function is converted to the type of the statement function name; the rules for type conversion in a statement function are the same as the rules for type conversion during assignment. See section 4 for a description of type conversion during assignment.

A statement function name must not appear in an EQUIVALENCE, COMMON, EXTERNAL, or INTRINSIC statement. A statement function name must not be dimensioned or initialized.

REFERENCING STATEMENT FUNCTIONS

A statement function is referenced by using a statement function reference. See figure 7-21 for the format of a statement function reference.

A statement function reference can appear in an arithmetic, logical, or character expression.

$sfname\ (aarg_1, \ldots, aarg_n)$

sfname	The name of a statement function that is defined in the program unit in which the statement function reference appears.
$aarg_i$	An actual argument, which can be a scalar expression of the same data type as the corresponding dummy argument; optional. The actual arguments must agree in number and order with the dummy arguments. The parentheses are required even if no actual arguments appear.

Figure 7-21. Statement Function Reference Format

A statement function is evaluated during the evaluation of the expression in which the statement function reference appears. The values that the actual arguments have at the time the statement function is evaluated are the values that are used during evaluation of the statement function.

STATEMENT FUNCTION EXAMPLE

See figure 7-22 for an example of a statement function. The first statement function in the example is called SQUARE and has one dummy argument, B, which is real. The second statement function in the example is called HEIGHT and has one dummy argument, B, which is real. Statement function HEIGHT references the intrinsic function SQRT and the statement function SQUARE.

```
PROGRAM SFUNC
IMPLICIT REAL(A-Z)
SQUARE(B) = B**2
HEIGHT(B) = SQRT(SQUARE(B) - SQUARE(B/2))
.
.
.
ANYTRI = (BASE * HEIGHT(BASE))/2
.
.
.
END
```

Figure 7-22. Statement Function Example

ARRAY ASSIGNMENT STATEMENTS 8

An array assignment statement is an executable statement that assigns the results of array expressions to array elements. A single array assignment statement can assign different values to different array elements.

In order to assign different values to different array elements using scalar assignment statements, you would have to write a DO loop. An array assignment statement could be used instead of the DO loop; therefore, an array assignment statement can be thought of as a shorthand notation for a FORTRAN DO loop.

Using an array assignment statement does not prohibit vectorization. If the DO loop equivalent of an array assignment statement is vectorizable, the array assignment statement can be compiled into vector machine instructions. In order to do this, you must specify the V compilation option on the FORTRAN control statement. See section 9 for a description of vectorizable DO loops.

Array assignment statements are neither a part of the standard FORTRAN language nor a part of the vector programming features of the FORTRAN 200 language.

This section describes subarray references, conformable subarrays, array expressions, and array assignment statements.

SUBARRAY REFERENCES

A subarray reference is a reference to a portion of an array. A subarray reference can be a reference to one element, several elements, or all elements of an array.

A subarray reference consists of an array name and a subscript. The subscript must contain at least one implied DO subscript expression. The subscript can also contain standard subscript expressions, which are described in section 2.

An implied DO subscript expression can have one of three formats. See figure 8-1 for the formats of implied DO subscript expressions.

The first form indicates subscript expression values from $avar_1$ through $avar_2$ starting with $avar_1$ and incrementing by $avar_3$.

The second form is equivalent to the first form where $avar_1$ and $avar_2$ are the lower and upper bounds of the corresponding dimension respectively, and $avar_3$ is 1.

$avar_1$:$avar_2$:$avar_3$
or
*
or
$avar_1$:*:$avar_3$

$avar_1$	An integer constant or simple integer variable; $avar_1$ is the initial value of the subscript expression.
$avar_2$	An integer constant or simple integer variable; $avar_2$ is the terminal value of the subscript expression.
$avar_3$	An integer constant or simple integer variable; optional; $avar_3$ is the incrementation value for the subscript expression. If $avar_3$ is not specified, the colon that precedes it must not appear. The default is 1.
*	When used in the second format, the asterisk represents lower:upper; when used in the third format, the asterisk represents upper; lower and upper are the lower bound and upper bound of the corresponding dimension, respectively.

Figure 8-1. Implied DO Subscript Expression Format

The third form is equivalent to the first form where $avar_2$ is the upper bound of the corresponding dimension.

The second and third forms must not be used for the last subscript of an assumed-size columnwise array. The second and third forms must not be used for the first subscript of an assumed-size rowwise array.

If the value of $(avar_2-avar_1)/avar_3$ is not integral, the subscript expression is never equal to the terminal value $avar_2$. There are no restrictions on the values of $avar_1$, $avar_2$, and $avar_3$. The value of $avar_3$ can be negative, $avar_1$ can be greater than $avar_2$, or both. If $avar_3$ is negative and $avar_1$ is less than $avar_2$, or if $avar_3$ is positive and $avar_1$ is greater than $avar_2$, the subarray reference contains no elements.

An implied DO subscript expression can appear only in an array expression. An array expression can appear only in an array assignment statement.

The order of the elements of a subarray reference is always columnwise: the leftmost implied DO subscript expression varies most rapidly. This rule applies to all subarray references, regardless of whether the array is declared in a DIMENSION statement or in a ROWWISE statement. See figure 8-2 for an example that illustrates the order of the elements of a subarray reference.

The order of the elements of a subarray reference does not have to be the same as the order of the the elements of the array itself. A subarray reference can involve a columnwise array or a rowwise array. The order of elements of the array itself determines whether the elements of the subarray reference are located in consecutive memory locations.

See figure 8-3 for an illustration of the differences between a subarray reference involving a columnwise array and a subarray reference involving a rowwise array.

Declaration:

```
DIMENSION A(2,2,2)
   or
ROWWISE A(2,2,2)
```

Subarray Reference	Order of Subarray Elements
A(*,*,1)	A(1,1,1) A(2,1,1) A(1,2,1) A(2,2,1)
A(1:2,1,1:2)	A(1,1,1) A(2,1,1) A(1,1,2) A(2,1,2)

Figure 8-2. Order of Subarray Elements Example

Declaration:

```
DIMENSION I(5,2)
DATA I /0,1,2,3,4,5,6,7,8,9/
```

Order of array elements:

I(1,1)	I(2,1)	I(3,1)	I(4,1)	I(5,1)	I(1,2)	I(2,2)	I(3,2)	I(4,2)	I(5,2)
0	1	2	3	4	5	6	7	8	9

Declaration:

```
ROWWISE I(5,2)
DATA I /0,1,2,3,4,5,6,7,8,9/
```

Order of array elements:

I(1,1)	I(1,2)	I(2,1)	I(2,2)	I(3,1)	I(3,2)	I(4,1)	I(4,2)	I(5,1)	I(5,2)
0	1	2	3	4	5	6	7	8	9

Subarray Reference	Order of Subarray Elements	Values of Subarray Reference for DIMENSION	Values of Subarray Reference for ROWWISE
I(1:3,*)	I(1,1)	0	0
	I(2,1)	1	2
	I(3,1)	2	4
	I(1,2)	5	1
	I(2,2)	6	3
	I(3,2)	7	5

Figure 8-3. Subarray References Using Columnwise and Rowwise Arrays Example

CONFORMABLE SUBARRAY REFERENCES

Two subarray references are conformable subarray references if they satisfy the following conditions:

The number of implied DO subscript expressions in each subarray reference must be the same. The number of standard subscript expressions in each subarray reference has no effect on the conformability of two subarray references.

The first implied DO subscript expression in one subarray reference must be the same as the first implied DO subscript expression in the other subarray reference, the second implied DO subscript expression in one subarray reference must be the same as the second implied DO subscript expression in the other subarray reference, and so on. The appearance of standard subscript expressions in the subarray references has no effect on the conformability of two subarray references. Two implied DO subscripts are the same if their initial values, terminal values, and incrementation values are the same.

Two subarray references need not have the same number of standard subscripts nor be of the same data type in order to be conformable subarray references.

The number of elements of a subarray reference is the same as the number of elements of any subarray reference that is conformable with it.

See figure 8-4 for examples of conformable and nonconformable subarray references.

```
Declaration:

    DIMENSION A(5,3),B(8,5),C(5,3,4)


Conformable Subarray References:

    A                and    A
    A(1:5,3)         and    B(1:5:1,2)
    A(1:5,3)         and    B(1,1:5)
    A(1:4,3)         and    C(1,2,1:4)
    A                and    B(1:5,1:3)
    A(1:5,1:3)       and    C(1:5,2,1:3)
    A(1:5:2,2)       and    B(1:5:2,4)


Nonconformable Subarray References:

    A                and    B
    B(1:5,1:3)       and    B(1:3,1:5)
    A(1:4,3)         and    C(1:1,2,1:4)
    A(1:5,1:3)       and    B(1:3,1:5)
```

Figure 8-4. Conformable and Nonconformable Subarray References Examples

ARRAY EXPRESSIONS

An array expression is the same as a scalar expression, except an array expression must contain at least one subarray reference. Any two subarray references in an array expression must be conformable subarray references.

When an array expression is evaluated, the operations specified are performed on corresponding elements of the array operands. Any scalar entities that appear in an expression are treated as if they were arrays that have the same number of elements as the subarray references in the expression, and as if the values of those elements were the value of the scalar entity.

An array expression can appear only in an array assignment statement.

See figure 8-5 for examples of array expressions.

```
Declaration:

    DIMENSION A(5,5),B(10,5),C(5,10)


Array Expressions:

    A + 3.1
    A(1:3,1) * A(1:3,2)/A(1:3,3) * A(1:3,4)
    A(I,1:5) ** 2.0
    B(10,1:5) + C(1:5,10) + 1.0 - A(1,1)
    (A - B(1:5,*))/24.5 * C(*,1:5)
```

Figure 8-5. Array Expressions Examples

ARRAY ASSIGNMENT STATEMENT

An array assignment statement consists of a subarray reference and an array expression. See figure 8-6 for the format of an array assignment statement.

```
suba = arexp

suba     A subarray reference that is conformable with the result of arexp

arexp    An array expression or any scalar expression
```

Figure 8-6. Array Assignment Statement Format

If a scalar expression is specified in an array assignment statement, the elements of the subarray reference are replaced with the scalar value.

If an array expression is specified in an array assignment statement, the values of the elements of the subarray reference are replaced with the values of the corresponding elements in the array expression result.

The data type conversion rules for scalar assignment statements apply to array assignment statements. See section 4 for a description of the data type conversion rules for assignment statements.

See figure 8-7 for examples of array assignment statements.

Each of the statement pairs:

```
      DIMENSION X(5,3),Y(2,5)
      X(1:5,3) = Y(2,1:5)

      DIMENSION X(5,3),Y(2,5)
      X(*,3) = Y(2,*)
```

has the same effect as the statements:

```
      DIMENSION X(5,3),Y(2,5)
      DO 100 I=1,5,1
      X(I,3) = Y(2,I)
  100 CONTINUE
```

which accomplishes the following assignments:

```
      X(1,3) = Y(2,1)
      X(2,3) = Y(2,2)
      X(3,3) = Y(2,3)
      X(4,3) = Y(2,4)
      X(5,3) = Y(2,5)
```

The statement pair:

```
      DIMENSION X(5,3), Y(10,3,2)
      X(1:*:3,*) = Y(1:5:3,*,2)
```

has the same effect as the statements:

```
      DIMENSION X(5,3),Y(10,3,2)
      DO 200 I2=1,3,1
      DO 100 I1=1,5,3
      X(I1,I2) = Y(I1,I2,2)
  100 CONTINUE
  200 CONTINUE
```

which accomplishes the following assignments:

```
      X(1,1) = Y(1,1,2)
      X(4,1) = Y(4,1,2)
      X(1,2) = Y(1,2,2)
      X(4,2) = Y(4,2,2)
      X(1,3) = Y(1,3,2)
      X(4,3) = Y(4,3,2)
```

Figure 8-7. Array Assignment Statement Examples

VECTOR PROGRAMMING 9

The FORTRAN 200 compiler provides a set of features that enables you to use the vector processing hardware of the CYBER 200 computer. These features are extensions to the standard FORTRAN language.

Use of the vector programming features can decrease the amount of time needed to execute a program. This section describes the vector programming features of the FORTRAN 200 language.

OVERVIEW

The FORTRAN 200 compiler translates FORTRAN statements into machine language instructions. Two types of machine language instructions can be generated by the compiler and executed by the CYBER 200 computer: scalar machine instructions and vector machine instructions.

There are three distinct steps in the performance of an operation. The steps are:

Values are loaded from memory.

An operation is performed on the values.

The result is stored back into memory.

A scalar machine instruction is a machine instruction that performs an operation on a single set of values. Thus, each of the three steps is performed in order and cannot be overlapped.

A vector machine instruction is a machine instruction that performs an operation on a stream of values. The stream of values is called a vector. The values in a vector must be located in contiguous memory locations.

Execution of a vector machine instruction involves the same three steps as for a scalar machine instruction; however, the steps can be overlapped. While the result of one operation is being stored back into memory, the operation can be performed on another set of values; while the operation is being performed on that set of values, another set of values can be loaded from memory. Thus, the central processor does not have to wait for values to be loaded and stored.

When the same operation is to be performed on a series of operands, a vector machine instruction can perform the operation faster than a series of scalar machine instructions. The more sets of operands there are, the more efficient the vector machine instruction is. See figure 9-1 for an illustration of the difference between scalar machine instructions and vector machine instructions. (Figure 9-1 is intended to illustrate the concept of vector processing, and does not necessarily represent the actual implementation or the time proportions between scalar and vector processing.)

The CYBER 200 central processor consists of two parts: a scalar instruction processor and a vector instruction processor. The scalar processor executes scalar instructions; the vector processor executes vector instructions. Both processors can execute instructions simultaneously as long as the instructions do not conflict. See the appropriate hardware reference manual for a detailed description of the vector processing hardware.

Vector machine instructions can be generated by the compiler when you do either or both of the following:

Use vector programming statements, such as vector assignment statements, or vector function references

Specify the optimize V compilation option in the FTN200 control statement, which causes the compiler to generate vector machine instructions for certain types of DO loops

VECTORS AND DESCRIPTORS

A vector is a series of values that are stored in contiguous memory locations. The first element of a vector must be an array element.

Vectors are referenced by using vector references or descriptors. A vector reference or a descriptor specifies the following information:

The first element of the vector, which must be an array element

The length of the vector

The data type of the vector

Vector references specify this information explicitly. Descriptors can be thought of as pointers to vectors. Vector references and descriptors are described in the following paragraphs.

VECTOR REFERENCES

A vector reference explicitly specifies the first element of the vector, the length of the vector, and the data type of the vector. See figure 9-2 for the format of a vector reference.

The first element of a vector must be an array element; however, a vector has no other relation to the array.

The order of elements of the array affects the order of elements in a vector. See section 2 for more information about array storage.

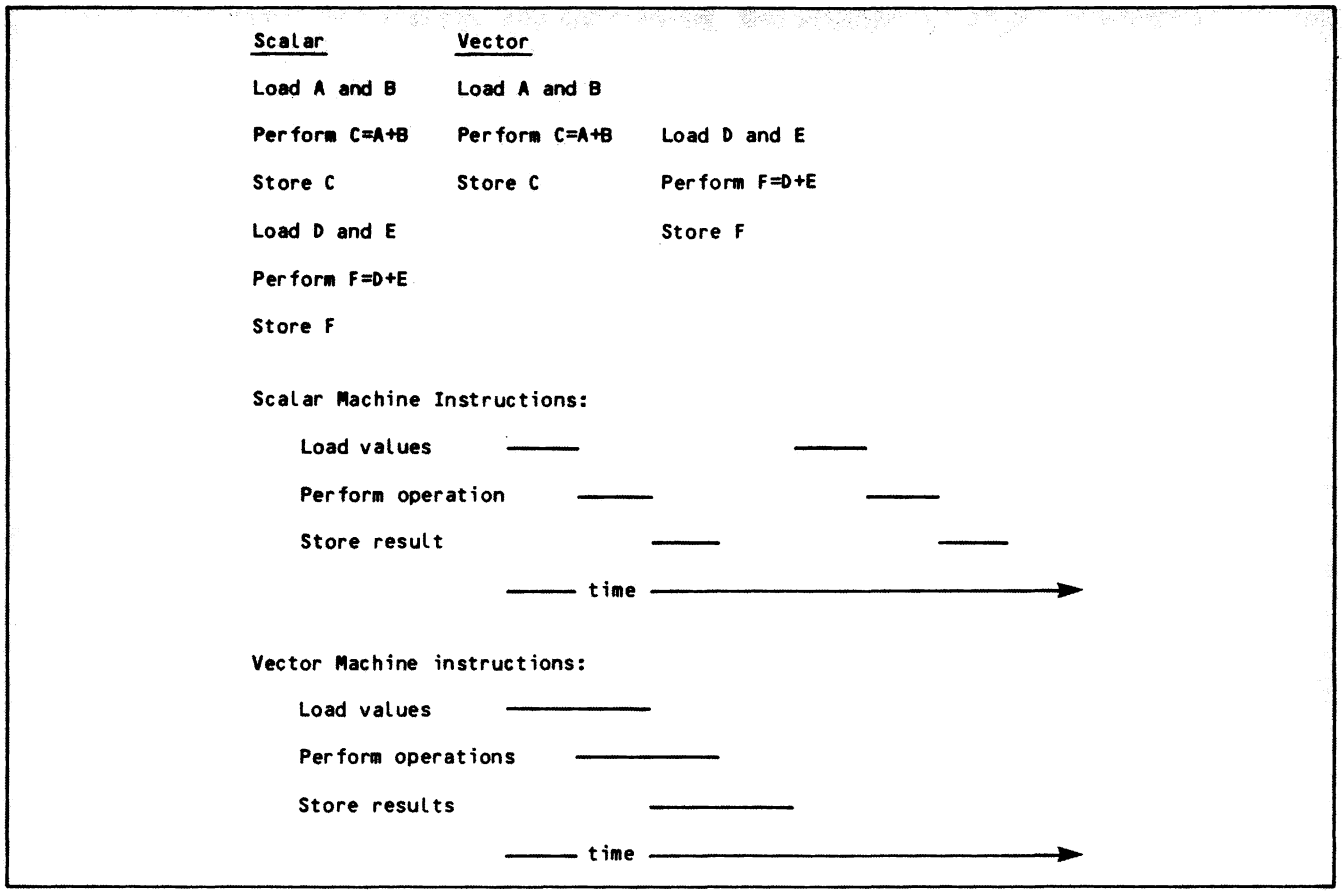


Figure 9-1. Scalar vs. Vector Processing Illustration

aname(base;len)

aname	The name of an array of type integer, half-precision, real, double-precision, complex, or bit.
base	A list of subscript expressions separated by commas; base indicates the element of array aname that is the first element of the vector. The number of subscript expressions in base must be the same as the number of dimensions in array aname. The value of each subscript expression must not be less than the lower bound of the dimension or greater than the upper bound of the dimension. If an expression in base is not integer, the value is truncated to integer.
len	A scalar expression of type integer whose result is positive; len specifies the number of elements in the vector. The length must not exceed 65535 elements for vectors of type integer, half-precision, real, or bit; the length must not exceed 32767 elements for vectors of type double-precision or complex.

Figure 9-2. Vector Reference Format

An array element can be an element of more than one vector. An array element that is part of a vector can be modified as though it were not part of any vector. The array elements that are part of a vector can be input and output. You can place a vector reference in an input/output statement. You can place a vector reference in a DATA statement in order to initialize the elements of the vector. Initialization of vectors is described later in this section.

The data type of a vector is the data type of the array element that is used to specify the first element of the vector. Vectors can be integer, half-precision, real, double-precision, complex, and bit. Vectors of type double-precision and complex are highly restricted.

See figure 9-3 for examples of vector references.

DESCRIPTORS

Each vector that is defined in a program unit requires at least one vector reference; however, once you define the vector by using a vector reference, you can reference that vector symbolically by using a descriptor.

A descriptor is a symbolic name that represents a vector. The symbolic name must be declared in a DESCRIPTOR statement. The DESCRIPTOR statement is described later in this section. The symbolic name must be associated with the same data type as the

```
Declarations:

    DIMENSION A(10),B(10,10)
    ROWWISE C(10,10)

Vector reference        Vector

A(3;5)                  A(3)
                        A(4)
                        A(5)
                        A(6)
                        A(7)

B(1,1;5)                B(1,1)
                        B(2,1)
                        B(3,1)
                        B(4,1)
                        B(5,1)

C(1,1;5)                C(1,1)
                        C(1,2)
                        C(1,3)
                        C(1,4)
                        C(1,5)
```

Figure 9-3. Vector Reference Examples

data type of the vector that it represents; however, the symbolic name cannot be associated with the double-precision data type. Double-precision vectors can be defined and referenced by using vector references only.

A descriptor must be associated with a vector before the descriptor can be used. A descriptor can be associated with a vector by using a DATA statement or a descriptor ASSIGN statement.

A DATA statement associates a descriptor with a vector before program execution begins. When a descriptor appears in the variable list of a DATA statement, the descriptor is initialized rather than the vector that it references. Initialization of descriptors is described later in this section.

The descriptor ASSIGN statement is an executable statement that associates a descriptor with a vector during program execution. Thus, any subsequent reference to the descriptor references the vector with which it is associated. A descriptor can be redefined by using other descriptor ASSIGN statements. The descriptor ASSIGN statement is described later in this section.

See figure 9-4 for the internal format of a descriptor. The vector length of an integer, real, or complex vector is the number of fullwords the vector occupies. The vector length of a half-precision vector is the number of halfwords the vector occupies. The vector length of a bit vector is the number of bits the vector occupies.

See figure 9-5 for examples of descriptors. The DESCRIPTOR statement in the example declares MYVEC and YOURVEC to be descriptors. MYVEC is an integer descriptor; an integer vector must be associated with MYVEC before MYVEC can be used. YOURVEC is a real descriptor; a real vector must be associated with YOURVEC before YOURVEC can be used.

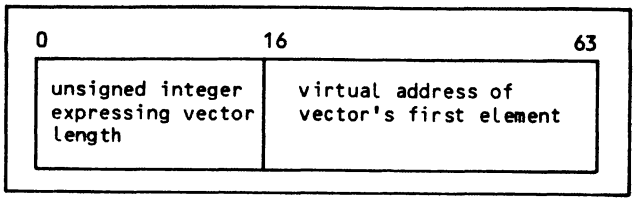


Figure 9-4. Descriptor Representation

```
.
.
.
INTEGER MYVEC
REAL YOURVEC
DESCRIPTOR MYVEC,YOURVEC
.
.
.
```

Figure 9-5. Descriptor Examples

DESCRIPTOR Statement

The DESCRIPTOR statement is a nonexecutable specification statement that declares a symbolic name to be a descriptor. See figure 9-6 for the format of the DESCRIPTOR statement.

DESCRIPTOR $v_1, \ldots, v_n$

v_i A variable name, array name, or array declarator of type integer, half-precision, real, complex, or bit

Figure 9-6. DESCRIPTOR Statement Format

A symbolic name that appears in a DESCRIPTOR statement is associated with a data type. The data type with which it is associated is determined by the first letter of the symbolic name or by a type specification statement. A symbolic name that appears in a DESCRIPTOR statement can be associated with any of the following data types: integer, half-precision, real, complex, or bit. Although vectors can be associated with the double-precision data type, descriptors cannot.

The DESCRIPTOR statement can be used to declare a descriptor array.

See figure 9-7 for an example of the DESCRIPTOR statement. The DESCRIPTOR statement in the example specifies that A is a descriptor, B is a descriptor array of five elements, and C is a descriptor array of 10 elements.

Descriptor Arrays

Descriptors can be organized into arrays by using the DESCRIPTOR, DIMENSION, or ROWWISE statement, or by using a type specification statement. See figure 9-6 for the format of the DESCRIPTOR statement.

```
.
.
.
INTEGER A
REAL B(5)
DESCRIPTOR A,B,C(10)
.
.
.
```

Figure 9-7. DESCRIPTOR Statement Example

Descriptor arrays are referenced in the same way as other arrays: by specifying the descriptor array name and a subscript.

Initializing Vectors and Descriptors

The nonexecutable DATA statement can be used to initialize vectors and descriptors. Double-precision vectors cannot be initialized, and descriptors cannot be associated with the double-precision data type. See section 3 for a description of the DATA statement.

Initializing a vector causes the values you specify to be assigned to the vector elements before program execution begins. In order to initialize a vector, place a vector reference in the variable list of the DATA statement. The vector reference must contain integer constants for the subscript expression and the length specification.

Place the initial values for the vector in the value list of the DATA statement. The number of constants in the value list must be the same as the number of elements in the vector.

See figure 9-8 for an example of vector initialization. The first DATA statement in the example initializes the elements of vector A(1;5) to 1.0, 2.0, 3.0, 4.0, and 5.0 respectively. The second DATA statement in the example initializes each element of vector B(1;5) to 0.0.

```
.
.
.
REAL A(5),B(10)
DATA A(1;5) /1.0,2.0,3.0,4.0,5.0/
DATA B(1;5) /5*0.0/
.
.
.
```

Figure 9-8. Vector Initialization Example

Initializing a descriptor causes the descriptor to be associated with a vector before execution of the program. Initializing a descriptor does not cause values to be assigned to the vector itself. In order to initialize a descriptor, place the descriptor name in the variable list of a DATA statement. The symbolic name must be declared in a DESCRIPTOR statement.

Place a vector reference in the value list of the DATA statement. The vector reference must contain integer constants for the subscript expression and length specification. The number of vector references in the value list must be the same as the number of descriptors in the variable list.

The data type of the descriptor must be the same as the data type of the vector with which the descriptor is associated.

The repeat specification can be used in the value list of the DATA statement in order to repeat the use of a vector reference for initialization of more than one descriptor.

Descriptor arrays and descriptor array elements can be initialized by a DATA statement in the same way.

See figure 9-9 for an example of descriptor initialization. The first DATA statement in the example associates the elements of the descriptor array V1 with the vectors A(1;5), B(1;5), C(1;7), C(1;7), and C(1;7) respectively. The second DATA statement in the example associates the descriptors V2 and V3 with the vector C(8;2).

```
.
.
.
REAL A(5),B(10),C(10)
REAL V1(5),V2,V3
DESCRIPTOR V1,V2,V3
DATA V1 /A(1;5),B(1;5),3*C(1;7)/
DATA V2,V3 /2*C(8;2)/
.
.
.
```

Figure 9-9. Descriptor Initialization Example

Descriptor ASSIGN Statement

The descriptor ASSIGN statement is an executable statement that associates a descriptor with a vector. See figure 9-10 for the format of the descriptor ASSIGN statement.

Execution of a descriptor ASSIGN statement of the first form associates the descriptor p with the vector referenced by q.

See figure 9-11 for examples of descriptor ASSIGN statements of the first form. The first descriptor ASSIGN statement in the example associates the first element of the descriptor array V1 with the vector A(1;5). The second descriptor ASSIGN statement associates the second element of the descriptor array V1 with the first element of descriptor array V1, which is the vector A(1;5). The third descriptor ASSIGN statement associates the descriptor V2 with the vector C(8;2).

Execution of a descriptor ASSIGN statement of the second form causes storage to be dynamically allocated for a vector and causes the descriptor specified in the descriptor ASSIGN statement to be associated with that vector.

```
ASSIGN p,q
    or
ASSIGN p,.DYN. aexp
```

p	A descriptor or descriptor array element of type integer, half-precision, real, complex, or bit. The data type of p must be the same as the data type of q.
q	A vector reference, or a previously-assigned descriptor or descriptor array element; q can be of type integer, half-precision, real, complex, or bit. The data type of q must be the same as the data type of p.
aexp	A scalar expression of type integer; aexp indicates the number of halfwords, fullwords, doublewords, or bits to be allocated for the vector when the descriptor ASSIGN statement is executed.

Figure 9-10. Descriptor ASSIGN Statement Format

```
          .
          .
          .
          REAL A(5),C(10)
          REAL V1,V2
          DESCRIPTOR V1(5),V2
          .
          .
          .
          ASSIGN V1(1),A(1;5)
          ASSIGN V1(2),V1(1)
          ASSIGN V2,C(8;2)
          .
          .
          .
          ASSIGN V1(3),.DYN.100
          ASSIGN V1(4),.DYN.250
          .
          .
          .
          FREE
          .
          .
          .
```

Figure 9-11. Descriptor ASSIGN and FREE Statement Examples

Normally, storage for vectors is allocated at compilation time. However, by using a descriptor ASSIGN statement of the second form, you can allocate storage for a vector at execution time.

Storage is allocated in units that are dependent on the type of descriptor: fullwords are allocated for integer and real descriptors; halfwords are allocated for half-precision descriptors; doublewords are allocated for complex descriptors; bits are allocated for bit descriptors.

Storage allocated by using a descriptor ASSIGN statement of the second form is available until a FREE or RETURN statement is executed. A FREE or RETURN statement returns storage allocated by all descriptor ASSIGN statements in the program unit.

See figure 9-11 for examples of descriptor ASSIGN statements of the second form. The fourth descriptor ASSIGN statement in the example allocates storage for a 100-fullword vector; that vector can be referenced by using the descriptor array element V1(3). The fifth descriptor ASSIGN statement allocates storage for a 250-fullword vector; that vector can be referenced by using the descriptor array element V1(4).

FREE Statement

The FREE statement returns all storage that is allocated by descriptor ASSIGN statements containing .DYN. in the program unit in which the FREE statement appears. See figure 9-12 for the format of the FREE statement.

```
FREE
```

Figure 9-12. FREE Statement Format

Also, storage allocated by descriptor ASSIGN statements containing .DYN. is automatically returned after execution of each program unit is completed.

See figure 9-11 for an example of the FREE statement. The FREE statement in the example returns all storage allocated by the fourth and fifth descriptor ASSIGN statements. Thus, 350 fullwords are returned. Descriptor array elements V1(3) and V1(4) no longer reference that storage and must not be referenced after the FREE statement unless they are first associated with a vector.

BIT DATA TYPE

The vector programming features of the FORTRAN 200 language include the bit data type. Language elements that are associated with the bit data type are used in bit expressions and bit assignment statements. Also, vector relational expressions yield vectors of type bit.

The language elements that can be associated with the bit data type include constants, variables, arrays, function names, and descriptors.

Bit vectors are useful in controlling the evaluation of vector expressions in WHERE statements and block WHERE structures. The WHERE statement and block WHERE structures are described later in this section.

The following paragraphs describe the bit data type.

BIT CONSTANTS

A bit constant is a string of binary digits enclosed in apostrophes and preceded by the letter B. See figure 9-13 for the format of a bit constant. See figure 9-14 for examples of legal and illegal bit constants.

```
B'bin-digits'

bin-digits   A string of 1 through 255 of the
             binary digits 0 and 1. The binary
             digits correspond to the decimal
             values 0 and 1.
```

Figure 9-13. Bit Constants Format

```
Legal bit constants:

    B'0'
    B'1001001001'
    B'11111'

Illegal bit constants:

    B''
    B'1957'
    1010010B
```

Figure 9-14. Bit Constants Examples

BIT VARIABLES AND ARRAYS

Variables and arrays can be associated with the bit data type. In order to associate a variable or array with the bit data type, you must declare the variable name or array name in a BIT or IMPLICIT statement.

BIT ELEMENT REPRESENTATION

A bit constant occupies 1 bit for each binary digit in the bit value. The binary digits in a bit constant are stored in consecutive bits. The word boundary is not significant in bit elements.

BIT STATEMENT

The BIT statement can be used to associate a list of variable names, array names, and function names with the bit data type. The BIT statement can also be used to initialize bit variables and entire bit arrays. See figure 9-15 for the format of the BIT statement.

See figure 9-16 for an example of the BIT statement. The BIT statement in the example associates SEAL and WALRUS with the bit data type. The BIT statement also declares WALRUS to be an array of two elements, and intializes SEAL to B´1´, WALRUS(1) to B´1´, and WALRUS(2) to B´0´.

```
BIT v1/d1/, ... ,vn/dn/

vi    A variable name, array name, symbolic
      constant name, array declarator, or
      function name.

di    A bit constant; optional. This speci-
      fies an initial value for vi. If vi
      is a function name, di must be
      omitted. If di is omitted, the
      surrounding slashes must be omitted.
```

Figure 9-15. BIT Statement Format

```
      .
      .
      .
      BIT SEAL/B'1'/,WALRUS(2)/B'1',B'0'/
      .
      .
      .
```

Figure 9-16. BIT Statement Example

INITIALIZING BIT ITEMS

Bit variables and entire bit arrays can be initialized by using the BIT statement. Bit variables, arrays, and array elements can be initialized by using the DATA statement or the BIT statement. The rules for initializing variables and arrays of type bit are:

If you specify a bit variable or a bit array element in the list of variable names and array names, you must specify one 1-bit constant for each bit variable or bit array element.

If you specify a bit array name or an implied DO loop in the list of variable names and array names, you can provide one or more constants whose total length is equal to the length of the contiguous portion of the bit array that is to be initialized. The length of each constant must equal the length of the contiguous portion of the bit array to which it corresponds.

Dummy arguments must not be initialized.

Variables and arrays that are in an unnamed common block must not be initialized.

See figure 9-17 for examples of initialization of bit variables and arrays using the DATA statement. The first DATA statement in the example initializes BVAR1 and BARR1(1) to B´0´ and B´1´ respectively. The second DATA statement initializes the elements of array BARR2 to B´1´, B´1´, B´0´, B´1´, B´0´, B´0´, B´0´, B´1´, B´0´, and B´1´ respectively. The third DATA statement initializes both BARR3(1) and BARR3(6) to B´0´. The fourth DATA statement initializes BARR4(1,1), BARR4(2,1), BARR4(1,4), and BARR4(2,4) to B´0´, B´1´, B´0´, and B´1´ respectively. The fifth DATA statement initializes the elements of array BARR5 to B´1´, B´1´, B´1´, B´1´, B´0´, B´0´, B´0´, and B´0´ respectively.

```
.
.
.
BIT BVAR1
BIT BARR1(10),BARR2(10),BARR3(10),BARR4(5,5),BARR5(8)
DATA BVAR1,BARR1(1) /B'0',B'1'/
DATA BARR2 /B'1101000101'/
DATA (BARR3(I),I=1,10,5) /2*B'0'/
DATA ((BARR4(I,J),I=1,2),J=1,5,3) /2*B'01'/
DATA (BARR5(I),I=1,8) /X'F0'/
.
.
.
```

Figure 9-17. Initialization of Bit Items Examples

VECTOR EXPRESSIONS

A vector expression is a string of operators and operands that defines the rules for computing the values of vector elements. At least one of the operands must be a vector reference, descriptor, descriptor array element, or vector function reference. A vector expression can contain scalar data elements. A vector expression is evaluated during program execution. There are three kinds of vector expressions:

Vector arithmetic expressions

Vector relational expressions

Bit expressions

Vector expressions are described in the following paragraphs.

VECTOR ARITHMETIC EXPRESSIONS

A vector arithmetic expression is a vector expression that yields a numeric vector. A vector arithmetic expression can appear in a vector arithmetic assignment statement or in a vector relational expression. See figure 9-18 for the format of a vector arithmetic expression.

The operators that can be used in a vector arithmetic expression are the arithmetic operators +, -, *, /, and **; however, there are two restrictions:

The exponentiation operator must not appear in a vector arithmetic expression that yields a complex result.

Arithmetic operators must not appear in a vector arithmetic expression that yields a double-precision result. (A double-precision vector arithmetic expression must consist of a double-precision vector reference or a double-precision vector function reference, and must appear only in a vector arithmetic assignment statement of type double-precision.)

See section 4 for a description of the arithmetic operators.

```
vaexp

vaexp    A vector arithmetic expression of one
         of the forms:

             term
             + term
             - term
             vaexp + term
             vaexp - term

term     An arithmetic term of one of the forms:

             fact
             term * fact
             term/fact

fact     An arithmetic factor of one of the
         forms:

             prim
             prim ** fact

prim     An arithmetic primary. An arithmetic
         primary can be an unsigned arithmetic
         constant, arithmetic symbolic constant,
         arithmetic variable, arithmetic array
         element, scalar arithmetic function
         reference, scalar arithmetic expression
         enclosed in parentheses, vector refer-
         ence, descriptor, descriptor array
         element, vector function reference, or
         vector arithmetic expression enclosed
         in parentheses.
```

Figure 9-18. Vector Arithmetic Expression Format

The order in which a vector arithmetic expression is evaluated is the same as the order in which a scalar arithmetic expression is evaluated. The order of evaluation of expressions is described in section 4.

The operands that can appear in a vector arithmetic expression can be scalar operands and vector operands. The scalar operands can be constants, symbolic constants, variables, array elements, and

scalar function references. The vector operands can be vector references, descriptors, descriptor array elements, and vector function references. A vector arithmetic expression must contain at least one vector operand.

The data type of the operands that appear in a vector arithmetic expression can be integer, half-precision, real, double-precision, or complex; however, the following restrictions apply to vector arithmetic expressions that contain double-precision operands:

The expression must contain no operators.

The expression can appear only in a vector arithmetic assignment statement of type double-precision.

All vector operands that appear in a vector arithmetic expression must have the same length.

When a vector arithmetic expression is evaluated, each operation in the expression is evaluated in the order described in section 4.

Operations that involve two vector operands separated by a binary operator are performed using the corresponding elements of the vector operands.

Operations that involve a scalar operand and a vector operand separated by a binary operator are performed using the scalar operand and each element of the vector operand.

Operations that involve a vector operand and a unary operator are performed using each element of the vector operand.

The results of an operation are placed in a single result vector. The elements of the result vector correspond to the elements of the vector operand. The result vector can be used as a vector operand in subsequent operations.

After all of the operations in a vector arithmetic expression are performed, the result vector is assigned to the vector appearing on the left side of the vector arithmetic assignment statement that contains the expression. Vector arithmetic assignment statements are described later in this section.

See figure 9-19 for examples of vector arithmetic expressions.

```
SD1
SD1 + SD2
SD1 + R * 3.0
SDARR(3)
VSQRT(Q(1;100);SD1)
```

SD1 and SD2 are real descriptors, R is a real scalar variable, SDARR is a real descriptor array, and Q is a real scalar array.

Figure 9-19. Vector Arithmetic Expression Examples

VECTOR RELATIONAL EXPRESSIONS

A vector relational expression is a vector expression that yields a bit vector. A vector relational expression can appear in a bit assignment statement. See figure 9-20 for the format of a vector relational expression.

```
saexp op vaexp
    or
vaexp op saexp
    or
vaexp op vaexp
```

saexp	A scalar arithmetic expression of type integer, half-precision, or real.
vaexp	A vector arithmetic expression of type integer, half-precision, or real.
op	A relational operator

Figure 9-20. Vector Relational Expression Format

The operators that can be used in a vector relational expression are the relational operators .EQ., .NE., .GE., .GT., .LE., and .LT.. The periods are part of the operators and must appear. See section 4 for a description of the relational operators.

The order in which a vector relational expression is evaluated is the same as the order in which a scalar relational expression is evaluated. The order of evaluation of expressions is described in section 4.

The operands that can appear in a vector relational expression can be scalar operands and vector operands. The scalar operands can be scalar arithmetic expressions. The vector operands can be vector arithmetic expressions. A vector relational expression must contain at least one vector operand.

The data type of the operands that appear in a vector relational expression can be integer, half-precision, or real.

All vector operands that appear in a vector relational expression must have the same length.

When a vector relational expression is evaluated, each arithmetic expression in the vector relational expression is evaluated. Evaluation of the arithmetic expressions results in two operands for the vector relational expression. The operands can be two vector operands or one scalar operand and one vector operand.

Vector relational expressions that involve two vector operands are performed using the corresponding elements of the vector operands. The comparison specified by the relational operator is performed for the corresponding elements of the vector operands. The result of each comparison can be either a 1, which means that the relation is true, or a 0, which means that the relation is false.

The results of the comparisons are placed in a single result vector, which is a bit vector. The elements of the resulting bit vector correspond to the elements of the vector operands.

Vector relational expressions that involve a scalar operand and a vector operand are performed using the scalar operand and each element of the vector operand. The comparison specified by the relational operator is performed for the scalar operand and each element of the vector operand. The result of each comparison can be either a 1, which means that the relation is true, or a 0, which means that the relation is false. The results of the comparisons are placed in a single result vector, which is a bit vector. The elements of the resulting bit vector correspond to the elements of the vector operand.

See figure 9-21 for examples of vector relational expressions.

```
5.0 .EQ. A(1;5)
A(1;5) .EQ. 5.0
SALARY .LT. EXPENSES
A(1;5) .NE. B(1;5) + C(1;5) * 2.0


A, B, and C are real arrays, and SALARY and
EXPENSES are real descriptors.
```

Figure 9-21. Vector Relational Expression Examples

BIT EXPRESSIONS

A bit expression is a vector expression that yields a bit vector. A bit expression can appear in a bit assignment statement. See figure 9-22 for the format of a bit expression.

```
bexp
    or
bexp op bexp
    or
.NOT. bexp


bexp    A vector relational expression, vector
        reference of type bit, descriptor of
        type bit, descriptor array element of
        type bit, vector function reference of
        type bit, bit constant, bit variable,
        bit array element, or bit expression
        enclosed in parentheses

op      One of the logical operators .AND.,
        .OR., .XOR., .EQV., or .NEQV.
```

Figure 9-22. Bit Expression Format

The operators that can be used in a bit expression are the logical operators .AND., .OR., .XOR., .EQV., .NEQV., and .NOT.. The periods are part of the operators and must appear. See section 4 for a description of the logical operators.

The order in which a bit expression is evaluated is the same as the order in which a scalar logical expression is evaluated. The order of evaluation of expressions is described in section 4.

The operands that can appear in a bit expression can be scalar operands and vector operands. The scalar operands can be constants, variables, and array elements. The vector operands can be vector references, descriptors, descriptor array elements, vector function references, vector relational expressions, and bit expressions enclosed in parentheses.

The data type of the operands that appear in a bit expression must be associated with the bit data type.

The vector operands of a bit expression must have the same length. If operands of different lengths are used, results are unpredictable. No diagnostic is issued for operands of different lengths.

When a bit expression is evaluated, all vector relational expressions in the bit expression are evaluated. Evaluation of the vector relational expressions results in bit vectors; thus, a logical operation specified by a logical operator is performed on two vector operands of type bit, or a scalar operand of type bit and a vector operand of type bit.

Bit expressions that involve two vector operands are performed using the corresponding elements of the vector operands. The operation specified by the bit operator is performed for the corresponding elements of the vector operands. The result of each operation can be either a 1, which represents the logical value .TRUE., or a 0, which represents the logical value .FALSE.. The results of the logical operations are placed in a single result vector, which is a bit vector. The elements of the resulting bit vector correspond to the elements of the longest vector operand.

Bit expressions that involve a scalar operand and a vector operand are performed using the scalar operand and each element of the vector operand. The operation specified by the logical operator is performed for the scalar operand and each element of the vector operand. The result of each logical operation can be either a 1, which represents the logical value .TRUE., or a 0, which represents the logical value .FALSE.. The results of the operations are placed in a single result vector, which is a bit vector. The elements of the resulting bit vector correspond to the elements of the vector operand.

See figure 9-23 for examples of bit expressions.

```
BVAR
BV1 .OR. BV2
(BV1 .OR. BV2) .OR. BV3
(B1(1;10) .GE. B2(1;10)) .AND. B3(1;10)


BVAR is a bit variable; BV1, BV2, and BV3 are
descriptors of type bit; and B1, B2, and B3
are bit arrays.
```

Figure 9-23. Bit Expression Examples

VECTOR ASSIGNMENT STATEMENTS

A vector assignment statement is a statement that causes the result of a vector expression to be assigned to a vector. A vector assignment statement is performed during program execution. There are two kinds of vector assignment statements:

Vector arithmetic assignment statements

Bit assignment statements

Vector assignment statements are described in the following paragraphs.

VECTOR ARITHMETIC ASSIGNMENT STATEMENTS

A vector arithmetic assignment statement assigns the result of a vector arithmetic expression to a vector. See figure 9-24 for the format of a vector arithmetic assignment statement.

```
v = aexp

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

aexp    A vector arithmetic expression or a
        scalar arithmetic expression
```

Figure 9-24. Vector Arithmetic Assignment Statement Format

If the type of the vector reference, descriptor, or descriptor array element that appears to the left of the equals sign differs from the type of the vector expression on the right of the equals sign, type conversion is performed. The result vector of the vector expression is converted to the type of the vector reference, descriptor, or descriptor array element and replaces the value of the vector. See table 9-1 for the rules for type conversion during vector arithmetic assignment.

If the type of the vector reference, descriptor, or descriptor array element that appears to the left of the equals sign is double-precision, the vector expression on the right of the equals sign must be a vector reference of type double-precision, or a reference to a double-precision vector function.

If the vector expression on the right of the equals sign evaluates to a scalar, that scalar is stored into each element of the vector represented on the left of the equals sign.

If the vector expression on the right of the equals sign evaluates to a vector, the elements of the resulting vector are assigned to the corresponding elements of the vector represented on the left of the equals sign.

The result of the vector expression on the right side of the equal sign must have the same length as the vector on the left side of the equal sign. If the lengths are different, results are unpredictable. No execution-time check is performed to determine if the lengths are equal, and no diagnostic is issued.

See figure 9-25 for examples of vector arithmetic assignment statements.

TABLE 9-1. TYPE CONVERSION FOR VECTOR ARITHMETIC ASSIGNMENT (v = aexp)

Type of v	Expression Result Type				
	Integer	Half-precision	Real	Double-precision	Complex
Integer	No conversion	Convert to real then truncate fractional part	Truncate fractional part	Not allowed	Truncate real part; discard imaginary part
Half-precision	Convert to half-precision	No conversion	Convert to half-precision	Not allowed	Convert real part to half-precision; discard imaginary part
Real	Convert to real	Convert to real	No conversion	Convert to real part	Use real part; discard imaginary
Double-precision	Not allowed	Not allowed	Not allowed	No conversion	Not allowed
Complex	Convert to real for real part; use 0 for imaginary part	Convert to real for real part; use 0 for imaginary part	Use for real part; use 0 for imaginary part	Not allowed	No conversion

```
SD1 = SD1 + SD2
SD1 = Q(1;100) + R * 3.0
SDARR(3) = VSQRT(Q(1;100);SD1)
```

SD1 and SD2 are descriptors of type real, Q is an array of type real, R is a variable of type real, and SDARR is a descriptor array of type real.

Figure 9-25. Vector Arithmetic Assignment Statement Examples

BIT ASSIGNMENT STATEMENTS

A bit assignment statement assigns the result of a vector relational expression to a vector, or assigns the result of a bit expression to a vector. See figure 9-26 for the format of a bit assignment statement.

```
v = bexp
```

v	A vector reference of type bit, or a descriptor or descriptor array element of type bit
bexp	A bit expression

Figure 9-26. Bit Assignment Statement Format

The type of the vector reference, descriptor, or descriptor array element that appears to the left of the equals sign must be the same as the type of the vector expression on the right of the equals sign. Type conversion is not performed.

Execution of a bit assignment statement causes the result of the vector relational expression or bit expression to be assigned to the vector represented on the left of the equals sign.

The vector represented on the left of the equals sign must have the same length as the result of the vector expression. If the lengths are different, results are unpredictable. No execution-time check is performed to determine if the lengths are different, and no diagnostic is issued.

See figure 9-27 for examples of bit assignment statements.

```
B(1;100) = BD
BD = BD .OR. B(1;100)
```

Figure 9-27. Bit Assignment Statement Examples

WHERE STATEMENT

The WHERE statement provides for the execution of one vector assignment statement using a control vector. A control vector is a bit vector that controls the storing of values into a vector. See figure 9-28 for the format of a WHERE statement.

```
WHERE (bexp) vast
```

bexp	A bit expression
vast	A vector assignment statement

Figure 9-28. WHERE Statement Format

All vector operands in the 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 must 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 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.

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. If a value is not assigned to a vector element, data flag branches are disabled for the operations that compute that value. See section 11 for a description of data flag branches.

See figure 9-29 for examples of the WHERE statement. The first WHERE statement in the example causes 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.

The second WHERE statement 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.

```
PROGRAM WHEREX
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'/
.
.
.
WHERE (A(1;5).LT.B(1;5)) C(1;5)=B(1;5)-A(1;5)
.
.
.
WHERE (CA(1;5).AND.CB(1;5)) C(1;5)=A(1;5)+B(1;5)
.
.
.
WHERE (C(1;5).NE.0.0) A(1;5)=B(1;5)/C(1;5)
.
.
.
```

Figure 9-29. WHERE Statement Examples

The third WHERE statement 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, no data flag branch occurs.

BLOCK WHERE STATEMENT

The block WHERE statement is used to define a block WHERE structure. See figure 9-30 for the format of the block WHERE statement.

```
WHERE (bexp)

bexp   A bit expression
```

Figure 9-30. Block WHERE Statement Format

When the block WHERE statement is executed, the 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 can be used in a block WHERE structure 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, rather than a 1. See figure 9-31 for the format of the OTHERWISE statement.

```
OTHERWISE
```

Figure 9-31. OTHERWISE Statement Format

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 terminates a block WHERE structure. Each block WHERE statement must have one corresponding END WHERE statement. See figure 9-32 for the format of the END WHERE statement.

```
END WHERE
```

Figure 9-32. END WHERE Statement Format

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 must 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.

See figure 9-33 for the format of a simple block WHERE structure. When the block WHERE statement is executed, the 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 9-33. 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. 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. If a value is not assigned to a vector element, data flag branches are disabled for the operations that compute that value. See section 11 for a description of data flag branches.
3. Steps 1 and 2 are repeated for each vector assignment statement in the where-block.

See figure 9-34 for the format of a block WHERE structure that contains an OTHERWISE statement. When the block WHERE statement is executed, the 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 9-34. 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. 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. If a value is not assigned to a vector element, data flag branches are disabled for the operations that compute that value. See section 11 for a description of data flag branches.
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. 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. If a value is not assigned to a vector element, data flag branches are disabled for the operations that compute that value. See section 11 for a description of data flag branches.
3. Steps 1 and 2 are repeated for each vector assignment statement in the otherwise-block.

See figure 9-35 for examples of block WHERE structures. The statements in the where-block of the first block WHERE structure 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.

The statements in the where-block of the second block WHERE structure 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 of the third block WHERE structure 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, elseif-block, or else-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.

```
PROGRAM BWHEREX
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'/
.
.
.
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
.
.
.
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
.
.
.
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
.
.
.
```

Figure 9-35. Block WHERE Structure Examples

VECTOR FUNCTION SUBPROGRAMS

Vector function subprograms are similar to scalar function subprograms; however, vector function subprograms return a vector result rather than a scalar result. Vector function subprograms are described in the following paragraphs.

DEFINING VECTOR FUNCTIONS

Vector functions, like scalar functions, begin with a FUNCTION statement and end with an END statement. See figure 9-36 for the format of the FUNCTION statement for vector functions.

The name of a vector function must be declared in a DESCRIPTOR statement in the vector function body.

Except for these differences, vector functions are defined in the same way as scalar functions. See section 7 for a description of scalar functions.

REFERENCING VECTOR FUNCTIONS

Vector functions, like scalar functions, are referenced by placing a function reference in a statement. See figure 9-37 for the format of a vector function reference.

```
typ FUNCTION fname (darg1, ... ,dargn;*)

typ      A type specification for fname;
         optional; type can be any of the
         following:

             INTEGER
             HALF PRECISION
             REAL
             COMPLEX
             BIT

fname    A symbolic name that is used as the
         name of the function; fname must
         appear in a DESCRIPTOR statement.

dargi    A dummy argument, which can be a
         variable, array, descriptor, descrip-
         tor array, dummy function name, or
         dummy subroutine name. It cannot be a
         vector reference. No two dummy argu-
         ments can have the same name.
```

Figure 9-36. FUNCTION Statement Format for Vector Functions

```
fname(iarg1, ... ,iargn;oarg)

fname    The name of an entry point of a vec-
         tor function subprogram.

iargi    An input argument, which can be a
         constant, symbolic constant, scalar
         expression (except concatenation of
         an operand whose length is specified
         as (*)), substring, variable, array,
         array element, vector reference, de-
         scriptor, descriptor array, descrip-
         tor array element, actual function
         name, actual subroutine name, dummy
         function name, or dummy subroutine
         name.

oarg     Output argument specifying the vector
         in which the function result is re-
         turned or the vector length.

         A vector is specified as a vector
         reference, descriptor, or descriptor
         array element.

         A vector length is specified as an
         integer expression. The compiler
         allocates a temporary vector of the
         specified length in which the function
         result is returned. The data type of
         the temporary vector is the same as
         the data type of the function.
```

Figure 9-37. Vector Function Reference Format

Input arguments are values that are passed to the vector function; the output argument is the value that is returned from the vector function.

Except for these differences, vector functions are referenced in the same way as scalar functions. See section 7 for a description of scalar functions.

VECTOR FUNCTION EXAMPLE

The two programs shown in figure 9-38 both produce the same result. Each defines and references a vector function named VPYTHAG. The difference between the two programs is in how each passes arguments.

```
1.  Array Arguments

    PROGRAM VECFUNC
    REAL A(100), B(100), C(100)
    .
    .
    .
    C(1;100) = VPYTHAG(A,B;100)
    STOP
    END

    FUNCTION VPYTHAG(VA,VB;*)
    REAL VA(100), VB(100), VC(100), VD(100)
    DESCRIPTOR VPYTHAG
    VC(1;100) = VA(1;100) + VB(1;100)
    VPYTHAG = VSQRT(VC(1;100);VD(1;100))
    RETURN
    END

2.  Descriptor Arguments

    PROGRAM VECFUNC
    REAL A(100, B(100), C(100)
    .
    .
    .
    C(1;100) = VPYTHAG(A(1;100),B(1;100);C(1;100))
    STOP
    END

    FUNCTION VPYTHAG(VA,VB;*)
    REAL VG(100), VD(100)
    DESCRIPTOR VPYTHAG, VA, VB
    VC(1;100) = VA + VB
    VPYTHAG = VSQRT(VC(1;100);VD(1;100))
    RETURN
    END
```

Figure 9-38. Vector Function Examples

The function definition in example 1 declares the dummy arguments VA and VB as arrays. The function reference, therefore, specifies array names as the actual arguments.

The function definition in example 2 declares the dummy arguments VA and VB as vector descriptors. The function reference, therefore, specifies vector references as the actual arguments.

The examples differ, also, in how each specifies the output argument. Example 1 specifies the vector length, 100; example 2 specifies the result vector reference.

SECONDARY ENTRY POINTS

Vector functions, like scalar functions, can have more than one entry point. Normally, a vector function has only one entry point, which is established by the FUNCTION statement; however, the ENTRY statement can appear in the body of the vector function in order to define secondary entry points.

See figure 9-39 for the format of the ENTRY statement for vector functions.

```
ENTRY sename (darg1, ... ,dargn;*)

sename    A symbolic name that is used as the
          name of the secondary entry point.

dargi     A dummy argument, which can be a
          variable, array, descriptor, descrip-
          tor array, dummy function name, or
          dummy subroutine name. It cannot be a
          vector reference. No two dummy argu-
          ments can have the same name.
```

Figure 9-39. ENTRY Statement for Vector Functions Format

The ENTRY point name must appear in a DESCRIPTOR statement in the body of the vector function body.

Except for these differences, secondary entry points of vector functions are defined and referenced in the same way as secondary entry points of scalar functions. See section 7 for a description of secondary entry points in scalar functions.

See figure 9-40 for an example of a secondary entry point in a vector function. The main entry point of the vector function in the example is VECISOS. A secondary entry point is VECTRI.

```
    PROGRAM VECENT
    REAL VECISOS,VECTRI
    REAL AR(100),BS(100),HT(100)
    .
    .
    .
    AR(1;100)=VECISOS(BS,BS;100)
    .
    .
    .
    AR(1;100)=VECTRI(BS,HT;100)
    .
    .
    .
    END

    FUNCTION VECISOS(B,H;*)
    REAL B(100),H(100),R(100)
    DESCRIPTOR VECISOS,VECTRI
     R(1;100)=(B(1;100)**2)-((B(1;100)/2)**2)
    H(1;100)=VSQRT(R(1;100);H(1;100))
    ENTRY VECTRI(B,H;*)
    VECTRI=(B(1;100)*H(1;100))/2
    RETURN
    END
```

Figure 9-40. Example of Secondary Entry Points in Vector Functions

LOOP VECTORIZATION

Vector machine instructions can be generated for certain types of DO loops in a FORTRAN 200 program without using any of the other vector programming features. This reduces the execution time of the statements in the DO loop. The generation of vector machine instructions for FORTRAN DO loops is called loop vectorization.

In order to use the loop vectorization feature, specify the OPTIMIZE=V compilation option on the FTN200 control statement. This causes the compiler to analyze each DO loop in the program. If the loop can be vectorized, the compiler generates vector machine instructions for the loop.

If the loop cannot be vectorized, the compiler attempts to transform the loop into a call to a STACKLIB routine. A STACKLIB routine is a predefined subroutine. STACKLIB routines are described in section 11.

If the loop cannot be vectorized or transformed into a call to a STACKLIB routine, the compiler generates the usual scalar machine instructions for the loop.

The vectorizer generates a listing that indicates how many loops were vectorized and how many loops were transformed into STACKLIB calls. Diagnostics are issued that indicate the reasons particular loops could not be vectorized. The vectorizer diagnostics are listed in appendix B. Because vectorized expressions are reordered and sometimes evaluated with different algorithms, the vectorized and original scalar code may produce different results, especially in the low order bits. This difference is primarily attributed to precision differences between the vector and scalar hardware.

The following paragraphs describe the characteristics of vectorizable loops, the generation of STACKLIB calls, and the vectorizer messages.

CHARACTERISTICS OF VECTORIZABLE DO LOOPS

A vectorizable DO loop is a loop with certain characteristics. These characteristics are summarized in tables 9-2 through 9-5. Many of these conditions are explained in detail below.

A vectorizable DO loop body contains statements with the conditions shown in table 9-2.

A vectorizable assignment is a scalar assignment with the following characteristics:

Left hand side: A simple variable,
An array element with subscripts not dependent on a control variable,
A vectorizable array element;

Right hand side: A vectorizable expression.

The assignment is subject to the additional conditions shown in table 9-3.

A vectorizable expression is a scalar expression which can contain the constructs shown in table 9-4.

A vectorizable array element is an array element whose subscripts are dependent on the control variables of vectorizable loops. The subscripted array and the subscripts are subject to the additional conditions shown in table 9-5.

TABLE 9-2. CRITERIA FOR VECTORIZABLE LOOPS

Permissible Constructs	Permissible Constructs If Unsafe Is Specified	Constructs Which Inhibit Vectorization
Control variable is type integer.	Same as permissible constructs.	Control variable is not type integer.
Control variable appears in an array subscript.	Same as permissible constructs.	Control variable is unused.
The initial or terminal parameter is not a constant, and the control variable subscripts arrays dimensions which have been declared as constant (rather than assumed or adjustable). If this is the innermost loop of a loop nest, the constant dimension restriction does not apply.	The initial or terminal parameter is not a constant and the control variable only subscripts array dimensions which are assumed or adjustable.	An inner loop has an initial or terminal parameter which is not a constant.
Incrementation parameter is an expression. (If not explicitly specified, it is a constant 1.)	Same as permissible constructs.	
Contains an inner vectorizable DO loop.	Same as permissible constructs.	Contains an inner nonvectorizable DO loop.

TABLE 9-2. CRITERIA FOR VECTORIZABLE LOOPS (Contd)

Permissible Constructs	Permissible Constructs If Unsafe Is Specified	Constructs Which Inhibit Vectorization
Contains an inner loop which has constant initial and terminal parameters.	Same as permissible constructs.	Contains an inner loop and its initial or terminal value is not a constant.
Contains an inner loop and the incrementation parameter (explicit or defaulted) of the outer loop is a constant 1.	Same as permissible constructs.	Contains an inner loop and the increment of the outer loop is not a constant 1.
Contains an inner loop and the total iteration count of all loops is less than 65536 (2**16).	Same as permissible constructs.	Contains an inner loop and the total iteration count is at least 65536.
An innermost loop with any iteration count.	Same as permissible constructs.	
The loop body contains DO and CONTINUE statements.	Same as permissible constructs.	Contains any flow control statement other than DO or CONTINUE.
The loop body contains a vectorizable scalar assignment statement.	Same as permissible constructs.	Contains any other assignment.
No other statement.		Contains any input, output, or memory transfer statement.
No label in the DO statement is referenced by a GOTO.	Same as permissible constructs.	The loop has an extended range.
Contains an inner loop with an increment of 1.	Same as permissible constructs.	Contains an inner loop with a nonunit increment.

TABLE 9-3. CRITERIA FOR VECTORIZABLE SCALAR ASSIGNMENTS

Permissible Constructs	Permissible Constructs If Unsafe Is Specified	Constructs Which Inhibit Vectorization
The assignment is feedback free.	Same as permissible constructs.	The assignment might cause feedback.
The assignment is recursion free, or the assignment is a recursive reduction assignment (sum, product, or dot product), or the assignment is a recursive interval assignment.	Same as permissible constructs.	The assignment might cause recursion.

TABLE 9-4. CRITERIA FOR VECTORIZABLE EXPRESSIONS

Permissible Constructs	Permissible Constructs If Unsafe Is Specified	Constructs Which Inhibit Vectorization
Arithmetic operators (+, -, *, /, **) and logical operators.	Same as permissible constructs.	Relational operators.
Integer, real, half-precision, and logical data elements.	Same as permissible constructs.	Any data element with type other than integer, real, half-precision, or logical.
References to the intrinsic functions ABS, ACOS, ALOG, ALOG10, ASIN, ATAN, COS, EXP, FLOAT, IABS, IFIX, SIN, SQRT, and TAN.	Same as permissible constructs. References to AINT, AIMAG, and AMOD.	References to external subroutines or functions, or references to intrinsics other than ABS, ACOS, ALOG, ALOG10, ASIN, ATAN, COS, EXP, FLOAT, IABS, IFIX, SIN, SQRT, or TAN.
Any variable which has been equivalenced.	Any variable which has been equivalenced.	
A simple variable or an array element whose subscripts are independent of the control variable of any vectorizable loop.	Same as permissible constructs.	
A vectorizable array element.	Same as permissible constructs.	An array element with loop dependent subscripts which cannot be vectorized.
The control variable of the immediately enclosing DO loop.	Same as permissible constructs.	A control variable of some outer DO loop.

TABLE 9-5. CRITERIA FOR VECTORIZABLE ARRAY ELEMENTS

Permissible Constructs	Permissible Constructs If Unsafe Is Specified	Constructs Which Inhibit Vectorization
The array has type integer, real, half-precision, or logical.	Same as permissible constructs.	The array is any type other than integer, real, half-precision, or logical.
The array has been equivalenced, and equivalenced arrays do not have feedback.	The array has been equivalenced.	
Subscript is of the form c+n, where c is a control variable and n is a loop invariant expression.	Same as permissible constructs.	
Subscript of the form n1*c+n2, where c is the control variable of the immediately enclosing DO loop, and n1 and n2 are loop invariant expressions.	Same as permissible constructs.	Subscript of the form n1*c+n2, where c is the control variable of some outer loop and n1 is not constant 1.
Subscript is a vectorizable expression.	Same as permissible constructs.	Subscript is not a vectorizable expression.
Loop dependent subscripts which increase by an invariant amount.	Same as permissible constructs.	Subscripts which do not increase by an invariant amount.
Loop dependent subscripts which are contiguous.	Same as permissible constructs.	Subscripts which are not contiguous.

The range of a vectorizable loop can contain assignment statements, CONTINUE statements, and DO statements. If other statements, such as input/output statements and IF statements, appear in the range of a DO loop, the DO loop cannot be vectorized. See loop 3 in figure 9-41 for an example of a simple vectorizable DO loop.

```
          .
          .
          .
          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
          .
          .
          .
          DO 300 MM=1,11 <--------(4)
           DO 200 JJ=1,30 <-------(5)
            DO 400 IN=1,200 <------(6)
                  .
                  .
                  .
400          CONTINUE
200         CONTINUE
300        CONTINUE
          DO 500 IN=1,70000 <------(7)
          B(IN)=C(IN)
          .
          .
          .
500        CONTINUE
```

Figure 9-41. DO Loops

The initial and terminal parameters of the DO statement can be any integer expression for the DO loop to vectorize. However, both must be constant expressions for an outer loop to vectorize. See figure 9-41 for an example. Loop 3 in the example has a variable terminal parameter. Loop 2 contains loop 3; therefore, loop 2 (the outer loop) cannot be vectorized.

The incrementation parameter of the DO statement of the innermost loop can be any expression for that loop to vectorize. However, it must be a constant one for an outer loop to vectorize. See figure 9-41.1 for an example.

The incrementation parameter of the DO statement of an outer loop must be a constant one for that loop to vectorize. See figure 9-41.2 for an example.

```
          .
          .
          .
          DO 1 I=1,100,1          -- Might vectorize
             DO A1 J=1,100,1      -- Might vectorize
          .
          .
          .
1         CONTINUE

          DO 2 I=1,100,1          -- Cannot vectorize
             DO 2 J=1,100,2       -- Might vectorize
          .
          .
          .
2         CONTINUE
```

Figure 9-41.1. DO Loops With the Incrementation Parameter #1

```
          DO 3 I=1,100,2          -- Cannot vectorize
             DO 3 J=1,100,1       -- Might vectorize
          .
          .
          .
3         CONTINUE
```

Figure 9-41.2. DO Loops With the Incrementation Parameter #2

The innermost loop can be vectorized regardless of its iterative count. However, the entire nest of loops must be less than or equal to 65535. See figure 9-41 for an example. Loop 7 in figure 9-41 will vectorize if it is the innermost loop. Loops 6 and 5 in the example can be vectorized, depending on the range of the innermost loop; however, loop 4 cannot be vectorized because 30*200*11 = 66000.

When the initial or terminal parameter of a loop is a variable, the dimensions of loop-dependent array references in 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 UNSAFE compilation option can be specified for unsafe vectorization. When UNSAFE is specified, the compiler vectorizes loops that contain variably-dimensioned arrays, even if the terminal value of the loop is variable. The optimization is considered unsafe because the presence of only variable dimensions might cause the iterative loop count to exceed 65535.

If the UNSAFE compilation option is used with variably-dimensioned arrays and variable initial or terminal values, the iterative count is assumed to be less than 65536.

The UNSAFE compilation 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, then a loop that contains the nonvectorizable loop cannot be vectorized. See figure 9-41 for an example. Loop 2 in the example cannot be vectorized; therefore, loop 1 cannot be vectorized.

If the UNSAFE compilation option is specified, every array is considered independent of any other. Otherwise, possible overlaps of equivalenced arrays are considered.

Arithmetic 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 appearing in the range of a vectorizable loop can be of type integer, half-precision, real, or logical. See figure 9-42 for an example of a vectorizable loop that contains a logical assignment statement.

References to variably-dimensioned arrays cause a loop with a variable initial or terminal value to be nonvectorizable, unless the loop is the innermost loop, or the UNSAFE compilation option is selected. In figure 9-43, loop 1 is vectorizable but loop 2 is only vectorized if the UNSAFE compilation option is selected.

Function and array references can appear in the range of a vectorizable loop. However, function references are restricted to references to the intrinsic functions ABS, IABS, FLOAT, IFIX, SQRT, EXP, ALOG, ALOG10, SIN, COS, TAN, ACOS, ASIN, and ATAN. References in a loop to other intrinsic functions, or to any functions that you provide, cause the loop to be nonvectorizable.

Loop-dependent array references are subject to several restrictions. Loop-independent array references are considered to be scalars in the context of loop vectorization.

```
      .
      .
      .
      FUNCTION F(OFFS,F1,F2,N)
      DIMENSION OFFS(10,N), F1(10,N),F2(N)
      DO 1 I=1,N <-------------------------- (1)
1       OFFS (1,I) = F1(2,I) / 4.0
      DO 2 J=1,N <-------------------------- (2)
          DO 3 I=1,10
              OFFS(I,J) = F1(I,J) + 5.0
3         CONTINUE
      F2(J)=2*J
2     CONTINUE
      F=OFFS(2,2)
      RETURN
      END
      .
      .
      .
```

Figure 9-43. Vectorizable Loop #2

The left side of an assignment statement appearing in the range of a vectorizable DO loop must be a loop-dependent array reference or a scalar reference. A vector reference or descriptor on the left side makes the loop nonvectorizable. A loop-dependent array reference is an array reference with at least one loop-dependent subscript expression. See figures 9-41, 9-42, 9-43, and 9-44 for examples. The left sides of the assignment statements in the examples are all loop-dependent array references.

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

Figure 9-44. Vectorizable Loop #3

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

Figure 9-42. Vectorizable Loop #1

Scalar Assignments in Vectorizable Loops

A scalar reference is a simple variable or a loop independent array reference. Scalars appearing as the left hand side of an assignment are subject to certain restrictions in order that the containing loop vectorizes. The restrictions are:

If a scalar is defined by a recursive reduction assignment, the scalar cannot be assigned or referenced in any other vectorizable statement. If a scalar is defined by a recursive interval assignment, the scalar cannot be referenced in any statement preceding the recursive definition. It cannot be defined in any other statement, before or after. See figure 9-45.

If a scalar is not defined by a recursive assignment, the scalar cannot be referenced before the first definition of that scalar in the loop. The right hand side of an assignment is referenced before the left hand side.

If the scalar is defined in an outer loop and also appears in a loop contained within the outer loop, the outer loop cannot be vectorized.

If the scalar is an array element, every reference to the array must have the same subscript.

```
Reduction Assignments:

    s = s + u(i)                (SUM)
    s = s * u(i)                (PRODUCT)
    s = s + u(i) * v(i)         (DOT PRODUCT)

Interval Assignments:

    s = s + r


    s       Scalar

    r       Loop invariant scalar expression

    u       Vector expression

    v       Vector expression

All variables and expressions must be real,
integer, or half-precision.
```

Figure 9-45. General Form of Recursive Assignments

See figure 9-46 for examples. Loop 1 has nonvectorizable recursive statements, while loop 2 is vectorizable. Loop 3 is not vectorizable because T is referenced before its definition. Loop 4 is not vectorizable because T is defined in the outer loop and referenced in an inner loop. Loop 5 is vectorizable. Loop 6 is not vectorizable due to different subscripts on array A.

```
      DIMENSION A(10,10), B(10,10), C(10,10)
      DO 1 I=1,10 <-------------------------(1)
      R = R + A(I,1)
      R = S * B(1,I)
      S = S + I * C(I,I)
      P = P + Q
1     Q = Q + 2
      DO 2 I=1,10 <-------------------------(2)
      R = R + A(I,1)
      S = S * B(1,I)
      T = T + I * C(I,I)
      Q = Q + 2
2     P = P + Q
      DO 3 I=1,10 <-------------------------(3)
      A(I,1) = T
3     T = B(I,2)
      DO 4 J=1,10 <-------------------------(4)
      T = A(I,1) + B(I,1)
      C(I,1) = C(I,1) * T + A(I,1)/T
      DO 5 J=1,10 <-------------------------(5)
      T = A(J,I) + B(J,I)
      C(J,I) = C(J,I) * T + A(J,I)/T
5     CONTINUE
4     CONTINUE
      DO 6 I=1,10 <-------------------------(6)
      A(1,K) = 2
6     A(1,K+1) = 3
```

Figure 9-46. Vectorizable and Nonvectorizable Loops with Scalars

Loop-Dependent Array References in Vectorizable Loops

The subscript expression of a loop-dependent array reference must be of the proper form in order for the loop to be vectorized. See figure 9-47 for the subscript expression forms that can be used for loop-dependent array references without causing the loop to be nonvectorizable.

```
c
c+n
c-n
c*n


Any expression that can appear on the right
hand side of an assignment.

c    A control variable.

n    Loop invariant expression.
```

Figure 9-47. Subscript Expression Forms

See figures 9-41, 9-42, and 9-48 for examples of loops that contain subscript expressions that can appear in a vectorizable loop.

As the control variable passes through the range of values, the loop-dependent subscripts of array references must increase by an invariant amount.

```
      .
      .
      .
      DIMENSION A(-2:2), X(10), B(100), C(100000)
      READ (62) X
      DO 1 I=1,10
1       A(SIN(I)) = B(I**2) + C(X(I))
      .
      .
      .
```

Figure 9-48. Vectorizable Loop #4

See figure 9-44 for an example. In the example, the subscripts of array A are increasing in increments of 2, and the subscripts of array B are increasing in increments of 22. Therefore, loop 2 can be vectorized, but loop 1 cannot be vectorized.

Although loop-dependent array references to a particular array can appear on both sides of assignment statements in the range of a DO loop, in certain cases this could inhibit vectorization of the loop. When an array reference appears on the right side of an assignment statement, elements of the array are being accessed. When an array reference appears on the left side, elements of the array are being defined. If, for any particular array, an array element is defined and then redefined or accessed in any subsequent iteration, that loop has feedback. Unless the compiler can be certain feedback does not occur, the containing loop is not vectorized. It is possible to redefine an array element in the same iteration or to overlap accesses and definitions so that an array element is accessed before or in the same iteration as the definition. Because of the parallel nature of vector operations, vectors are not suitable for use in describing any iterative procedure containing feedback.

See figure 9-49 for an example that illustrates feedback.

The program segment in the example consists of a DO loop whose terminal statement is a CONTINUE statement. The loop contains an assignment statement, which contains two loop-dependent array references. The array elements referenced and defined by successive iterations of the loop are shown in the example.

Elements A(2), A(3), and A(4) constitute the overlap. On the first iteration of the loop, A(2) is 1. On the second iteration, A(2) is accessed and is used to define A(3). After execution of the loop, the five elements of A have the values 1, 2, 4, 8, and 16 respectively.

A vectorizer interpretation of the same loop would be to assign the $(i-1)^{th}$ element multiplied by 2 to the i^{th} element of A, where i ranges from 2 to 5. In this case the five elements of A would have the values 1, 2, 4, 6, and 8 respectively after execution of the loop.

The loop is not vectorizable.

```
      .
      .
      .
      DIMENSION A(5)
      DATA A/1,2,3,4,5/
      .
      .
      .
      DO 1 I=1,4
      A(I+1)=A(I)*2
1     CONTINUE
      .
      .
      .
```

The loop references and defines the elements of A as follows:

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

Figure 9-49. Feedback Example

See figure 9-50 for an example of a loop that contains overlap but not feedback. The array elements referenced and defined by successive iterations of this loop are shown in the example.

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

```
      .
      .
      .
      DIMENSION A(5)
      DATA A/1,2,3,4,5/
      .
      .
      .
      DO 2 I=2,5
      A(I-1)=A(I)*2
2     CONTINUE
      .
      .
      .
```

The loop references and defines the elements of A as follows:

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

Figure 9-50. Overlap Example

Unless all subscripts are of the form n1*c+n2, it is very difficult to determine if feedback exists. In figure 9-51, loop 1 is not vectorizable. If any two elements of X are identical, this loop indeed has feedback. If the same loop is split into two parts, loops 2 and 3, the loops are vectorizable.

```
      .
      .
      .
      REAL A(100000), TEMP(100)
      INTEGER X(100)
      READ(62)X
      DO 1 I=1,100 <-------------------- (1)
1       A(X(I)) = A(X(I)) + 1
      DO 2 I=1,100 <-------------------- (2)
2       TEMP(I) = A(X(I)) + 1
      DO 3 I=1,100 <-------------------- (3)
3       A(X(I)) = TEMP(I)
      .
      .
      .
```

Figure 9-51. Possible Feedback With Generalized Subscripts

GENERATION OF CALLS TO STACKLIB ROUTINES

If a loop cannot be vectorized, the compiler attempts to transform the loop into a call to a STACKLIB routine or replace the loop with inline vector macro code. A STACKLIB routine is a predefined subroutine; STACKLIB routines are described in section 11. Inline vector macro code is composed of machine instructions that are placed in the object code produced by the compiler.

In order for a loop to be transformed into a call to a STACKLIB routine or into inline vector macro code, the loop must contain no nested loops and the loop must be of the proper type. See figure 9-52 for the types of loops that can be transformed.

In all of the loops in figure 9-52, X and Y represent distinct one-dimensional arrays of type real which do not appear in an EQUIVALENCE statement. S represents a simple real variable. Variables L and M represent any DO loop initial and terminal value parameters. The variable I represents any DO loop control variable.

A transformable loop must contain only one assignment statement of one of the forms indicated in figure 9-52. CONTINUE statements can appear in the loop. The loop incrementation parameter must be 1, which is the default.

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

Figure 9-52. Transformable Loops

Loops 1 and 2 in figure 9-52 are transformed into calls to a STACKLIB routine that performs addition recursively. Loops 3 through 10 are transformed into inline vector macro code.

LOOP VECTORIZATION MESSAGES

Messages are printed on the source listing that indicate how many loops exist in the program, how many loops are vectorized, and how many loops are transformed into calls to STACKLIB routines.

For loops that cannot be vectorized, a message is issued that indicates the first impediment to vectorization encountered by the compiler. The compiler analyzes a loop for vectorization from the bottom to the top; therefore, the diagnostic might not reflect the impediment to vectorization with the lowest source line number. See appendix B for a complete list of the vectorizer messages.

Messages are also printed on the source listing that indicate which loops are transformed into calls to STACKLIB routines.

See figure 9-53 for an example of the source listing of a program compiled with the OPTIMIZE=V compilation option.

```
00001           PROGRAM VECTRISE
00002           DIMENSION A(100),B(100)
00003           EXTERNAL F
00004           DO 1 I=1,100
00005    1        A(I)=A(I)+B(I)
00006           DO 2 I=1,100
00007    2        A(I)=A(I-1)+B(I)
00008           DO 3 I=1,100
00009    3        B(F(I))=A(I)
00010           END

 3 LOOPS WERE EXAMINED FOR POSSIBLE VECTORIZATION.

 1 LOOP WAS VECTORIZED.

     [FIRST LINE OF VECTORIZED LOOPS     (NUMBER OF LOOPS VECTORIZED)]
     LINE  (COUNT)
     -------------
     00004 (     1)

 1 LOOP WAS STACKLIBBED.

     [FIRST LINE OF STACKLIBBED LOOPS    (NUMBER OF LOOPS STACKLIBBED)]
     LINE  (COUNT)
     -------------
     00006 (     1)

 1 LOOP WAS LEFT SCALAR.

     [FIRST LINE OF THE LOOP / LINE THAT PREVENTS VECTORIZATION / THE REASON THAT PREVENTS VECTORIZATION]
     FIRST  AT LINE  REASON THAT THE LOOP WAS NOT VECTORIZED.
     -----  -------  ----------------------------------------
     00008  00009    A DESTINATION VARIABLE CONTAINS AN EXTERNAL SUBPROGRAM REFERENCE (F).

VECTRISE - NO ERRORS
```

Figure 9-53. Vectorizer Output

The FORTRAN 200 language includes a number of functions that are predefined and can be referenced from a program. These functions are called intrinsic functions.

Intrinsic functions perform common operations, such as converting values from one data type to another, truncating values, and finding the largest or smallest values in a list of values. Some of the intrinsic functions perform common mathematical computations, such as computing the square root of a number, changing the sign of a number, and performing trigonometric operations.

An intrinsic function is referenced by placing a function reference in a FORTRAN statement. Scalar function references are described in section 7; vector function references are described in section 9.

You can reference a function that you have supplied that has an entry point of the same name as an intrinsic function. To do this, you must declare the entry point name in an EXTERNAL statement and supply the function that has that entry point. If you declare the entry point name in an EXTERNAL statement and provide a function that has an entry point of that name, the intrinsic function cannot be referenced in that program unit.

If an intrinsic function name appears in the actual argument list of a function reference or subroutine call, the intrinsic function name must be declared in an INTRINSIC statement in the same program unit.

An intrinsic function can be a specific function or a generic function. A specific function accepts arguments of a specific data type and returns a result of a specific data type. A generic function accepts arguments of more than one data type. Except for type conversion generic functions, the data type of the arguments determines the data type of the result.

A generic function name must not appear in the actual argument list of a function reference or subroutine call unless the name is used as a specific function name. For example, ABS is both a generic function name and a specific function name. If it is used as a generic function name, it must not be passed as an argument; if it is used as a specific function name, it can be passed as an argument.

An intrinsic function can be a scalar intrinsic function or a vector intrinsic function. A scalar intrinsic function produces a scalar result; a vector intrinsic function produces a vector result.

SCALAR INTRINSIC FUNCTIONS

A scalar intrinsic function produces a scalar result. The arguments that are passed to a scalar intrinsic function can be scalar arguments, vector arguments, or both, depending on the function. A scalar argument is a constant, symbolic constant, expression (except concatenation of an operand whose length is specified as (*)), substring, variable, or array element. A vector argument is a vector, descriptor, or descriptor array element.

At execution time, the result of a reference to a scalar intrinsic function is returned through the function name and replaces the function reference in the statement. Scalar intrinsic functions do not alter the values of the arguments that are passed to them.

Some of the scalar intrinsic function names begin with the prefix Q8S. These functions perform more complicated manipulations and usually use a CYBER 200 hardware feature. Functions that begin with the prefix Q8S must not appear in the argument list of a function reference or subroutine call.

See table 10-1 for a list of the scalar intrinsic functions. For each function, the table shows the purpose of the function, the arguments accepted by the function, the generic name of the function, the specific names of the function, the data type of the arguments accepted by the function, and the data type of the result returned by the function. The arguments are represented as a for a scalar, v for a vector, cv for a control vector, and i for integer scalar.

TABLE 10-1. SCALAR INTRINSIC FUNCTIONS

Purpose	Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Absolute value	(a)	ABS	IABS HABS ABS DABS CABS	INTEGER HALF REAL DOUBLE COMPLEX	INTEGER HALF REAL DOUBLE REAL
Arccosine	(a)	ACOS	HACOS ACOS DACOS	HALF REAL DOUBLE	HALF REAL DOUBLE

TABLE 10-1. SCALAR INTRINSIC FUNCTIONS (Contd)

Purpose	Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Imaginary part of a complex number	(a)	-	AIMAG	COMPLEX	REAL
Truncate	(a)	AINT	HINT AINT DINT	HALF REAL DOUBLE	HALF REAL DOUBLE
Nearest whole number	(a)	ANINT	HNINT ANINT DNINT	HALF REAL DOUBLE	HALF REAL DOUBLE
Arcsine	(a)	ASIN	HASIN ASIN DASIN	HALF REAL DOUBLE	HALF REAL DOUBLE
Arctangent	(a)	ATAN	HATAN ATAN DATAN	HALF REAL DOUBLE	HALF REAL DOUBLE
	(a_1,a_2)	ATAN2	HATAN2 ATAN2 DATAN2	HALF REAL DOUBLE	HALF REAL DOUBLE
Convert to logical	(a)	-	BTOL	BIT	LOGICAL
Return character whose internal hexadecimal representation is equivalent to the integer argument	(a)	-	CHAR	INTEGER	CHARACTER
Convert to complex	(a) or (a_1,a_2)	CMPLX	- - CMPLX - -	INTEGER HALF REAL DOUBLE COMPLEX	COMPLEX COMPLEX COMPLEX COMPLEX COMPLEX
Conjugate of a complex number	(a)	-	CONJG	COMPLEX	COMPLEX
Cosine	(a)	COS	HCOS COS DCOS CCOS	HALF REAL DOUBLE COMPLEX	HALF REAL DOUBLE COMPLEX
Hyperbolic cosine	(a)	COSH	HCOSH COSH DCOSH	HALF REAL DOUBLE	HALF REAL DOUBLE
Cotangent	(a)	COTAN	HCOTAN COTAN	HALF REAL	HALF REAL
Date	( ) or (a)	-	DATE	any	CHARACTER*8
Convert to double-precision	(a)	DBLE	DFLOAT - - - -	INTEGER HALF REAL DOUBLE COMPLEX	DOUBLE DOUBLE DOUBLE DOUBLE DOUBLE
Positive difference	(a_1,a_2)	DIM	IDIM HDIM DIM DDIM	INTEGER HALF REAL DOUBLE	INTEGER HALF REAL DOUBLE
Extended precision product	(a_1,a_2)	-	DPROD RPROD	REAL HALF	DOUBLE REAL

TABLE 10-1. SCALAR INTRINSIC FUNCTIONS (Contd)

Purpose	Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Exponential	(a)	EXP	HEXP EXP DEXP CEXP	HALF REAL DOUBLE COMPLEX	HALF REAL DOUBLE COMPLEX
Convert to half-precision	(a)	HALF	- - - - -	INTEGER HALF REAL DOUBLE COMPLEX	HALF HALF HALF HALF HALF
Return integer equivalent of the internal hexadecimal representation of the character argument	(a)	-	ICHAR	CHARACTER	INTEGER
Index of a substring	(a_1,a_2)	-	INDEX	CHARACTER	INTEGER
Convert to integer	(a)	INT	- IHINT INT IFIX IDINT -	INTEGER HALF REAL REAL DOUBLE COMPLEX	INTEGER INTEGER INTEGER INTEGER INTEGER INTEGER
Length	(a)	-	LEN	CHARACTER	INTEGER
Lexically greater than or equal to	(a_1,a_2)	-	LGE	CHARACTER	LOGICAL
Lexically greater than	(a_1,a_2)	-	LGT	CHARACTER	LOGICAL
Lexically less than or equal to	(a_1,a_2)	-	LLE	CHARACTER	LOGICAL
Lexically less than	(a_1,a_2)	-	LLT	CHARACTER	LOGICAL
Natural logarithm	(a)	LOG	HLOG ALOG DLOG CLOG	HALF REAL DOUBLE COMPLEX	HALF REAL DOUBLE COMPLEX
Common logarithm	(a)	LOG10	HLOG10 ALOG10 DLOG10	HALF REAL DOUBLE	HALF REAL DOUBLE
Convert to bit	(a)	-	LTOB	LOGICAL	BIT
Largest value	(a_1,a_2...)	MAX	MAXO HMAX1 AMAX1 DMAX1	INTEGER HALF REAL DOUBLE	INTEGER HALF REAL DOUBLE
		-	AMAXO MAX1	INTEGER REAL	REAL INTEGER
Smallest value	(a_1,a_2,...)	MIN	MINO HMIN1 AMIN1 DMIN1	INTEGER HALF REAL DOUBLE	INTEGER HALF REAL DOUBLE
		-	AMINO MIN1	INTEGER REAL	REAL INTEGER

TABLE 10-1. SCALAR INTRINSIC FUNCTIONS (Contd)

Purpose	Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Remainder	(a_1,a_2)	MOD	MOD HMOD AMOD DMOD	INTEGER HALF REAL DOUBLE	INTEGER HALF REAL DOUBLE
Nearest integer	(a)	NINT	IHNINT NINT IDNINT	HALF REAL DOUBLE	INTEGER INTEGER INTEGER
Count number of 1 bits in bit vector	(v)	-	Q8SCNT	BIT	INTEGER
Test a bit in the data flag branch register	(a_1,a_2)	-	Q8SDFB	INTEGER	LOGICAL
Compute dot product of two vectors	(v_1,v_2)	Q8SDOT	- - -	INTEGER HALF REAL	INTEGER HALF REAL
Number of pairs of corresponding vector elements preceding first pair of corresponding vector elements in which the element of the first vector is equal to the element of the second vector	(v_1,v_2)	Q8SEQ	- - -	INTEGER HALF REAL	INTEGER INTEGER INTEGER
Extract bits	(a,i_1,i_2)	Q8SEXTB	- - -	REAL INTEGER LOGICAL	TYPELESS TYPELESS TYPELESS
Number of pairs of corresponding vector elements preceding first pair of corresponding vector elements in which the element of the first vector is greater than or equal to the element of the second vector	(v_1,v_2)	Q8SGE	- - -	INTEGER HALF REAL	INTEGER INTEGER INTEGER
Insert bits	(a_1,i_1,i_2,a_2)	Q8SINSB	- - -	REAL INTEGER LOGICAL	TYPELESS TYPELESS TYPELESS
Obtain length of vector	(v)	Q8SLEN	- - - -	INTEGER HALF REAL COMPLEX	INTEGER INTEGER INTEGER INTEGER
Number of pairs of corresponding vector elements preceding first pair of corresponding vector elements in which the element of the first vector is less than the element of the second vector	(v_1,v_2)	Q8SLT	- - -	INTEGER HALF REAL	INTEGER INTEGER INTEGER
Largest vector element	(v) or (v,cv)	Q8SMAX	- - -	INTEGER HALF REAL	INTEGER HALF REAL
Number of elements preceding largest vector element	(v) or (v,cv)	Q8SMAXI	- - -	INTEGER HALF REAL	INTEGER INTEGER INTEGER

TABLE 10-1. SCALAR INTRINSIC FUNCTIONS (Contd)

Purpose	Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Smallest vector element	(v) or (v,cv)	Q8SMIN	- - -	INTEGER HALF REAL	INTEGER INTEGER INTEGER
Number of elements preceding smallest vector element	(v) or (v,cv)	Q8SMINI	- - -	INTEGER HALF REAL	INTEGER HALF REAL
Number of pairs of corresponding vector elements preceding first pair of corresponding vector elements in which the element of the first vector is not equal to the element of the second vector	(v_1,v_2)	Q8SNE	- - -	INTEGER HALF REAL	INTEGER INTEGER INTEGER
Compute product of vector elements	(v) or (v,cv)	Q8SPROD	- - -	INTEGER HALF REAL	INTEGER HALF REAL
Compute sum of vector elements	(v) or (v,cv)	Q8SSUM	- - -	INTEGER HALF REAL	INTEGER HALF REAL
Random number in the range 0 to 1	( ) or (a)	-	RANF	any	REAL
Convert to real	(a)	REAL	REAL FLOAT EXTEND - SNGL -	INTEGER INTEGER HALF REAL DOUBLE COMPLEX	REAL REAL REAL REAL REAL REAL
CPU time in seconds since start of job	( ) or (a)	-	SECOND	any	REAL
Transfer of sign	(a_1,a_2)	SIGN	ISIGN HSIGN SIGN DSIGN	INTEGER HALF REAL DOUBLE	INTEGER HALF REAL DOUBLE
Sine	(a)	SIN	HSIN SIN DSIN CSIN	HALF REAL DOUBLE COMPLEX	HALF REAL DOUBLE COMPLEX
Hyperbolic sine	(a)	SINH	HSINH SINH DSINH	HALF REAL DOUBLE	HALF REAL DOUBLE
Square root	(a)	SQRT	HSQRT SQRT DSQRT CSQRT	HALF REAL DOUBLE COMPLEX	HALF REAL DOUBLE COMPLEX
Tangent	(a)	TAN	HTAN TAN DTAN	HALF REAL DOUBLE	HALF REAL DOUBLE
Hyperbolic tangent	(a)	TANH	HTANH TANH DTANH	HALF REAL DOUBLE	HALF REAL DOUBLE
Time of day	( ) or (a)	-	TIME	any	CHARACTER*8

To reference a function that you supply that has the same name as one of the intrinsic functions described in this section, you must declare the function name in an EXTERNAL statement and supply the function. However, if you do this, you cannot also reference the intrinsic function having that name within that program unit.

Note that you must both declare the function name in an EXTERNAL statement and supply the function. If you declare a function name in an EXTERNAL statement and do not supply the function, a reference to the function references the intrinsic function having that name. However, the DATE, TIME and SECOND functions are exceptions to this rule.

VECTOR INTRINSIC FUNCTIONS

A vector intrinsic function produces a vector result. The argument list of a vector intrinsic function reference consists of one or more input arguments followed by one output argument. A semicolon separates the input arguments from the output argument.

The input arguments are the arguments whose values are passed to the vector intrinsic function. The input arguments can be scalar arguments, vector arguments, or both, depending on the function. A scalar argument is a constant, symbolic constant, expression (except concatenation of an operand whose length is specified as (*)), substring, variable, or array element. A vector argument is a vector, descriptor, or descriptor array element.

The output argument is the argument whose value is changed or returned by the vector intrinsic function. The output argument can be a vector argument or an integer expression. If the output argument is a vector argument, the vector function result is returned through the output argument. If the output argument is an integer expression, the compiler automatically allocates a temporary vector through which the result is returned. The temporary vector data type will be the same as the function, and the vector length will be the integer expression result.

At execution time, the result of a reference to a vector intrinsic function is returned through the output argument. Some vector intrinsic functions alter the values that the output argument had before the function reference; other vector intrinsic functions assign new values to the output argument without regard to its previous values. A vector intrinsic function does not return values through the function name.

Many of the vector intrinsic functions are the vector equivalents of the scalar intrinsic functions. The names of these functions are the letter V followed by the scalar intrinsic function name.

Some of the vector intrinsic function names begin with the prefix Q8V. These functions perform more complicated manipulations and usually use a CYBER 200 hardware feature. Functions that begin with the prefix Q8V must not appear in the argument list of a function reference or subroutine call.

See table 10-2 for a list of the vector intrinsic functions. For each function, the table shows the purpose of the function, the arguments accepted by the function, the generic name of the function, the specific names of the function, the data type of the arguments accepted by the function, and the data type of the result returned by the function. The arguments are represented as u for the output argument, v for a vector, cv for a control vector, and a, i, or n for a scalar.

TABLE 10-2. VECTOR INTRINSIC FUNCTIONS

Purpose	Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Compute averages of adjacent elements	(v;u)	Q8VADJM	- -	HALF REAL	HALF REAL
Compute averages of corresponding elements	(v_1,v_2;u)	Q8VAVG	- -	HALF REAL	HALF REAL
Compute average differences of corresponding elements	(v_1,v_2;u)	Q8VAVGD	- -	HALF REAL	HALF REAL
Delete selected elements from vector	(v,cv;u)	Q8VCMPRS	- - -	INTEGER HALF REAL	INTEGER HALF REAL
Store selected elements into vector	(v,cv;u)	Q8VCTRL	- - -	INTEGER HALF REAL	INTEGER HALF REAL
Compute differences between adjacent elements of vector	(v;u)	Q8VDELT	- -	HALF REAL	HALF REAL
Number of elements preceding the value of each element of the first vector in the second vector	(v_1,v_2;u)	Q8VEQI	- -	HALF REAL	INTEGER INTEGER
Select elements at specified interval from one vector to create a vector	(v,i,n;u)	Q8VGATHP	- - -	INTEGER HALF REAL	INTEGER HALF REAL

TABLE 10-2. VECTOR INTRINSIC FUNCTIONS (Contd)

Purpose	Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Select elements from one vector to create a vector	(v,i;u)	Q8VGATHR	- - -	INTEGER HALF REAL	INTEGER HALF REAL
For each element in the first vector, the number of elements preceding the first element of the second vector for which the element in the first vector is greater than or equal to the element in the second vector.	(v_1,v_2;u)	Q8VGEI	- -	HALF REAL	INTEGER INTEGER
Create a vector whose elements form an arithmetic progression	(a_1,a_2;u)	Q8VINTL	- - -	INTEGER HALF REAL	INTEGER HALF REAL
For each element in the first vector, the number of elements preceding the first element of the second vector for which the element in the first vector is less than the element in the second vector.	(v_1,v_2;u)	Q8VLTI	- -	HALF REAL	INTEGER INTEGER
Mask values of two vectors into one vector	(v_1,v_2,cv;u)	Q8VMASK	- - -	INTEGER HALF REAL	INTEGER HALF REAL
Merge values of two vectors into one vector	(v_1,v_2,cv;u)	Q8VMERG	- - -	INTEGER HALF REAL	INTEGER HALF REAL
Create a bit pattern whose first group of bits are 1's	(a_1,a_2;u)	Q8VMKO	-	INTEGER	BIT
Create a bit pattern whose first group of bits are 0's	(a_1,a_2;u)	Q8VMKZ	-	INTEGER	BIT
Number of elements preceding the first element of the second vector that is not equal to each element of the first vector	(v_1,v_2;u)	Q8VNEI	- -	HALF REAL	INTEGER INTEGER
Reverse order of elements in vector	(v;u)	Q8VREV	- - -	INTEGER HALF REAL	INTEGER HALF REAL
Scatter elements taken at specified interval from one vector into another vector	(v,i,n;u)	Q8VSCATP	- - -	INTEGER HALF REAL	INTEGER HALF REAL
Scatter elements from one vector into another vector	(v,i;u)	Q8VSCATR	- - -	INTEGER HALF REAL	INTEGER HALF REAL
Insert zeros into vector	(v,cv;u)	Q8VXPND	- - -	INTEGER HALF REAL	INTEGER HALF REAL
Absolute value	(v;u)	VABS	VIABS VHABS VABS VCABS	INTEGER HALF REAL COMPLEX	INTEGER HALF REAL REAL
Arccosine	(v;u)	VACOS	VHACOS VACOS	HALF REAL	HALF * REAL

TABLE 10-2. VECTOR INTRINSIC FUNCTIONS (Contd)

Purpose	Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Imaginary part of a complex number	(v;u)	-	VAIMAG	COMPLEX	REAL
Truncate	(v;u)	VAINT	VAINT VHINT	REAL HALF	REAL HALF
Nearest whole number	(v;u)	VANINT	VANINT VHNINT	REAL HALF	REAL HALF
Arcsine	(v;u)	VASIN	VHASIN VASIN	HALF REAL	HALF REAL
Arctangent	(v;u)	VATAN	VHATAN VATAN	HALF REAL	HALF REAL
	(v_1,v_2;u)	VATAN2	VHATAN2 VATAN2	HALF REAL	HALF REAL
Convert to complex	(v_1,v_2;u) or (v;u)	VCMPLX	- - - -	COMPLEX INTEGER REAL HALF	COMPLEX COMPLEX COMPLEX COMPLEX
Conjugate of a complex number	(v;u)	-	VCONJG	COMPLEX	COMPLEX
Cosine	(v;u)	VCOS	VHCOS VCOS VCCOS	HALF REAL COMPLEX	HALF REAL COMPLEX
Convert to double-precision	(v;u)	VDBLE	-	REAL	DOUBLE
Positive difference	(v_1,v_2;u)	VDIM	VIDIM VHDIM VDIM	INTEGER HALF REAL	INTEGER HALF REAL
Exponential	(v;u)	VEXP	VHEXP VEXP VCEXP	HALF REAL COMPLEX	HALF REAL COMPLEX
Convert to half-precision	(v;u)	VHALF	- - - -	HALF INTEGER REAL COMPLEX	HALF HALF HALF HALF
Convert to integer	(v;u)	VINT	- VIHINT VINT VIFIX -	INTEGER HALF REAL REAL COMPLEX	INTEGER INTEGER INTEGER INTEGER INTEGER
Natural logarithm	(v;u)	VLOG	VHLOG VALOG VCLOG	HALF REAL COMPLEX	HALF REAL COMPLEX
Common logarithm	(v;u)	VLOG10	VHLOG10 VALOG10	HALF REAL	HALF REAL
Remainder	(v_1,v_2;u)	VMOD	VMOD VHMOD VAMOD	INTEGER HALF REAL	INTEGER HALF REAL
Nearest integer	(v;u)	VNINT	VNINT VIHNINT	REAL HALF	INTEGER INTEGER
Random Vector	(i;u)	-	VRAND	INTEGER	REAL

TABLE 10-2. VECTOR INTRINSIC FUNCTIONS (Contd)

Purpose	Arguments	Generic Name	Specific Name	Type of Argument	Type of Function
Convert to real	(v;u)	VREAL	- VFLOAT VREAL VEXTEND VSNGL -	REAL INTEGER INTEGER HALF DOUBLE COMPLEX	REAL REAL REAL REAL REAL REAL
Transfer of sign	$(v_1,v_2;u)$	VSIGN	VISIGN VHSIGN VSIGN	INTEGER HALF REAL	INTEGER HALF REAL
Sine	(v;u)	VSIN	VHSIN VSIN VCSIN	HALF REAL COMPLEX	HALF REAL COMPLEX
Square root	(v;u)	VSQRT	VHSQRT VSQRT VCSQRT	HALF REAL COMPLEX	HALF REAL COMPLEX
Tangent	(v;u)	VTAN	VHTAN VTAN	HALF REAL	HALF REAL

FUNCTION DESCRIPTIONS

The scalar and vector intrinsic functions are described in the following paragraphs. The functions are listed in alphabetical order.

See table 10-1 or table 10-2 for a summary of the intrinsic functions.

The mathematical generic intrinsic functions are also listed in table 10-3. The mathematical generic intrinsic functions are listed in alphabetical order according to their generic scalar function names. The corresponding generic vector function name and the specific scalar and vector function names are also listed. This table gives the mathematical definition, domain, and range of each mathematical intrinsic function.

The values returned by some of the mathematical intrinsic functions can be infinite.

TABLE 10-3. MATHEMATICAL FUNCTIONS

Generic Scalar Function	Generic Vector Function	Specific Scalar Function	Specific Vector Function	Mathematical Definition	Domain (Argument Range)	Range (Result Range)
ACOS(a)	VACOS(v;u)	HACOS ACOS DACOS	VHACOS VACOS	cos (a)	$\|a\| \leq 1$	$0 \leq$ result $\leq \pi$
ASIN(a)	VASIN(v;u)	HASIN ASIN DASIN	VHASIN VASIN	$\sin^{-1}(a)$	$\|a\| \leq 1$	$-\pi/2 \leq$ result $\leq \pi/2$
ATAN(a)	VATAN(v;u)	HATAN ATAN DATAN	VHATAN VATAN	$\tan^{-1}(a)$		$-\pi/2 \leq$ result $\leq \pi/2$
ATAN2(a_1,a_2)	VATAN2(v;u)	HATAN2 ATAN2 DATAN2	VHATAN2 VATAN2	$\tan^{-1}(a_1/a_2)$		$-\pi <$ result $\leq \pi$
COS(a)	VCOS(v;u)	HCOS COS DCOS	VHCOS VCOS	cos(a)		$-1 \leq$ result ≤ 1
		CCOS	VCCOS		imaginary part < 19905.80	

TABLE 10-3. MATHEMATICAL FUNCTIONS (Contd)

Generic Scalar Function	Generic Vector Function	Specific Scalar Function	Specific Vector Function	Mathematical Definition	Domain (Argument Range)	Range (Result Range)
COSH(a)	-	HCOSH	-	cosh(a)	|a| < 92.88171	1 ≤ result
		COSH DCOSH			|a| -19842.031 < a < 19905.80	
COTAN(a)	-	HCOTAN COTAN	-	cotan(a)		
-	-	DPROD	-	$a_1 * a_2$		
EXP(a)	VEXP(v;u)	HEXP	VHEXP	e^a	a < 92.88171	0 ≤ result
		EXP DEXP	VEXP		a < 19905.80	
		CEXP	VCEXP		real part < 19905.80	
LOG(a)	VLOG(v;u)	HLOG	VHLOG	$\log_e(a)$	a > 0	result < 92.88171
		ALOG DLOG	VALOG			result < 19905.80
		CLOG	VCLOG		a ≠ (0.,0.)	real part < 19905.80 - π < imaginary ≤ π
LOG10(a)	VLOG10(v;u)	HLOG10	VHLOG10	$\log_{10}(a)$	a > 0	result < 40.33801
		ALOG10 DLOG10	VALOG10			result < 8644.979
MOD(a_1,a_2)	VMOD(v_1,v_2;u)	MOD HMOD AMOD DMOD	VMOD VHMOD VAMOD	$a_1-[a_1/a_2]*a_2$	$a_2 \neq 0$	
-	-	RPROD	-	$a_1 * a_2$		
SIN(a)	VSIN(v;u)	HSIN SIN DSIN	VHSIN VSIN	sin(a)		-1 ≤ result ≤ 1
		CSIN	VCSIN		imaginary part < 19905.80	
SINH(a)	-	HSINH	-	sinh(a)	|a| < 92.88171	
		SINH DSINH			|a| -19842.031 < a < 19905.80	
SQRT(a)	VSQRT(v;u)	HSQRT SQRT DSQRT CSQRT	VHSQRT VSQRT VCSQRT	$a^{1/2}$	a ≥ 0	0 ≤ result
TAN(a)	VTAN(v;u)	HTAN TAN DTAN	VHTAN VTAN	tan(a)		
TANH(a)	-	HTANH TANH DTANH	-	tanh(a)		-1 ≤ result ≤ 1

ABS

ABS(a) is a generic scalar function that returns the absolute value of the argument. The argument can be of type integer, real, double-precision, half-precision, or complex. The result is of the same data type as the argument unless the argument is of type complex. For a complex argument, the result is of type real.

For an integer, real, double-precision, or half-precision argument, the result is |a|. For a complex argument, the result is the square root of (ar^2+ai^2).

ABS is also a specific scalar function that accepts a real argument and returns a real result. The other specific scalar function names are IABS, DABS, HABS, and CABS.

ACOS

ACOS(a) is a generic scalar function that returns the arccosine of the argument. The argument can be of type real, double-precision, or half-precision. The result is of the same data type as the argument and is expressed in radians. See table 10-3 for the domain and range of the function.

ACOS is also a specific scalar function that accepts a real argument and returns a real result. The other specific scalar function names are DACOS and HACOS.

AIMAG

AIMAG(a) is a specific scalar function that returns the imaginary part of the argument. The argument must be of type complex. The result is of type real. For a complex argument (ar,ai), the result is ai. There is no generic scalar function name.

AINT

AINT(a) is a generic scalar function that truncates the fractional part of the argument and returns the whole number part of the argument. The argument can be of type real, double-precision, or half-precision. The result is of the same data type as the argument.

The result is 0 if the absolute value of the argument is less than 1. If the absolute value of the argument is not less than 1, the result is the largest integer with the same sign as the argument that does not exceed the magnitude of the argument.

AINT is also a specific scalar function that accepts a real argument and returns a real result. The other specific scalar function names are DINT and HINT.

ALOG

ALOG(a) is a specific scalar function that returns the natural logarithm of the argument. The argument must be of type real. The result is of type real. See table 10-3 for the domain and range of the function. The generic scalar function name is LOG.

ALOG10

ALOG10(a) is a specific scalar function that returns the common logarithm of the argument. The argument must be of type real. The result is of type real. See table 10-3 for the domain and range of the function. The generic scalar function name is LOG10.

AMAX0

AMAX0($a_1,a_2,\ldots$) is a specific scalar function that returns the value of the largest argument. All of the arguments must be of type integer. The result is of type real. There is no generic scalar function name.

AMAX1

AMAX1($a_1,a_2,\ldots$) is a specific scalar function that returns the value of the largest argument. All of the arguments must be of type real. The result is of type real. The generic scalar function name is MAX.

AMIN0

AMIN0($a_1,a_2,\ldots$) is a specific scalar function that returns the value of the smallest argument. All of the arguments must be of type integer. The result is of type real. There is no generic scalar function name.

AMIN1

AMIN1($a_1,a_2,\ldots$) is a specific scalar function that returns the value of the smallest argument. All of the arguments must be of type real. The result is of type real. The generic scalar function name is MIN.

AMOD

AMOD(a_1,a_2) is a specific scalar function that returns the result of the first argument modulo the second argument. The arguments must be of type real. The result is of type real. See table 10-3 for the domain and range of the function. The generic scalar function name is MOD.

ANINT

ANINT(a) is a generic scalar function that returns the whole number that is nearest to the argument. The argument can be of type real, double-precision, or half-precision. The result is of the same data type as the argument.

ANINT is also a specific scalar function that accepts a real argument and returns a real result. The other specific scalar function names are DNINT and HNINT.

ASIN

ASIN(a) is a generic scalar function that returns the arcsine of the argument. The argument can be of type real, double-precision, or half-precision. The result is of the same data type as the argument and is expressed in radians. See table 10-3 for the domain and range of the function.

ASIN is also a specific scalar function that accepts a real argument and returns a real result. The other specific scalar function names are DASIN and HASIN.

ATAN

ATAN(a) is a generic scalar function that returns the arctangent of the argument. The argument can be of type real, double-precision, or half-precision. The result is of the same data type as the argument and is expressed in radians. See table 10-3 for the domain and range of the function.

ATAN is also a specific scalar function that accepts a real argument and returns a real result. The other specific scalar function names are DATAN and HATAN.

ATAN2

ATAN2(a_1,a_2) is a generic scalar function that returns the arctangent of the ratio of the two arguments. The arguments can be of type real, double-precision, or half-precision, but both arguments must be of the same data type. The arguments must not both be 0. The result is of the same data type as the arguments and is expressed in radians. See table 10-3 for the domain and range of the function.

ATAN2 is also a specific scalar function that accepts two real arguments and returns a real result. The other specific scalar function names are DATAN2 and HATAN2.

BTOL

BTOL(a) is a specific scalar function that converts a bit value into a logical value. The argument must be of type bit. The result is of type logical. If the argument has the value B'0', the function returns the value .FALSE.; if the argument has the value B'1', the function returns the value .TRUE.. There is no generic scalar function name.

CABS

CABS(a) is a specific scalar function that computes the modulus of the argument and returns a result that is greater than or equal to 0. The argument must be of type complex. The result is of type real. For a complex argument (ar,ai), the result is the square root of (ar^2+ai^2). The generic scalar function name is ABS.

CCOS

CCOS(a) is a specific scalar function that returns the cosine of the argument. The argument must be of type complex and is expressed in radians. The result is of type complex. See table 10-3 for the domain and range of the function. The generic scalar function name is COS.

CEXP

CEXP(a) is a specific scalar function that computes the exponential of the argument. The argument must be of type complex. The result is of type complex. See table 10-3 for the domain and range of the function. The generic scalar function name is EXP.

CHAR

CHAR(a) is a specific scalar function that returns the character represented by character code a in the ASCII character set. The argument must be of type integer. The value of the argument must be no less than 0 and no greater than 255. The result is of type character.

The argument is converted to hexadecimal, and the character that is represented internally by that hexadecimal value is returned.

For example, CHAR(65) returns the character A. (The integer 65 is converted to X'41', which is the internal hexadecimal representation of the character A.) See appendix A for the ASCII character code values. There is no generic scalar function name.

CLOG

CLOG(a) is a specific scalar function that returns the natural logarithm of the argument. The argument must be of type complex. The result is of type complex. See table 10-3 for the domain and range of the function. The generic scalar function name is LOG.

CMPLX

The scalar function CMPLX has two forms: CMPLX(a) and CMPLX(a_1,a_2).

CMPLX(a) is a generic scalar function that converts the argument into a complex value. The argument can be of type integer, real, double-precision, half-precision, or complex. The result is of type complex.

For an integer, real, double-precision, or half-precision argument, the result is a complex value whose real part is the value of the argument and whose imaginary part is 0. For a complex argument, the result is the value of the argument.

CMPLX(a_1,a_2) is a generic scalar function that converts the two arguments into a complex value. The arguments can be of type integer, real, double-precision, or half-precision, but both arguments must be of the same data type. The result is of type complex.

The result is a complex value whose real part is the value of the first argument and whose imaginary part is the value of the second argument.

CMPLX is also a specific scalar function that accepts a real argument and returns a complex result.

CONJG

CONJG(a) is a specific scalar function that returns the conjugate of the argument. The argument must be of type complex. The result is of type complex. For a complex number (ar,ai), the result is (ar,-ai). There is no generic scalar function name.

COS

COS(a) is a generic scalar function that returns the cosine of the argument. The argument can be of type real, double-precision, half-precision, or complex and is expressed in radians. The result is of the same data type as the argument. See table 10-3 for the domain and range of the function.

COS is also a specific scalar function that accepts a real argument and returns a real result. The other specific scalar function names are DCOS, HCOS, and CCOS.

COSH

COSH(a) is a generic scalar function that returns the hyperbolic cosine of the argument. The argument can be of type real, double-precision, or half-precision. The result is of the same data type as the argument. See table 10-3 for the domain and range of the function.

COSH is also a specific scalar function that accepts a real argument and returns a real result. The other specific scalar function names are DCOSH and HCOSH.

COTAN

COTAN(a) is a generic scalar function that returns the cotangent of the argument. The argument can be of type real or half-precision and is expressed in radians. COTAN first reduces its argument modulo 2pi. The result is of the same data type as the argument. See table 10-3 for the domain and range of the function.

COTAN is also a specific scalar function that accepts a real argument and returns a real result. The other specific scalar function name is HCOTAN.

CSIN

CSIN(a) is a specific scalar function that returns the sine of the argument. The argument must be of type complex and is expressed in radians. The result is of type complex. See table 10-3 for the domain and range of the function. The generic scalar function name is SIN.

CSQRT

CSQRT(a) is a specific scalar function that returns the square root of the argument. The argument must be of type complex. The real part of the argument must be greater than or equal to 0. The result is of type complex. See table 10-3 for the domain and range of the function. The generic scalar function name is SQRT.

DABS

DABS(a) is a specific scalar function that returns the absolute value of the argument. The argument must be of type double-precision. The result is of type double-precision. The generic scalar function name is ABS.

DACOS

DACOS(a) is a specific scalar function that returns the arccosine of the argument. The argument must be of type double-precision. The result is of type double-precision and is expressed in radians. See table 10-3 for the domain and range of the function. The generic scalar function name is ACOS.

DASIN

DASIN(a) is a specific scalar function that returns the arcsine of the argument. The argument must be of type double-precision. The result is of type double-precision and is expressed in radians. See table 10-3 for the domain and range of the function. The generic scalar function name is ASIN.

DATAN

DATAN(a) is a specific scalar function that returns the arctangent of the argument. The argument must be of type double-precision. The result is of type double-precision and is expressed in radians. See table 10-3 for the domain and range of the function. The generic scalar function name is ATAN.

DATAN2

DATAN2(a_1,a_2) is a specific scalar function that returns the arctangent of the ratio of the two arguments. The arguments must be of type double-precision. The result is of type double-precision and is expressed in radians. See table 10-3 for the domain and range of the function. The generic scalar function name is ATAN2.

DATE

DATE() is a specific scalar function that returns the current date. The parentheses are required, but an argument is not required. You can supply one argument of any data type, but the argument is ignored. The result is a character string of the form mm/dd/yy, where mm represents the month, dd represents the day, and yy represents the year.

There is no generic scalar function name.

DBLE

DBLE(a) is a generic scalar function that converts the argument into a double-precision value. The argument can be of type integer, real, double-precision, half-precision, or complex. The result is of type double-precision. There are no specific scalar function names.

DCOS

DCOS(a) is a specific scalar function that returns the cosine of the argument. The argument must be of type double-precision and is expressed in radians. The result is of type double-precision. See table 10-3 for the domain and range of the function. The generic scalar function name is COS.

DCOSH

DCOSH(a) is a specific scalar function that returns the hyperbolic cosine of the argument. The argument must be of type double-precision. The result is of type double-precision. See table 10-3 for the domain and range of the function. The generic scalar function name is COSH.

DDIM

DDIM(a_1,a_2) is a specific scalar function that returns the positive difference between the two arguments. The arguments must be of type double-precision. The result is of type double-precision. The result is the first argument minus the second argument; however, if the result is negative, the value 0 is returned. The generic scalar function name is DIM.

DEXP

DEXP(a) is a specific scalar function that returns the exponential of the argument. The argument must be of type double-precision. The result is of type double-precision. See table 10-3 for the domain and range of the function. The generic scalar function name is EXP.

DFLOAT

DFLOAT(i) is a specific scalar function that converts the argument to a double-precision value. The argument must be of type integer. The result is of type double-precision. The generic scalar function name is DBLE.

DIM

DIM(a_1,a_2) is a generic scalar function that returns the positive difference between the two arguments. The arguments can be of type integer, real, double-precision, or half-precision, but both arguments must be of the same data type. The result is of the same data type as the arguments.

The result is the first argument minus the second argument; however, if the result is negative, the value 0 is returned.

DIM is also a specific scalar function that accepts a real argument and returns a real result. The other specific scalar function names are IDIM, DDIM, and HDIM.

DINT

DINT(a) is a specific scalar function that truncates the fractional part of the argument and returns the whole number part of the argument. The argument must be of type double-precision. The result is of type double-precision.

The result is 0 if the absolute value of the argument is less than 1. If the absolute value of the argument is not less than 1, the result is the largest integer with the same sign as the argument that does not exceed the magnitude of the argument. The generic scalar function name is AINT.

DLOG

DLOG(a) is a specific scalar function that returns the natural logarithm of the argument. The argument must be of type double-precision. The result is of type double-precision. See table 10-3 for the domain and range of the function. The generic scalar function name is LOG.

DLOG10

DLOG10(a) is a specific scalar function that returns the common logarithm of the argument. The argument must be of type double-precision. The result is of type double-precision. See table 10-3 for the domain and range of the function. The generic scalar function name is LOG10.

DMAX1

DMAX1(a_1,a_2,...) is a specific scalar function that returns the value of the largest argument. All of the arguments must be of type double-precision. The result is of type double-precision. The generic scalar function name is MAX.

DMIN1

DMIN1(a_1,a_2,...) is a specific scalar function that returns the value of the smallest argument. All of the arguments must be of type double-precision. The result is of type double-precision. The generic scalar function name is MIN.

DMOD

DMOD(a_1,a_2) is a specific scalar function that returns the result of the first argument modulo the second argument. The arguments must be of type double-precision. The result is of type double-precision. See table 10-3 for the domain and range of the function. The generic scalar function name is MOD.

DNINT

DNINT(a) is a specific scalar function that returns the whole number that is nearest to the argument. The argument must be of type double-precision. The result is of type double-precision. The generic scalar function name is ANINT.

DPROD

DPROD(a_1,a_2) is a specific scalar function that returns the product of the two arguments. The arguments must be of type real. The result is of type double-precision. See table 10-3 for the domain and range of the function. There is no generic scalar function name.

DSIGN

DSIGN(a_1,a_2) is a specific scalar function that combines the absolute value of the first argument with the sign of the second argument. The arguments must be of type double-precision. The result is of type double-precision. The generic scalar function name is SIGN.

DSIN

DSIN(a) is a specific scalar function that returns the sine of the argument. The argument must be of type double-precision and is expressed in radians. The result is of type double-precision. See table 10-3 for the domain and range of the function. The generic scalar function name is SIN.

DSINH

DSINH(a) is a specific scalar function that returns the hyperbolic sine of the argument. The argument must be of type double-precision. The result is of type double-precision. See table 10-3 for the domain and range of the function. The generic scalar function name is SINH.

DSQRT

DSQRT(a) is a specific scalar function that returns the square root of the argument. The argument must be of type double-precision. The result is of type double-precision. See table 10-3 for the domain and range of the function. The generic scalar function name is SQRT.

DTAN

DTAN(a) is a specific scalar function that returns the tangent of the argument. The argument must be of type double-precision and is expressed in radians. DTAN first reduces its argument modulo 2pi. The result is of type double-precision. See table 10-3 for the domain and range of the function. The generic scalar function name is TAN.

DTANH

DTANH(a) is a specific scalar function that returns the hyperbolic tangent of the argument. The argument must be of type double-precision. The result is of type double-precision. See table 10-3 for the domain and range of the function. The generic scalar function name is TANH.

EXP

EXP(a) is a generic scalar function that returns the exponential of the argument. The argument can be of type real, double-precision, half-precision, or complex. The result is of the same data type as the argument. See table 10-3 for the domain and range of the function.

EXP is also a specific scalar function that accepts a real argument and returns a real result. The other specific scalar function names are DEXP, HEXP, and CEXP.

EXTEND

EXTEND(a) is a specific scalar function that converts the argument into a real value. The argument must be of type half-precision. The result is of type real. The generic scalar function name is REAL.

FLOAT

FLOAT(a) is a specific scalar function that converts the argument into a real value. The argument must be of type integer. The result is of type real. The generic scalar function name is REAL.

HABS

HABS(a) is a specific scalar function that returns the absolute value of the argument. The argument must be of type half-precision. The result is of type half-precision. The generic scalar function name is ABS.

HACOS

HACOS(a) is a specific scalar function that returns the arccosine of the argument. The argument must be of type half-precision. The result is of type half-precision and is expressed in radians. See table 10-3 for the domain and range of the function. The generic scalar function name is ACOS.

HALF

HALF(a) is a generic scalar function that converts the argument into a half-precision value. The argument can be of type integer, real, double-precision, half-precision, or complex. The result is of type half-precision. There are no specific scalar function names.

HASIN

HASIN(a) is a specific scalar function that returns the arcsine of the argument. The argument must be of type half-precision. The result is of type half-precision and is expressed in radians. See table 10-3 for the domain and range of the function. The generic scalar function name is ASIN.

HATAN

HATAN(a) is a specific scalar function that returns the arctangent of the argument. The argument must be of type half-precision. The result is of type half-precision and is expressed in radians. See table 10-3 for the domain and range of the function. The generic scalar function name is ATAN.

HATAN2

HATAN2(a_1,a_2) is a specific scalar function that returns the arctangent of the ratio of the two arguments. The arguments must be of type half-precision. The result is of type half-precision and is expressed in radians. See table 10-3 for the domain and range of the function. The generic scalar function name is ATAN2.

HCOS

HCOS(a) is a specific scalar function that returns the cosine of the argument. The argument must be of type half-precision and is expressed in radians. The result is of type half-precision. See table 10-3 for the domain and range of the function. The generic scalar function name is COS.

HCOSH

HCOSH(a) is a specific scalar function that returns the hyperbolic cosine of the argument. The argument must be of type half-precision. The result is of type half-precision. See table 10-3 for the domain and range of the function. The generic scalar function name is COSH.

HCOTAN

HCOTAN(a) is a specific scalar function that returns the cotangent of the argument. The argument must be of type half-precision and is expressed in radians. The result is of type half-precision. See table 10-3 for the domain and range of the function. The generic scalar function name is COTAN.

HDIM

HDIM(a_1,a_2) is a specific scalar function that returns the positive difference between the two arguments. The arguments must be of type half-precision. The result is of type half-precision. The result is the first argument minus the second argument; however, if the result is negative, the value 0 is returned. The generic scalar function name is DIM.

HEXP

HEXP(a) is a specific scalar function that returns the exponential of the argument. The argument must be of type half-precision. The result is of type half-precision. See table 10-3 for the domain and range of the function. The generic scalar function name is EXP.

HINT

HINT(a) is a specific scalar function that truncates the fractional part of the argument and returns the whole number part of the argument. The argument must be of type half-precision. The result is of type half-precision.

The result is 0 if the absolute value of the argument is less than 1. If the absolute value of the argument is not less than 1, the result is the largest integer with the same sign as the argument that does not exceed the magnitude of the argument. The generic scalar function name is AINT.

HLOG

HLOG(a) is a specific scalar function that returns the natural logarithm of the argument. The argument must be of type half-precision. The result is of type half-precision. See table 10-3 for the domain and range of the function. The generic scalar function name is LOG.

HLOG10

HLOG10(a) is a specific scalar function that returns the common logarithm of the argument. The argument must be of type half-precision. The result is of type half-precision. See table 10-3 for the domain and range of the function. The generic scalar function name is LOG10.

HMAX1

HMAX1(a_1,a_2,...) is a specific scalar function that returns the value of the largest argument. All of the arguments must be of type half-precision. The result is of type half-precision. The generic scalar function name is MAX.

HMIN1

HMIN1(a_1,a_2,...) is a specific scalar function that returns the value of the smallest argument. All of the arguments must be of type half-precision. The result is of type half-precision. The generic scalar function name is MIN.

HMOD

HMOD(a_1,a_2) is a specific scalar function that returns the result of the first argument modulo the second argument. The arguments must be of type half-precision. The result is of type half-precision. See table 10-3 for the domain and range of the function. The generic scalar function name is MOD.

HNINT

HNINT(a) is a specific scalar function that returns the whole number that is nearest to the argument. The argument must be of type half-precision. The result is of type half-precision. The generic scalar function name is ANINT.

HSIGN

HSIGN(a_1,a_2) is a specific scalar function that combines the magnitude of the first argument with the sign of the second argument. The arguments must be of type half-precision. The result is of type half-precision. The generic scalar function name is SIGN.

HSIN

HSIN(a) is a specific scalar function that returns the sine of the argument. The argument must be of type half-precision and is expressed in radians. The result is of type half-precision. See table 10-3 for the domain and range of the function. The generic scalar function name is SIN.

HSINH

HSINH(a) is a specific scalar function that returns the hyperbolic sine of the argument. The argument must be of type half-precision. The result is of type half-precision. See table 10-3 for the domain and range of the function. The generic scalar function name is SINH.

HSQRT

HSQRT(a) is a specific scalar function that returns the square root of the argument. The argument must be of type half-precision. The result is of type half-precision. See table 10-3 for the domain and range of the function. The generic scalar function name is SQRT.

HTAN

HTAN(a) is a specific scalar function that returns the tangent of the argument. The argument must be of type half-precision and is expressed in radians. The result is of type half-precision. See table 10-3 for the domain and range of the function. The generic scalar function name is TAN.

HTANH

HTANH(a) is a specific scalar function that returns the hyperbolic tangent of the argument. The argument must be of type half-precision. The result is of type half-precision. See table 10-3 for the domain and range of the function. The generic scalar function name is TANH.

IABS

IABS(a) is a specific scalar function that returns the absolute value of the argument. The argument must be of type integer. The result is of type integer. The generic scalar function name is ABS.

ICHAR

ICHAR(a) is a specific scalar function that returns the integer equivalent of the internal hexadecimal representation of character a in the ASCII character set. The argument must be of type character. The result is of type integer.

For example, CHAR('A') returns the integer 65. (The internal hexadecimal representation of the character A is X'41', which is 65 when converted to integer.) See appendix A for the ASCII character code values. There is no generic scalar function name.

IDIM

IDIM(a_1,a_2) is a specific scalar function that returns the positive difference between the two arguments. The arguments must be of type integer. The result is of type integer. The result is the first argument minus the second argument; however, if the result is negative or 0, the value 0 is returned. The generic scalar function name is DIM.

IDINT

IDINT(a) is a specific scalar function that converts the argument into an integer value. The argument must be of type double-precision. The result is of type integer. The result is the sign of the argument multiplied by the largest integer less than or equal to the absolute value of the argument. The generic scalar function name is INT.

IDNINT

IDNINT(a) is a specific scalar function that returns the integer that is nearest to the argument. The argument must be of type double-precision. The result is of type integer. The generic scalar function name is NINT.

IFIX

IFIX(a) is a specific scalar function that converts the argument into an integer value. The argument must be of type real. The result is of type integer. The result is the sign of the argument multiplied by the largest integer less than or equal to the absolute value of the argument. The generic scalar function name is INT.

IHINT

IHINT(a) is a specific scalar function that converts the argument into an integer value. The argument must be of type half-precision. The result is of type integer. The result is the sign of the argument multiplied by the largest integer less than or equal to the absolute value of the argument. The generic scalar function name is INT.

IHNINT

IHNINT(a) is specific scalar function that returns the integer that is nearest to the argument. The argument must be of type half-precision. The result is of type integer. The generic scalar function name is NINT.

INDEX

INDEX(a_1,a_2) is a specific scalar function that returns the character position in the first argument in which the second argument begins. The arguments must be of type character. The result is of type integer.

If character string a_2 appears in character string a_1 more than once, the starting position of the first occurrence of a_2 is returned. If character string a_2 does not appear in character string a_1, the value 0 is returned. If the length of a_1 is less than the length of a_2, the value 0 is returned.

For example, the result of INDEX('ABCDE','CDE') is 3. The result of INDEX('ABCDE','F') is 0.

There is no generic scalar function name.

INT

INT(a) is a generic scalar function that converts the argument into an integer value. The argument can be of type integer, real, double-precision, half-precision, or complex. The result is of type integer.

The result is the sign of the argument multiplied by the largest integer less than or equal to the absolute value of the argument. For an argument of type complex, the result is the sign of the real part of the argument multiplied by the largest integer less than or equal to the absolute value of the real part of the argument.

INT is also a specific scalar function that accepts a real argument and returns an integer result. The other specific scalar function names are IFIX, IDINT, and IHINT.

ISIGN

ISIGN(a_1,a_2) is a specific scalar function that combines the magnitude of the first argument with the sign of the second argument. The arguments must be of type integer. The result is of type integer. The generic scalar function name is SIGN.

LEN

LEN(a) is a specific scalar function that returns the length in characters of the argument. The argument must be of type character. The result is of type integer. For example, the result of LEN('ABCDE') is 5. The argument need not be defined at the time the function reference is executed. There is no generic scalar function name.

LGE

LGE(a_1,a_2) is a specific scalar function that returns a logical value indicating if the first argument is lexically greater than or equal to the second argument. The arguments must be of type character. If the arguments have different lengths, the shorter argument is extended on the right with blanks. The result is of type logical.

The arguments are compared character by character from left to right until two corresponding characters are unequal. If a character of a_1 is greater than the corresponding character in a_2, the logical value .TRUE. is returned. If a character of a_1 is less than the corresponding character in a_2, the logical value .FALSE. is returned. If all characters of a_1 are equal to the corresponding characters in a_2, the logical value .TRUE. is returned. See appendix A for the collating sequence.

For example, the result of LGE('CDC','CDC') is .TRUE.. The result of LGE('CYBER203','CYBER205') is .FALSE.. The result of LGE('CONTROL','CDC') is .TRUE..

There is no generic scalar function name.

LGT

LGT(a_1,a_2) is a specific scalar function that returns a logical value indicating if the first argument is lexically greater than the second argument. The arguments must be of type character. If the arguments have different lengths, the shorter argument is extended on the right with blanks. The result is of type logical.

The arguments are compared character by character from left to right until two corresponding characters are unequal. If a character of a_1 is greater than the corresponding character in a_2, the logical value .TRUE. is returned. If a character of a_1 is less than the corresponding character in a_2, the logical value .FALSE. is returned. If all characters of a_1 are equal to the corresponding characters in a_2, the logical value .FALSE. is returned. See appendix A for the collating sequence.

For example, the result of LGT('CDC','CDC') is .FALSE.. The result of LGT('CYBER203','CYBER205') is .FALSE.. The result of LGT('CONTROL','CDC') is .TRUE..

There is no generic scalar function name.

LLE

LLE(a_1,a_2) is a specific scalar function that returns a logical value indicating if the first argument is lexically less than or equal to the second argument. The arguments must be of type character. If the arguments have different lengths, the shorter argument is extended on the right with blanks. The result is of type logical.

The arguments are compared character by character from left to right until two corresponding characters are unequal. If a character of a_1 is less than the corresponding character in a_2, the logical value .TRUE. is returned. If a character of a_1 is greater than the corresponding character in a_2, the logical value .FALSE. is returned. If all characters of a_1 are equal to the corresponding characters in a_2, the logical value .TRUE. is returned. See appendix A for the collating sequence.

For example, the result of LLE('CDC','CDC') is .TRUE.. The result of LLE('CYBER203','CYBER205') is .TRUE.. The result of LLE('CONTROL','CDC') is .FALSE..

There is no generic scalar function name.

LLT

LLT(a_1,a_2) is a specific scalar function that returns a logical value indicating if the first argument is lexically less than the second argument. The arguments must be of type character. If the arguments have different lengths, the shorter argument is extended on the right with blanks. The result is of type logical.

The arguments are compared character by character from left to right until two corresponding characters are unequal. If a character of a_1 is less than the corresponding character in a_2, the logical value .TRUE. is returned. If a character of a_1 is greater than the corresponding character in a_2, the logical value .FALSE. is returned. If all characters of a_1 are equal to the corresponding characters in a_2, the logical value .FALSE. is returned. See appendix A for the collating sequence.

For example, the result of LLT('CDC','CDC') is .FALSE.. The result of LLT('CYBER203','CYBER205') is .TRUE.. The result of LLT('CONTROL','CDC') is .FALSE..

There is no generic scalar function name.

LOG

LOG(a) is a generic scalar function that returns the natural logarithm of the argument. The argument can be of type real, double-precision, half-precision, or complex. The result is of the same data type as the argument. See table 10-3 for the domain and range of the function.

The specific scalar function names are ALOG, DLOG, HLOG, and CLOG.

LOG10

LOG10(a) is a generic scalar function that returns the common logarithm of the argument. The argument can be of type real, double-precision, or half-precision. The result is of the same data type as the argument. See table 10-3 for the domain and range of the function.

The specific scalar function names are ALOG10, DLOG10, and HLOG10.

LTOB

LTOB(a) is a specific scalar function that converts a logical value into a bit value. The argument must be of type logical. The result is of type bit. If the argument has the value .FALSE., the function returns the value B'0'; if the argument has the value .TRUE., the function returns the value B'1'. There is no generic scalar function name.

MAX

MAX(a_1,a_2,...) is a generic scalar function that returns the value of the largest argument. The arguments can be of type integer, real, double-precision, or half-precision, but all of the arguments must be of the same data type. The result is of the same data type as the argument.

The specific scalar function names are MAX0, AMAX1, DMAX1, and HMAX1.

MAX0

MAX0(a_1,a_2,...) is a specific scalar function that returns the value of the largest argument. All of the arguments must be of type integer. The result is of type integer. The generic scalar function name is MAX.

MAX1

MAX1(a_1,a_2,...) is a specific scalar function that returns the value of the largest argument. All of the arguments must be of type real. The result is of type integer. There is no generic scalar function name.

MIN

MIN(a_1,a_2,...) is a generic scalar function that returns the value of the smallest argument. The arguments can be of type integer, real, double-precision, or half-precision, but all of the arguments must be of the same data type. The result is of the same data type as the argument.

The specific scalar function names are MIN0, AMIN1, DMIN1, and HMIN1.

MIN0

MIN0(a_1,a_2,...) is a specific scalar function that returns the value of the smallest argument. All of the arguments must be of type integer. The result is of type integer. The generic scalar function name is MIN.

MIN1

MIN1(a_1,a_2,...) is a specific scalar function that returns the value of the smallest argument. All of the arguments must be of type real. The result is of type integer. There is no generic scalar function name.

MOD

MOD(a_1,a_2) is a generic scalar function that returns the result of the first argument modulo the second argument. The arguments can be of type integer, real, double-precision, or half-precision, but both arguments must be of the same data type. The result is of the same data type as the arguments. See table 10-3 for the domain and range of the function.

MOD is also a specific scalar function that accepts an integer argument and returns an integer result. The other specific scalar function names are AMOD, DMOD, and HMOD.

NINT

NINT(a) is a generic scalar function that returns the integer that is nearest to the argument. The argument can be of type real, double-precision, or half-precision. The result is of type integer.

NINT is also a specific scalar function that accepts a real argument and returns an integer result. The other specific scalar function names are IDNINT and IHNINT.

Q8SCNT

Q8SCNT(v) is a specific scalar function that returns the number of 1 bits in the argument. The argument must be a vector of type bit. The result is of type integer.

For example, if bit vector v_1 consists of the elements 1 0 0 1 1, the result of Q8SCNT(v_1) is 3.

There is no generic scalar function name.

Q8SDFB

Q8SDFB(a_1,a_2) is a specific scalar function that tests the bits of the data flag branch register. The arguments must be constants of type integer. The first argument is the number of the bit to be tested. The bits of the data flag branch register are numbered from the left beginning with 0. The second argument can have one of the following values:

- 0 The bit being tested is not to be changed.
- 1 The bit being tested is to be changed to 0.
- 2 The bit being tested is to be changed to 1.
- 3 The bit being tested is to be changed to 0 if it is currently 1, or the bit being tested is to be changed to 1 if it is currently 0.

The result is of type logical. The result is .TRUE. if the bit being tested is 1; the result is .FALSE. if the bit being tested is 0. The action specified by the second argument is performed after the bit is tested.

For example, if the 10th bit of the data flag branch register is 1, the result of Q8SDFB(9,3) is .TRUE.. This function reference causes the 10th bit of the data flag branch register to be changed to 0.

Figure 11-10 shows the data flag branch register format.

There is no generic scalar function name.

Q8SDOT

Q8SDOT(v_1,v_2) is a generic scalar function that returns the dot product of the two arguments. The arguments must be vectors and can be of type integer, real, or half-precision. If the arguments have different lengths, the excess elements of the longer argument are ignored. The result is of the same data type as the arguments. The result is the sum of the products of corresponding elements of the vector arguments.

For example, if vector v_1 consists of the elements 0 1 3 and vector v_2 consists of the elements 2 2 2, the result of Q8SDOT(v_1,v_2) is (0*2)+(1*2)+(3*2), which is 8.

There are no specific scalar function names.

Q8SEQ

Q8SEQ(v_1,v_2) is a generic scalar function that returns the number of pairs of corresponding vector elements that precede the first pair of corresponding vector elements in which the element of the first argument equals the element of the second argument. If this condition is never true, the function returns the number of elements in the shorter vector argument. The arguments must be two vectors or one scalar and one vector. The arguments can be of type integer, real, or half-precision. If the arguments are vectors that have different lengths, the excess elements of the longer vector are ignored. The result is of type integer.

For example, if vector v_1 consists of the elements 0 1 3 4 and vector v_2 consists of the elements 2 2 2 4, the result of Q8SEQ(v_1,v_2) is 3. If scalar s_1 has the value 4 and vector v_2 consists of the elements 2 2 2 4, the result of Q8SEQ(s_1,v_2) is 3.

There are no specific scalar function names.

Q8SEXTB

Q8SEXTB(a,i_1,i_2) is a generic scalar function that extracts bits from one of the arguments. The argument a can be of type integer, real, or logical. The arguments i_1 and i_2 must be of type integer. The result is typeless.

The function extracts i_1 bits from input value a, beginning at bit position i_2. (Bits are numbered from the left beginning with 0.)

The Q8SEXTB function result is right-justified with binary zero fill when it is an integer. For example, if I and M are integer variables and I has the value 6, then the statement

```
M = Q8SEXTB(I,4,59)
```

has the result M = 0011, which is 3 in decimal. The form of the Q8SEXTB function result might change when the result is not an integer. In the example above, if M were a real variable, M would have the real variable representation of the number 3.

There are no specific scalar function names.

Q8SGE

Q8SGE(v_1,v_2) is a generic scalar function that returns the number of pairs of corresponding vector elements that precede the first pair of corresponding vector elements in which the element of the first argument is greater than or equal to the element of the second argument. If this condition is never true, the function returns the number of elements in the shorter vector argument. The arguments must be two vectors or one scalar and one vector. The arguments can be of type integer, real, or half-precision. If the arguments are vectors that have different lengths, the excess elements of the longer vector are ignored. The result is of type integer.

For example, if vector v_1 consists of the elements 0 1 3 4 and vector v_2 consists of the elements 2 2 2 4, the result of Q8SGE(v_1,v_2) is 2. If scalar s_1 has the value 4 and vector v_2 consists of the elements 2 2 2 4, the result of Q8SGE(s_1,v_2) is 3.

There are no specific scalar function names.

Q8SINSB

Q8SINSB(a_1,i_1,i_2,a_2) is a generic scalar function that creates a word and inserts bits into the word. The arguments a_1 and a_2 can be of type integer, real, or logical; they need not be of the same data type. The arguments i_1 and i_2 must be of type integer. The result is typeless.

The result is the value of a_2, except that i_1 bits beginning with bit i_2 are replaced with the rightmost i_1 bits of a_1. The values of the arguments are not changed. Bits are numbered from the left beginning with 0.

For example, if I is an integer variable that has the value 1 and N is an integer variable that has the value 2, the result of Q8SINSB(I,1,63,N) is 0011.

There are no specific scalar function names.

Q8SLEN

Q8SLEN(v) is a generic scalar function that returns the number of elements in the argument. The argument must be a vector and can be of type integer, real, half-precision, or complex. The result is of type integer. For an argument of type complex, the result is half the number of elements in the vector.

For example, if vector v_1 consists of the elements 0 1 3 4, the result of Q8SLEN(v_1) is 4.

There are no specific scalar function names.

Q8SLT

Q8SLT(v_1,v_2) is a generic scalar function that returns the number of pairs of corresponding vector elements that precede the first pair of corresponding vector elements in which the element of the first argument is less than the element of the second argument. If this condition is never true, the function returns the number of elements in the shorter vector argument. The arguments must be two vectors or one scalar and one vector. The arguments can be of type integer, real, or half-precision. If the arguments are vectors that have different lengths, the excess elements of the longer vector are ignored. The result is of type integer.

For example, if vector v_1 consists of the elements 0 1 3 4 and vector v_2 consists of the elements 2 2 2 4, the result of Q8SLT(v_1,v_2) is 0. If scalar s_1 has the value 4 and vector v_2 consists of the elements 2 2 2 4, the result of Q8SLT(s_1,v_2) is 4.

There are no specific scalar function names.

Q8SMAX

The scalar function Q8SMAX has two forms: Q8SMAX(v) and Q8SMAX(v,cv).

Q8SMAX(v) is a generic scalar function that returns the value of the largest element of the argument. The argument must be a vector and can be of type integer, real, or half-precision. The result is of the same data type as the argument.

For example, if vector v_1 consists of the elements 0 1 3 4, the result of Q8SMAX(v_1) is 4.

Q8SMAX(v,cv) is a generic scalar function that returns the value of the largest element of the argument v whose corresponding element in the control vector cv contains a 1 bit. The argument v must be a vector and can be of type integer, real, or half-precision. The argument cv, which is used as the control vector, must be a vector of type bit. The result is of the same data type as the argument v.

For example, if vector v_1 consists of the elements 0 1 3 4 and bit vector c_1 consists of the elements 1 1 1 0, the result of Q8SMAX(v_1,c_1) is 3.

There are no specific scalar function names.

Q8SMAXI

The scalar function Q8SMAXI has two forms: Q8SMAXI(v) and Q8SMAXI(v,cv).

Q8SMAXI(v) is a generic scalar function that returns the number of elements preceding the largest element of the argument. The argument must be a vector and can be of type integer, real, or half-precision. The result is of the same data type as the argument.

For example, if vector v_1 consists of the elements 0 1 3 4, the result of Q8SMAXI(v_1) is 3.

Q8SMAXI(v,cv) is a generic scalar function that returns the number of elements preceding the largest element of the argument v whose corresponding element in the control vector cv contains a 1 bit. The argument v must be a vector and can be of type integer, real, or half-precision. The argument cv, which is used as the control vector, must be a vector of type bit. The result is of the same data type as the argument v.

For example, if vector v_1 consists of the elements 0 1 3 4 and bit vector c_1 consists of the elements 1 0 1 0, the result of Q8SMAXI(v_1,c_1) is 2.

There are no specific scalar function names.

Q8SMIN

The scalar function Q8SMIN has two forms: Q8SMIN(v) and Q8SMIN(v,cv).

Q8SMIN(v) is a generic scalar function that returns the value of the smallest element of the argument. The argument must be a vector and can be of type integer, real, or half-precision. The result is of the same data type as the argument.

For example, if vector v_1 consists of the elements 0 1 3 4, the result of Q8SMIN(v_1) is 0.

Q8SMIN(v,cv) is a generic scalar function that returns the value of the smallest element of the argument v whose corresponding element in the control vector cv contains a 1 bit. The argument v must be a vector and can be of type integer, real, or half-precision. The argument cv, which is used as the control vector, must be a vector of type bit. The result is of the same data type as the argument v.

For example, if vector v_1 consists of the elements 0 1 3 4 and bit vector c_1 consists of the elements 0 1 1 1, the result of Q8SMIN(v_1,c_1) is 1.

There are no specific scalar function names.

Q8SMINI

The scalar function Q8SMINI has two forms: Q8SMINI(v) and Q8SMINI(v,cv).

Q8SMINI(v) is a generic scalar function that returns the number of elements preceding the smallest element of the argument. The argument must be a vector and can be of type integer, real, or half-precision. The result is of the same data type as the argument.

For example, if vector v_1 consists of the elements 1 2 3 4, the result of Q8SMINI(v_1) is 0.

Q8SMINI(v,cv) is a generic scalar function that returns the number of elements preceding the smallest element of the argument v whose corresponding element in the control vector cv contains a 1 bit. The argument v must be a vector and can be of type integer, real, or half-precision. The argument cv, which is used as the control vector, must be a vector of type bit. The result is of the same data type as the argument v.

For example, if vector v_1 consists of the elements 1 2 3 4 and bit vector c_1 consists of the elements 0 0 1 1, the result of Q8SMINI(v_1,c_1) is 2.

There are no specific scalar function names.

Q8SNE

Q8SNE(v_1,v_2) is a generic scalar function that returns the number of pairs of corresponding vector elements that precede the first pair of corresponding vector elements in which the element of the first argument does not equal the element of the second argument. If this condition is never true, the function returns the number of elements in the shorter vector. The arguments must be two vectors or one scalar and one vector. The arguments can be of type integer, real, or half-precision. If the arguments are vectors that have different lengths, the excess elements of the longer vector are ignored. The result is of type integer.

For example, if vector v_1 consists of the elements 0 1 3 4 and vector v_2 consists of the elements 2 2 2 4, the result of Q8SNE(v_1,v_2) is 0. If scalar s_1 has the value 4 and vector v_2 consists of the elements 4 4 4 2, the result of Q8SNE(s_1,v_2) is 3.

There are no specific scalar function names.

Q8SPROD

The scalar function Q8SPROD has two forms: Q8SPROD(v) and Q8SPROD(v,cv).

Q8SPROD(v) is a generic scalar function that returns the product of the elements of the argument. The argument must be a vector and can be of type integer, real, or half-precision. The result is of the same data type as the argument.

For example, if vector v_1 consists of the elements 1 3 7, the result of Q8SPROD(v_1) is (1*3*7), which is 21.

Q8SPROD(v,cv) is a generic scalar function that returns the product of the elements of the argument v whose corresponding element in the control vector cv contains a 1 bit. The argument v must be a vector and can be of type integer, real, or half-precision. The argument cv, which is used as the control vector, must be a vector of type bit. The result is of the same data type as the argument v.

For example, if vector v_1 consists of the elements 1 3 7 and bit vector c_1 consists of the elements 1 1 0, the result of Q8SPROD(v_1,c_1) is (1*3), which is 3.

There are no specific scalar function names.

Q8SSUM

The scalar function Q8SSUM has two forms: Q8SSUM(v) and Q8SSUM(v,cv).

Q8SSUM(v) is a generic scalar function that returns the sum of the elements of the argument. The argument must be a vector and can be of type integer, real, or half-precision. The result is of the same data type as the argument.

For example, if vector v_1 consists of the elements 1 3 7, the result of Q8SSUM(v_1) is (1+3+7), which is 11.

Q8SSUM(v,cv) is a generic scalar function that returns the sum of the elements of the argument v whose corresponding element in the control vector cv contains a 1 bit. The argument v must be a vector and can be of type integer, real, or half-precision. The argument cv, which is used as the control vector, must be a vector of type bit. The result is of the same data type as the argument v.

For example, if vector v_1 consists of the elements 1 3 7 and bit vector c_1 consists of the elements 1 1 0, the result of Q8SSUM(v_1,c_1) is (1+3), which is 4.

There are no specific scalar function names.

Q8VADJM

Q8VADJM(v;u) is a generic vector function that returns the averages of adjacent elements of the input argument. The input argument v must be a vector and can be of type real or half-precision. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result. The output argument must be one element shorter than the input argument, or longer.

The first element of the output argument is the average of the first and second elements of the input argument; the second element of the output argument is the average of the second and third elements of the input argument, and so on.

For example, if input argument v_1 is a vector that consists of the elements 0.0 2.0 4.0 6.0, the function reference Q8VADJM(v_1;u_1) assigns the values 1.0 3.0 5.0 to the output argument u_1.

There are no specific vector function names.

Q8VAVG

Q8VAVG(v_1,v_2;u) is a generic vector function that returns the averages of the corresponding elements of the two input arguments. The input arguments must be two vectors or one scalar and one vector. The input arguments can be of type real or half-precision. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result. All input arguments and output arguments must have the same length. If one of the input arguments is a scalar, it is treated as if it were a vector of the appropriate length with all elements having the value of the scalar.

The first element of the output argument is the average of the first element of one input argument and the first element of the other input argument; the second element of the output argument is the average of the second element of one input argument and the second element of the other input argument, and so on.

For example, if input argument v_1 is a vector that consists of the elements 0.0 2.0 4.0 6.0 and input argument v_2 is a vector that consists of the elements 2.0 4.0 6.0 8.0, the function reference Q8VAVG(v_1,v_2;u_1) assigns the values 1.0 3.0 5.0 7.0 to the output argument u_1. If the input argument v_1 is a vector that consists of the elements 0.0 2.0 4.0 6.0 and the input argument s_1 is a scalar that has the value 10.0, the function reference Q8VAVG(v_1,s_1;u_1) assigns the values 5.0 6.0 7.0 8.0 to the output argument u_1.

There are no specific vector function names.

Q8VAVGD

Q8VAVGD(v_1,v_2;u) is a generic vector function that returns the averages of the differences of corresponding elements of the two input arguments. The input arguments must be two vectors or one scalar and one vector. The input arguments can be of type real or half-precision. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result. All input arguments and output arguments must have the same length. If one of the input arguments is a scalar, it is treated as if it were a vector of the appropriate length with all elements having the value of the scalar.

The first element of the output vector is half of the difference of the first element of the first input argument minus the first element of the second input argument; the second element of the output vector is half of the difference of the second element of the first input argument minus the second element of the second input argument, and so on.

For example, if input argument v_1 is a vector that consists of the elements 2.0 4.0 6.0 8.0 and input argument v_2 is a vector that consists of the elements 0.0 1.0 2.0 3.0, the function reference Q8VAVGD(v_1,v_2;u_1) assigns the values 1.0 1.5 2.0 2.5 to the output argument ul. If the input argument v_1 is a vector that consists of the elements 2.0 4.0 6.0 8.0 and the input argument s_1 is a scalar that has the value 0.0, the function reference Q8VAVGD(v_1,s_1;u_1) assigns the values 1.0 2.0 3.0 4.0 to the output argument u_1.

There are no specific vector function names.

Q8VCMPRS

Q8VCMPRS(v,cv;u) is a generic vector function that creates a vector consisting of selected elements of the input argument v. The input argument v must be a vector and can be of type integer, real, or half-precision. The input argument cv, which is used as a control vector, must be a vector of type bit. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result. The input arguments must have the same length. The length of the output argument is determined by the number of 1 bits in cv.

The output argument consists of all of the elements of the input argument v whose corresponding elements in the control vector cv contain a 1 bit.

For example, if input argument v_1 is a vector that consists of the elements 2 4 6 8 and input argument cv_1 is a bit vector that consists of the elements 0 1 0 1, the function reference Q8VCMPRS(v_1,cv_1;u_1) assigns the values 4 8 to the output argument u_1.

There are no specific vector function names.

Q8VCTRL

Q8VCTRL(v,cv;u) is a generic vector function that replaces selected elements of the output argument with the corresponding elements of the input argument v. The input argument v must be a vector and can be of type integer, real, or half-precision. The input argument cv must be a vector of type bit. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result. All input arguments and output arguments must have the same length.

All of the elements of the output argument retain their previous values except for those elements whose corresponding elements in the control vector cv contain a 1 bit. Those elements whose corresponding elements in the control vector cv contain a 1 bit are replaced with the corresponding element from the input vector v.

For example, if input argument v_1 is a vector that consists of the elements 2 4 6 8, the input argument cv_1 is a bit vector that consists of the elements 0 1 0 1, and output argument u_1 is a vector that consists of the elements 4 5 8 9; the function reference Q8VCTRL(v_1,cv_1;u_1) assigns the values 4 4 8 8 to the output argument u_1.

There are no specific vector function names.

Q8VDCMPR

Q8VDCMPR(v_1,v_2,cv;u) is a generic vector function that replaces selected elements of the v_2 argument with the corresponding elements of the v_1 argument using the cv control argument. The input arguments v_1 and v_2 must be vectors and can be of type integer, real, or half precision. The input argument cv, which is used as a control vector, must be a vector of type bit. The output argument u can be a vector of the same data type as the input arguments v_1 and v_2, or an integer expression that specifies the length of the vector function result. The input arguments v_2 and cv and the output argument u must have the same length.

All of the elements of the input argument v_2 move to the output argument except for those elements where corresponding elements in the control vector cv contain a 1 bit; those elements are replaced with consecutive elements from the input argument v_1.

For example, if the input argument v_1 is a vector that consists of the elements 20 40, the input argument v_2 consists of the elements 1 2 3 4, and the input argument cv is a bit vector that consists of the elements 0 1 0 1; the function reference Q8VDCMPR(v_1,v_2,cv;u) assigns the values 1 20 3 40 to the output argument u.

There are no specific vector function names.

Q8VDELT

Q8VDELT(v;u) is a generic vector function that returns the differences of adjacent elements of the input argument. The input argument must be a vector and can be of type real or half-precision. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result. The output argument must be one element shorter than the input argument, or longer.

The first element of the output argument is the value of the second element of the input argument minus the first element of the input argument; the second element of the output argument is the value of the third element of the input argument minus the second element of the input argument, and so on.

For example, if input argument v_1 is a vector that consists of the elements 0.0 2.0 5.0 9.0, the function reference Q8VDELT(v_1;u_1) assigns the values 2.0 3.0 4.0 to the output argument u_1.

There are no specific vector function names.

Q8VEQI

Q8VEQI(v_1,v_2;u) is a generic vector function that returns the number of elements that precede the value of each element of the first input argument in the second input argument. The input arguments must be two vectors and can be of type real or half-precision. The output argument can be a vector of type integer, or an integer expression that specifies the length of the vector function result. The input arguments can have different lengths, but the output argument must not be shorter than input argument v_1.

The function searches argument v_2 for an element that is equal to the first element of v_1. If the function finds such an element, it assigns to the first element of the output argument the number of elements that precede that element in argument v_2. If the function does not find such an element, it returns the number of elements in argument v_2. The function repeats this operation for each element of argument v_1, assigning the result to the corresponding element of the output argument.

For example, if input argument v_1 is a vector that consists of the elements 1.0 2.0 3.0 4.0 and input argument v_2 is a vector that consists of the elements 4.0 5.0 1.0 2.0, the function reference Q8VEQI(v_1,v_2;u_1) assigns the values 2 3 4 0 to the output argument u_1.

There are no specific vector function names.

Q8VGATHP

Q8VGATHP(v,i,n;u) is a generic vector function that creates a vector consisting of selected elements of the input argument v. The input argument v must be a vector and can be of type integer, half-precision, or real. The input arguments i and n must be scalars of type integer. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

The input argument i determines which elements of input argument v are assigned to the output argument. Beginning with the first element of v, every ith element is assigned to the output argument, until n elements have been assigned.

The number of elements assigned is determined by the input argument n; the lengths of input argument v and the output argument have no effect on the number of elements assigned. If the value of n causes the length of either the input argument v or the output argument to be exceeded, results are unpredictable.

See figure 10-1 for an example of the Q8VGATHP function.

```
Arguments

   V1       10.0  19.0  11.0  15.0  0.0  2.0  5.0

   I1       2

   N1       4

Function Reference

   Q8VGATHP (V1,I1,N1;U1)

Result

   U1       10.0  11.0  0.0  5.0
```

Figure 10-1. Function Q8VGATHP Example

This function contrasts with the generic vector function Q8VSCATP.

There are no specific vector function names.

Q8VGATHR

Q8VGATHR(v,i;u) is a generic vector function that creates a vector consisting of selected elements of the input argument v. The input argument v must be a vector and can be of type integer, real, or half-precision. The input argument i must be a vector of type integer. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result. The input vector i and the output vector u must be the same length. The length of input vector v should not be less than the greatest integer in input vector i.

Each element of the output argument corresponds to an element in input argument i. The elements in input argument i indicate which elements of input argument v are assigned to the corresponding elements in the output argument. For example, if an element of i contains a 1, the first element of input argument v is assigned to the element of the output argument that corresponds to the element of i. An element of input argument v can be assigned to more than one element of the output argument and not all elements of the input argument v must be assigned to the output argument.

For example, if input argument v_1 is a vector that consists of the elements 2.0 4.0 6.0 8.0 9.0 and input argument i_1 is a vector that consists of the elements 1 4 4 2, the function reference Q8VGATHR(v_1,i_1;u_1) assigns the values 2.0 8.0 8.0 4.0 to the output argument u_1. See figure 10-2 for an illustration of this example.

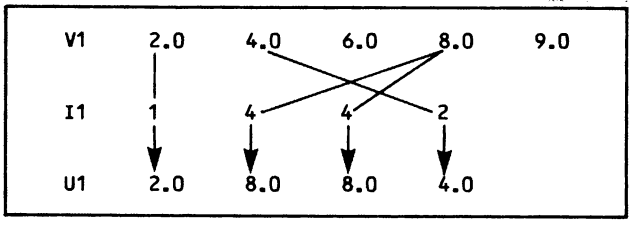


Figure 10-2. Q8VGATHR Function Example

This function contrasts with the generic vector function Q8VSCATR.

There are no specific vector function names.

Q8VGEI

Q8VGEI(v_1,v_2;u) is a generic vector function. For each element of the first argument, vector v_1, Q8VGEI returns a nonnegative integer that is a count of certain elements in the second argument, vector v_2. This count is the number of elements that precede the first element found in v_2 for which the element in v_1 is greater than or equal to the element of v_2. The two input vectors must both be the same type, either real or half precision. (The output vector does not have to be the same type.) The output argument u can be a vector of type integer, or an integer expression that specifies the length of the vector function result. The input arguments can have different lengths, but the output argument must not be shorter than the input argument v_1.

To create output argument u, Q8VGEI repeats the same search procedure for each element of argument v_1. It assigns the result of the procedure to the corresponding element of the output argument u. The paragraph below describes this search and assign process by showing how it works for finding the first element of the output argument u from the first element of argument v_1.

Q8VGEI obtains the result to put in the first element of the output argument u by searching argument v_2. The search compares successive elements of argument v_2 with the first element of v_1. The comparisons stop when the first element of v_1 is greater than or equal to an element of argument v_2. This element of argument v_2 is

the object of the search. The number of elements that precede that target element in argument v_2 is the result of the successful search. The function puts this result into the first element of argument u. If the search is unsuccessful, its result is the total number of elements in argument v_2, and the function assigns this number to the first element of argument u.

For example, if input argument v_1 is a vector that consists of the elements 1.0 0.0 9.0 4.5 and input argument v_2 is a vector that consists of the elements 5.0 4.0 1.0 2.0, the function reference Q8VGEI(v_1,v_2;u) assigns the values 2 4 0 1 to the output argument u. See figure 10-2.1 for an illustration of this example.

There are no specific vector function names.

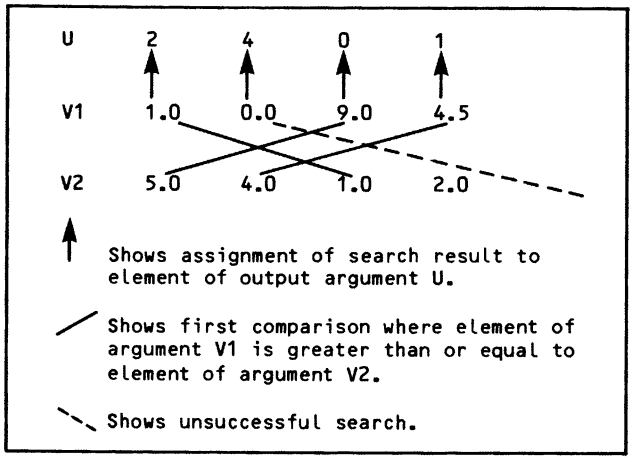


Figure 10-2.1. Q8VGEI Function Example

Q8VINTL

Q8VINTL(a_1,a_2;u) is a generic vector function that creates a vector whose adjacent elements differ by a specified interval. The input arguments must be two scalar expressions of type real, integer, or half precision. Both input arguments must be of the same type. The output argument can be a vector of the same type as the input argument or an integer expression that specifies the length of the vector function result. The output argument can be of any length.

The function assigns the value of input argument a_1 to the first element of the output argument. Each succeeding element of the output argument is assigned the value of the previous element plus the value of the input argument a_2.

For example, if input argument s_1 is a scalar that has the value 3, input argument s_2 is a scalar that has the value 2, and output argument u_1 is a vector that consists of four elements; the function reference Q8VINTL(s_1,s_2;u_1) assigns the values 3 5 7 9 to the output argument u_1.

There are no specific vector function names.

Q8VLTI

Q8VLTI(v_1,v_2;u) is a generic vector function. For each element of the first argument, vector v_1, Q8VLTI returns a nonnegative integer that is a count of certain elements in the second argument, vector v_2. This count is the number of elements that precede the first element found in v_2 for which the element in v_1 is less than the element of v_2. The two input vectors must both be the same type, either real or half precision. (The output vector does not have to be the same type.) The output argument u can be a vector of type integer, or an integer expression that specifies the length of the vector function result. The input arguments can have different lengths, but the output argument must not be shorter than the input argument v_1.

To create output argument u, Q8VLTI repeats the same search procedure for each element of argument v_1. It assigns the result of the procedure to the corresponding element of the output argument u. The paragraph below describes this search and assign process by showing how it works for finding the first element of the output argument u from the first element of argument v_1.

Q8VLTI obtains the result to put in the first element of the output argument u by searching argument v_2. The search compares successive elements of argument v_2 with the first element of v_1. The comparisons stop when the first element of v_1 is less than an element of argument v_2. This element of argument v_2 is the object of the search. The number of elements that precede that target element in argument v_2 is the result of the successful search. The function puts this result into the first element of argument u. If the search is unsuccessful, its result is the total number of elements in argument v_2, and the function assigns this number to the first element of argument u.

For example, if input argument v_1 is a vector that consists of the elements 3.0 0.0 9.0 5.0 and input argument v_2 is a vector that consists of the elements 2.0 1.0 4.0 5.0, the function reference Q8VLTI(v_1,v_2;u) assigns the values 2 0 4 4 to the output argument u. See figure 10-2.2 for an illustration of this example.

There are no specific vector function names.

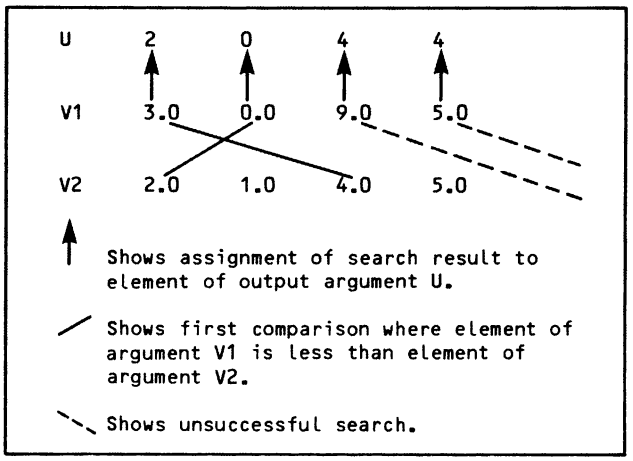


Figure 10-2.2. Q8VLTI Function Example

Q8VMASK

Q8VMASK(v_1,v_2,cv;u) is a generic vector function that creates a vector, each element of which has the same value as its corresponding element in input argument v_1 or input argument v_2. The input arguments v_1 and v_2 must be two vectors, a scalar and a vector, or two scalars. The input arguments v_1 and v_2 can be of type integer, real, or half-precision, but both input arguments must be of the same data type. The input argument cv, which is used as the control vector, must be a vector of type bit. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result. The length of the output argument must be the same as the length of the input argument cv, or longer. If input argument v_1 or input argument v_2 is a scalar, it is treated as if it were a vector of the appropriate length with all elements having the value of the scalar.

If an element in the control vector cv contains a 1 bit, the corresponding element in the output argument is assigned the value of the corresponding element in input argument v_1. If an element in the control vector cv contains a 0 bit, the corresponding element in the output argument is assigned the value of the corresponding element in the input argument v_2.

For example, if input argument v_1 is a vector that consists of the elements 1 2 3 4, input argument v_2 is a vector that consists of the elements 0 1 2 3, and input argument cv_1 is a bit vector that consists of the elements 0 1 0 1; the function reference Q8VMASK(v_1,v_2,cv_1;u_1) assigns the values 0 2 2 4 to the output argument u_1. If input argument s_1 is a scalar that has the value 5, input argument v_2 is a vector that consists of the elements 0 1 2 3, and input argument cv_1 is a bit vector that consists of the elements 0 1 0 1; the function reference Q8VMASK(s_1,v_2,cv_1;u_1) assigns the values 0 5 2 5 to the output argument u_1.

There are no specific vector function names.

Q8VMERG

Q8VMERG(v_1,v_2,cv;u) is a generic vector function that merges the input arguments v_1 and v_2 into a single output argument. The input arguments v_1 and v_2 must be two vectors. The input arguments v_1 and v_2 can be of type integer, real, or half-precision, but both input arguments must be of the same data type. The input argument cv, which is used as the control vector, must be a vector of type bit. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result. The length of the output argument must be the same as the length of the input argument cv, or longer.

If an element in the control vector cv contains a 1 bit, the corresponding element in the output argument is assigned the value of the first element in input argument v_1 that has not already been assigned to the output argument. If an element in the control vector cv contains a 0 bit, the corresponding element in the output argument is assigned the value of the first element in input argument v_2 that has not already been assigned to the output argument.

For example, if input argument v_1 is a vector that consists of the elements 1 5 7 9, input argument v_2 is a vector that consists of the elements 0 3 4 8, and input argument cv_1 is a bit vector that consists of the elements 0 1 0 0 1 1 0 1; the function reference Q8VMERG(v_1,v_2,cv_1;u_1) assigns the values 0 1 3 4 5 7 8 9 to the output argument u_1.

There are no specific vector function names.

Q8VMKO

Q8VMKO(a_1,a_2;u) is a generic vector function that creates a bit vector that consists of all 1 bits, all 0 bits, or a pattern of 1 bits and 0 bits beginning with a 1 bit. The input arguments must be two scalar constants of type integer. The output argument can be a vector of type bit, or an integer expression that specifies the length of the vector function result. The output argument can be of any length.

The function assigns a_1 1 bits followed by (a_2 -a_1) 0 bits to the output argument. This pattern of bits is repeated until the output argument is filled.

For example, if input argument s_1 is a scalar that has the value 1, input argument s_2 is a scalar that has the value 3, and output argument u_1 is a bit vector that consists of 10 elements; the function reference Q8VMKO(s_1,s_2;u_1) assigns the values 1 0 0 1 0 0 1 0 0 1 to the output argument u_1. If input argument s_1 is a scalar that has the value 0, input argument s_2 is a scalar that has the value 1, and output argument u_1 is a bit vector that consists of four elements; the function reference Q8VMKO(s_1,s_2;u_1) assigns the values 0 0 0 0 to the output argument u_1.

There are no specific vector function names.

Q8VMKZ

Q8VMKZ(a_1,a_2;u) is a generic vector function that creates a bit vector that consists of all 0 bits, all 1 bits, or a pattern of 0 bits and 1 bits beginning with a 0 bit. The input arguments must be two scalar constants of type integer. The output argument can be a vector of type bit, or an integer expression that specifies the length of the vector function result. The output argument can be of any length.

The function assigns a pattern of a_2 bits consisting of a_1 0 bits followed by (a_2 - a_1) 1 bits to the output argument. This pattern of bits is repeated until the output argument is filled.

For example, if input argument s_1 is a scalar that has the value 1, input argument s_2 is a scalar that has the value 3, and output argument u_1 is a bit vector that consists of 10 elements; the function reference Q8VMKZ(s_1,s_2;u_1) assigns the values 0 1 1 0 1 1 0 1 1 0 to the output argument u_1. If input argument s_1 is a scalar that has the value 0, input argument s_2 is a scalar that

has the value 1, and output argument u_1 is a bit vector that consists of four elements; the function reference Q8VMKZ(s_1,s_2;u_1) assigns the values 1 1 1 1 to the output argument u_1.

There are no specific vector function names.

Q8VNEI

Q8VNEI(v_1,v_2;u) is a generic vector function. For each element of the first argument, vector v_1, Q8VNEI returns a nonnegative integer that is a count of certain elements in the second argument, vector v_2. The count is of the elements that precede the first element in v_2 that is not equal to the element of v_1. The input arguments must be two vectors and can be of type real or half-precision. The output argument can be a vector of type integer, or an integer expression that specifies the length of the vector function result. The input arguments can have different lengths, but the output argument must not be shorter than the input argument v_1.

The function searches argument v_2 for an element that is not equal to the first element of v_1. If the function finds such an element, it assigns to the first element of the output argument the number of elements that precede that element in argument v_2. If the function does not find such an element, it returns the number of elements in argument v_2. The function repeats this operation for each element of argument v_1, assigning the result to the corresponding element of the output argument.

For example, if input argument v_1 is a vector that consists of the elements 4.0 2.0 4.0 2.0 and input argument v_2 is a vector that consists of the elements 4.0 4.0 1.0 2.0, the function reference Q8VNEI(v_1,v_2;u_1) assigns the values 2 0 2 0 to the output argument u_1.

There are no specific vector function names.

Q8VREV

Q8VREV(v;u) is a generic vector function that moves the elements of the input argument into the output argument such that the elements of the output argument are in reverse order. The input argument must be a vector and can be of type integer, real, or half-precision. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result. The output argument must be the same length as the input argument.

For example, if input argument v_1 is a vector that consists of the elements 1 2 3 4, the function reference Q8VREV(v_1;u_1) assigns the values 4 3 2 1 to the output argument u_1.

There are no specific vector function names.

Q8VSCATP

Q8VSCATP(v,i,n;u) is a generic vector function that replaces selected elements of an output vector with elements from another vector or with a scalar. The input argument v can be a vector or a scalar and can be of type integer, half-precision, or real. The input arguments i and n must be scalars of type integer. The output argument u must be a vector of the same data type as the input argument v.

If input argument v is a vector, the input arguments i and n determine which elements and the number of elements to be replaced. The first element of v replaces the first element of u. Succeeding elements of v replace every ith element thereafter of u, until n elements of u have been replaced. In general, for $1<=j<=n$, v_j replaces $u_{1+(j-1)*i}$.

See figure 10-3 for an example of the Q8VSCATP function, in which the input argument v is a vector. Input arguments i_1 and n_1 are integer scalars, and output argument u_1 is a real vector. The example shows the status of u_1 before and after the call to Q8VSCATP.

```
Arguments

   V1          0.0  50.0  -1.0  60.0

   I1          2

   N1          4

   U1          9.0  9.0  9.0  9.0  9.0  9.0  9.0  9.0

Function Reference

   Q8VSCATP (V1,I1,N1;U1)

Result

   U1          0.0  9.0  50.0  9.0  -1.0  9.0  60.0  9.0
```

Figure 10-3. Q8VSCATP Function Example With Vector Input Argument

If input argument v is a scalar, the input arguments i and n determine the elements of the output argument that are to be replaced by the input argument v. The value of v replaces the first element of the output argument, and every ith element thereafter, until n elements are replaced.

See figure 10-4 for an example of the Q8VSCATP function in which the input argument v is a scalar. The example shows the status of the output argument U1 before and after the call to Q8VSCATP.

The number of elements replaced in the output argument is determined by the value of the input argument n. The lengths of the input argument v or of the output argument have no effect on the number of elements replaced. If the value of n exceeds the number of elements in either the input argument v or the output argument, the results are unpredictable.

This function contrasts with the the generic vector function Q8VGATHP.

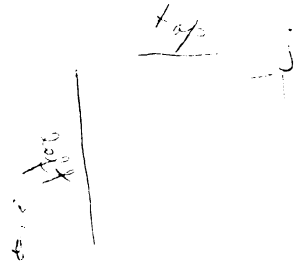
There are no specific vector function names.

Q8VSCATR

Q8VSCATR(v,i;u) is a generic vector function that creates a vector consisting of selected elements of the input argument v. The input argument v must be a vector and can be of type integer, real, or half-precision. The input argument i must be a vector of type integer. The output argument must be a vector of the same data type as the input argument v. The input arguments v and i should be the same length. The output argument length should be not less than the greatest integer in the input argument i.

```
Arguments

   V1          -1.0

   I1          2

   N1          4

   U1          9.0  9.0  9.0  9.0  9.0  9.0  9.0  9.0

Function Reference

   Q8VSCATP (V1,I1,N1;U1)

Result

   U1          -1.0  9.0  -1.0  9.0  -1.0  9.0  -1.0  9.0
```

Figure 10-4. Q8VSCATP Function Example With Scalar Input Argument



Each element of the input argument v corresponds to an element in input argument i. The elements in input argument i indicate to which elements in the output argument the elements in input argument v are assigned. For example, if an element of i contains a 1, the element of input argument v that corresponds to that element in i is assigned to the first element of the output argument. An element of the output argument can be assigned more than one value; the last value an element is assigned is the value that it retains.

For example, if input argument v_1 is a vector that consists of the elements 2.0 4.0 6.0 8.0, input argument i_1 is a vector that consists of the elements 1 4 4 2, and output argument u_1 is a vector that consists of the elements 9.0 9.0 9.0 9.0 9.0; the function reference Q8VSCATR(v_1,i_1;u_1) assigns the values 2.0 8.0 9.0 6.0 9.0 to the output argument u_1. The fourth element of the output argument is assigned the value 4.0, but is then reassigned the value 6.0. The third element of the output vector is never assigned; therefore, it retains its previous value. See figure 10-5 for an illustration of this example.

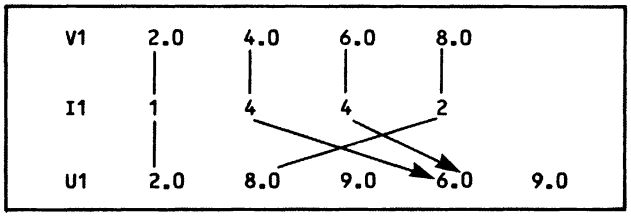


Figure 10-5. Q8VSCATR Function Example

This function contrasts with the generic vector function Q8VGATHR.

There are no specific vector function names.

Q8VXPND

Q8VXPND(v,cv;u) is a generic vector function that creates a vector that consists of the elements of input argument v plus additional elements having the value 0 or 0.0. The input argument v must be a vector of type integer, real, or half-precision. The input argument cv, which is used as the control vector, must be a vector of type bit. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result. The length of the output argument must be the same as the length of the input argument cv.

There is a one-to-one positional correspondence between the elements of the output vector (u) and the control vector (cv). For each position in the control vector (cv) that contains a 1, there will be a value in the output vector (u) in the same corresponding position. For each position in the control vector (cv) that contains a 0, there will be a 0 or 0.0 in the output vector (u) in the same corresponding position. The values that the control vector inserts into the output vector come from a third vector, the input vector. The input vector is an ordered list of the values to be placed in the output vector.

For example, if v_1 is an input vector comprising the ordered elements 5.0 15.0 25.0 and if cv_1 is a control vector comprising the elements 1 0 0 1, then Q8VXPND(v_1,cv_1;u_1) would create the output vector u_1 comprising the ordered elements 5.0 0.0 0.0 15.0.

There are no specific vector function names.

RANF

RANF() is a specific scalar function that returns a random number from 0 to 1. The parentheses are required, but an argument is not required. You can supply one argument of any data type, but the argument is ignored. The result is of type real. The multiplicative congruential method modulo 2^{47} is used to generate the next random number in the sequence:

$$x_{n+1} = (a*x_n) \bmod 2^{47}$$

The value of multiplier a is X'00004C65DA2C866D'. The seed can be obtained and reset by using the predefined subroutines RANGET and RANSET, respectively. The default value of the seed is X'000054F4A3B933BD'. A vector of random numbers can be generated by using the predefined subroutine VRANF.

There is no generic scalar function name.

REAL

REAL(a) is a generic scalar function that converts the argument into a real value. The argument can be of type integer, real, double-precision, half-precision, or complex. The result is of type real.

REAL is also a specific scalar function that accepts an integer argument and returns a real result. The other specific scalar function names are FLOAT, SNGL, and EXTEND.

RPROD

RPROD(a_1,a_2) is a specific scalar function that returns the product of the two arguments. The arguments must be of type half-precision. The result is of type real. See table 10-3 for the domain and range of the function. There is no generic scalar function name.

SECOND

SECOND() is a specific scalar function that returns the amount of central processor time that has elapsed since the job began. The parentheses are required, but an argument is not required. You can supply one argument of any data type, but the argument is ignored. The result is of type real. The result is expressed in seconds, and is precise to within 1 microsecond.

There is no generic scalar function name.

SIGN

SIGN(a_1,a_2) is a generic scalar function that combines the magnitude of the first argument with the sign of the second argument. The arguments can be of type integer, real, double-precision, or half-precision, but both arguments must be of the same data type. The result is of the same data type as the arguments.

SIGN is also a specific scalar function that accepts a real argument and returns a real result. The other specific scalar function names are ISIGN, DSIGN, and HSIGN.

SIN

SIN(a) is a generic scalar function that returns the sine of the argument. The argument can be of type real, double-precision, half-precision, or complex and is expressed in radians. The result is of the same data type as the argument. See table 10-3 for the domain and range of the function.

SIN is also a specific scalar function that accepts a real argument and returns a real result. The other specific scalar function names are DSIN, HSIN, and CSIN.

SINH

SINH(a) is a generic scalar function that returns the hyperbolic sine of the argument. The argument can be of type real, double-precision, or half-precision. The result is of the same data type as the argument. See table 10-3 for the domain and range of the function.

SINH is also a specific scalar function that accepts a real argument and returns a real result. The other specific scalar function names are DSINH and HSINH.

SNGL

SNGL(a) is a specific scalar function that converts the argument into a real value. The argument must be of type double-precision. The result is of type real. The generic scalar function name is REAL.

SQRT

SQRT(a) is a generic scalar function that returns the square root of the argument. The argument can be of type real, double-precision, half-precision, or complex. The result is of the same data type as the argument. See table 10-3 for the domain and range of the function.

SQRT is also a specific scalar function that accepts a real argument and returns a real result. The other specific scalar function names are DSQRT, HSQRT, and CSQRT.

TAN

TAN(a) is a generic scalar function that returns the tangent of the argument. The argument can be of type real, double-precision, or half-precision and is expressed in radians. The result is of the same data type as the argument. See table 10-3 for the domain and range of the function.

TAN is also a specific scalar function that accepts a real argument and returns a real result. The other specific scalar function names are DTAN and HTAN.

TANH

TANH(a) is a generic scalar function that returns the hyperbolic tangent of the argument. The argument can be of type real, double-precision, or half-precision. The result is of the same data type as the argument. See table 10-3 for the domain and range of the function.

TANH is also a specific scalar function that accepts a real argument and returns a real result. The other specific scalar function names are DTANH and HTANH.

TIME

TIME() is a specific scalar function that returns the current time. The parentheses are required, but an argument is not required. You can supply one argument of any data type, but the argument is ignored. The result is a character string of the form hh:mm:ss, where hh represents the hour, mm represents the minute, and ss represents the second.

There is no generic scalar function name.

VABS

VABS(v;u) is a generic vector function that returns the absolute value of each element of the input argument. The input argument must be a vector and can be of type integer, real, half-precision, or complex. The output argument can be a vector of the same data type as the input argument unless the input argument is of type complex. For a complex input argument, the output argument must be of type real. The output argument can be an integer that specifies the length of the vector function result.

Each element of the output argument is assigned the result of ABS(a), where a is the corresponding element in the input argument. ABS is a generic scalar function.

VABS is also a specific vector function that accepts a real vector input argument and returns a real vector output argument. The other specific vector function names are VIABS, VHABS, and VCABS.

VACOS

VACOS(v;u) is a generic vector function that returns the arccosine of each element of the input argument. The input argument must be a vector and can be of type real or half-precision. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of ACOS(a), where a is the corresponding element in the input argument. ACOS is a generic scalar function.

VACOS is also a specific vector function that accepts a real vector input argument and returns a real vector output argument. The other specific vector function name is VHACOS.

VAIMAG

VAIMAG(v;u) is a specific vector function that returns the imaginary part of each element of the input argument. The input argument must be a vector of type complex. The output argument can be a vector of type real, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of AIMAG(a), where a is the corresponding element in the input argument. AIMAG is a specific scalar function.

There is no generic vector function name.

VAINT

VAINT(v;u) is a generic vector function that truncates the fractional part of the each element of the input argument and returns the whole number part of each element of the input argument. The input argument must be a vector and can be of type real or half-precision. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of AINT(a), where a is the corresponding element in the input argument. AINT is a generic scalar function.

VAINT is also a specific vector function that accepts a real vector input argument and returns a real vector output argument. The other specific vector function name is VHINT.

VALOG

VALOG(v;u) is a specific vector function that returns the natural logarithm of each element of the input argument. The input argument must be a vector of type real. The output argument can be a vector of type real, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of ALOG(a), where a is the corresponding element in the input argument. ALOG is a specific scalar function.

The generic vector function name is VLOG.

VALOG10

VALOG10(v;u) is a specific vector function that returns the common logarithm of each element of the input argument. The input argument must be a vector of type real. The output argument can be a vector of type real, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of ALOG10(a), where a is the corresponding element in the input argument. ALOG10 is a specific scalar function.

The generic vector function name is VLOG10.

VAMOD

VAMOD(v_1,v_2;u) is a specific vector function that returns the result of the elements of first input argument modulo the corresponding elements in the second input argument. The input arguments must be vectors of type real. The output argument can be a vector of type real, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of AMOD(a_1,a_2), where a_1 and a_2 are corresponding elements of the input arguments v_1 and v_2 respectively. AMOD is a specific scalar function.

The generic vector function name is VMOD.

VANINT

VANINT(v;u) is a generic vector function that returns the whole number that is nearest to each element of the input argument. The input argument must be a vector and can be of type real or half-precision. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of ANINT(a), where a is the corresponding element in the input argument. ANINT is a generic scalar function.

VANINT is also a specific vector function that accepts a real vector input argument and returns a real vector output argument. The other specific vector function name is VHNINT.

VASIN

VASIN(v;u) is a generic vector function that returns the arcsine of each element of the input argument. The input argument must be a vector and can be of type real or half-precision. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of ASIN(a), where a is the corresponding element in the input argument. ASIN is a generic scalar function.

VASIN is also a specific vector function that accepts a real vector input argument and returns a real vector output argument. The other specific vector function name is VHASIN.

VATAN

VATAN(v;u) is a generic vector function that returns the arctangent of each element of the input argument. The input argument must be a vector and can be of type real or half-precision. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of ATAN(a), where a is the corresponding element in the input argument. ATAN is a generic scalar function.

VATAN is also a specific vector function that accepts a real vector input argument and returns a real vector output argument. The other specific vector function name is VHATAN.

VATAN2

VATAN2(v_1,v_2;u) is a generic vector function that returns the arctangent of the ratio of each corresponding pair of input arguments. The input arguments must be vectors and can be of type real or half-precision. Both input arguments must be of the same data type. Two corresponding elements of the input arguments must not both be 0. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of ATAN2(a_1,a_2), where a_1 and a_2 are the corresponding elements in input arguments v_1 and v_2. ATAN2 is a generic scalar function.

VATAN2 is also a specific vector function that accepts two real vector input arguments and returns a real vector output argument. The other specific vector function name is VHATAN2.

VCABS

VCABS(v;u) is a specific vector function that computes the modulus of each element of the input argument and returns results that are greater than or equal to 0. The input argument must be a vector of type complex. The output argument can be a vector of type real, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result CABS(a), where a is the corresponding element in the input argument. CABS is a specific scalar function.

The generic vector function name is VABS.

VCCOS

VCCOS(v;u) is a specific vector function that returns the cosine of each element of the input argument. The input argument must be a vector of type complex. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of CCOS(a), where a is the corresponding element in the input argument. CCOS is a specific scalar function.

The generic vector function name is VCOS.

VCEXP

VCEXP(v;u) is a specific vector function that computes the exponential of each element of the input argument. The input argument must be a vector of type complex. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of CEXP(a), where a is the corresponding element in the input argument. CEXP is a specific scalar function.

The generic vector function name is EXP.

VCLOG

VCLOG(v;u) is a specific vector function that returns the natural logarithm of each element of the input argument. The input argument must be a vector of type complex. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of CLOG(a), where a is the corresponding element in the input argument. CLOG is a specific scalar function.

The generic vector function name is VLOG.

VCMPLX

The vector function VCMPLX has two forms: VCMPLX(v;u) and VCMPLX(v_1,v_2;u).

VCMPLX(v;u) is a generic vector function that converts each element of the input argument into a complex value. The input argument must be a vector and can be of type integer, real, half-precision, or complex. The output argument can be a vector of type real, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of CMPLX(a), where a is the corresponding element in the input argument. CMPLX is a generic scalar function.

VCMPLX(v_1,v_2;u) is a generic vector function that converts each corresponding pair of elements in the input arguments into a complex value. The input arguments must be vectors and can be of type real or half-precision. Both input arguments must be of the same data type. The output argument can be a vector of type real, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of CMPLX(a_1,a_2), where a_1 and a_2 are corresponding elements in the input arguments. CMPLX is a generic scalar function.

There are no specific vector function names.

VCONJG

VCONJG(v;u) is a specific vector function that returns the conjugate of each element of the input argument. The input argument must be a vector of type complex. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of CONJG(a), where a is the corresponding element in the input argument. CONJG is a specific scalar function.

There is no generic vector function name.

VCOS

VCOS(v;u) is a generic vector function that returns the cosine of each element of the input argument. The input argument must be a vector and can be of type real, half-precision, or complex. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of COS(a), where a is the corresponding element in the input argument. COS is a generic scalar function.

VCOS is also a specific vector function that accepts a real vector input argument and returns a real vector output argument. The other specific vector function names are VHCOS and VCCOS.

VCSIN

VCSIN(v;u) is a specific vector function that returns the sine of each element of the input argument. The input argument must be a vector of type complex. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of CSIN(a), where a is the corresponding element in the input argument. CSIN is a specific scalar function.

The generic vector function name is VSIN.

VCSQRT

VCSQRT(v;u) is a specific vector function that returns the square root of each element of the input argument. The input argument must be a vector of type complex. The real part of each element of the input argument must be greater than or equal to 0. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of CSQRT(a), where a is the corresponding element in the input argument. CSQRT is a specific scalar function.

The generic vector function name is VSQRT.

VDBLE

VDBLE(v;u) is a generic vector function that converts each element of the input argument into a double-precision value. The input argument must be a vector and can be of type real. The output argument can be a vector of type double-precision, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of DBLE(a), where a is the corresponding element in the input argument. DBLE is a generic scalar function.

There are no specific vector function names.

VDIM

VDIM(v_1,v_2;u) is a generic vector function that returns the positive difference between each corresponding pair of input arguments. The input arguments must be vectors and can be of type integer, real, or half-precision. Both input arguments must be of the same data type. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of DIM(a_1,a_2), where a_1 and a_2 are corresponding elements of the input arguments. DIM is a generic scalar function.

VDIM is also a specific vector function that accepts a real vector input argument and returns a real vector output argument. The other specific vector function names are VIDIM and VHDIM.

VEXP

VEXP(v;u) is a generic vector function that returns the exponential of each element of the input argument. The input argument must be a vector and can be of type real, half-precision, or complex. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of EXP(a), where a is the corresponding element in the input argument. EXP is a generic scalar function.

VEXP is also a specific vector function that accepts a real vector input argument and returns a real vector output argument. The other specific vector function names are VHEXP and VCEXP.

VEXTEND

EXTEND(v;u) is a specific vector function that converts each element of the input argument into a real value. The input argument must be a vector of type half-precision. The output argument can be a vector of type real, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of EXTEND(a), where a is the corresponding element in the input argument. EXTEND is a specific scalar function.

The generic vector function name is VREAL.

VFLOAT

VFLOAT(v;u) is a specific vector function that converts each element of the input argument into a real value. The input argument must be a vector of type integer. The output argument can be a vector of type real, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of FLOAT(a), where a is the corresponding element in the input argument. FLOAT is a specific scalar function.

The generic vector function name is VREAL.

VHABS

VHABS(v;u) is a specific vector function that returns the absolute value of each element of the input argument. The input argument must be a vector of type half-precision. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of HABS(a), where a is the corresponding element in the input argument. HABS is a specific scalar function.

The generic vector function name is VABS.

VHACOS

VHACOS(v;u) is a specific vector function that returns the arccosine of each element of the input argument. The input argument must be a vector of

type half-precision. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of HACOS(a), where a is the corresponding element in the input argument. HACOS is a specific scalar function.

The generic vector function name is VACOS.

VHALF

VHALF(v;u) is a generic vector function that converts each element of the input argument into a half-precision value. The input argument must be a vector and can be of type integer, real, half-precision, or complex. The output argument can be a vector of type half-precision, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of HALF(a), where a is the corresponding element in the input argument. HALF is a generic scalar function.

There are no specific vector function names.

VHASIN

VHASIN(v;u) is a specific vector function that returns the arcsine of each element of the input argument. The input argument must be a vector of type half-precision. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of HASIN(a), where a is the corresponding element in the input argument. HASIN is a specific scalar function.

The generic vector function name is VASIN.

VHATAN

VHATAN(v;u) is a specific vector function that returns the arctangent of each element of the input argument. The input argument must be a vector of type half-precision. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of HATAN(a), where a is the corresponding element in the input argument. HATAN is a specific scalar function.

The generic vector function name is VATAN.

VHATAN2

VHATAN2(v_1,v_2;u) is a specific vector function that returns the arctangent of the ratio of each corresponding pair of input arguments. The input arguments must be vectors of type half-precision. Two corresponding elements of the input arguments must not both be 0. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of HATAN2(a_1,a_2), where a_1 and a_2 are corresponding elements of the input arguments. HATAN2 is a specific scalar function.

The generic vector function name is VATAN2.

VHCOS

VHCOS(v;u) is a specific vector function that returns the cosine of each element of the input argument. The input argument must be a vector of type half-precision. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of HCOS(a), where a is the corresponding element in the input argument. HCOS is a specific scalar function.

The generic vector function name is VCOS.

VHDIM

VHDIM(v_1,v_2;u) is a specific vector function that returns the positive difference between each pair of corresponding input arguments. The input arguments must be vectors of type half-precision. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of HDIM(a_1,a_2), where a_1 and a_2 are corresponding elements of the input arguments. HDIM is a specific scalar function.

The generic vector function name is VDIM.

VHEXP

VHEXP(v;u) is a specific vector function that returns the exponential of each element of the input argument. The input argument must be a vector of type half-precision. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of HEXP(a), where a is the corresponding element in the input argument. HEXP is a specific scalar function.

The generic vector function name is VEXP.

VHINT

VHINT(a) is a specific vector function that truncates the fractional part of each element of the input argument and returns the whole number part of each element of the input argument. The input argument must be a vector of type half-precision. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of HINT(a), where a is the corresponding element in the input argument. HINT is a specific scalar function.

The generic vector function name is VAINT.

VHLOG

VHLOG(v;u) is a specific vector function that returns the natural logarithm of each element of the input argument. The input argument must be a vector of type half-precision. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of HLOG(a), where a is the corresponding element in the input argument. HLOG is a specific scalar function.

The generic vector function name is VLOG.

VHLOG10

VHLOG10(v;u) is a specific vector function that returns the common logarithm of each element of the input argument. The input argument must be a vector of type half-precision. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of HLOG10(a), where a is the corresponding element in the input argument. HLOG10 is a specific scalar function.

The generic vector function name is VLOG10.

VHMOD

VHMOD(v_1,v_2;u) is a specific vector function that returns the result of the elements of first input argument modulo the corresponding elements in the second input argument. The input arguments must be vectors of type half-precision. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of HMOD(a_1,a_2), where a_1 and a_2 are corresponding elements in the input argument. HMOD is a specific scalar function.

The generic vector function name is VMOD.

VHNINT

VHNINT(v;u) is a specific vector function that returns the whole number that is nearest to each element of the input argument. The input argument must be a vector of type half-precision. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of HNINT(a), where a is the corresponding element in the input argument. HNINT is a specific scalar function.

The generic vector function name is VANINT.

VHSIGN

VHSIGN(v_1,v_2;u) is a specific vector function that combines the magnitude of each element of the first input argument with the sign of the corresponding element in the second input argument. The input arguments must be vectors of type half-precision. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of HSIGN(a_1,a_2), where a_1 and a_2 are the corresponding elements in the input arguments. HSIGN is a specific scalar function.

The generic vector function name is VSIGN.

VHSIN

VHSIN(v;u) is a specific vector function that returns the sine of each element of the input argument. The input argument must be a vector of type half-precision. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of HSIN(a), where a is the corresponding element in the input argument. HSIN is a specific scalar function.

The generic vector function name is VSIN.

VHSQRT

VHSQRT(v;u) is a specific vector function that returns the square root of each element of the input argument. The input argument must be a vector of type half-precision. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of HSQRT(a), where a is the corresponding element in the input argument. HSQRT is a specific scalar function.

The generic vector function name is VSQRT.

VHTAN

VHTAN(v;u) is a specific vector function that returns the tangent of each element of the input argument. The input argument must be a vector of type half-precision. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of HTAN(a), where a is the corresponding element in the input argument. HTAN is a specific scalar function.

The generic vector function name is VTAN.

VIABS

VIABS(v;u) is a specific vector function that returns the absolute value of each element of the input argument. The input argument must be a vector of type integer. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of IABS(a), where a is the corresponding element in the input argument. IABS is a specific scalar function.

The generic vector function name is VABS.

VIDIM

VIDIM(v_1,v_2;u) is a specific vector function that returns the positive difference between each corresponding pair of input arguments. The input arguments must be vectors of type integer. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of IDIM(a_1,a_2), where a_1 and a_2 are the corresponding elements in the input arguments. IDIM is a specific scalar function.

The generic vector function name is VDIM.

VIFIX

VIFIX(v;u) is a specific vector function that converts each element of the input argument into an integer value. The input argument must be a vector of type real. The output argument can be a vector of type integer, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of IFIX(a), where a is the corresponding element in the input argument. IFIX is a specific scalar function.

The generic vector function name is VINT.

VIHINT

VIHINT(v;u) is a specific vector function that converts each element of the input argument into an integer value. The input argument must be a vector of type half-precision. The output argument can be a vector of type integer, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of IHINT(a), where a is the corresponding element in the input argument. IHINT is a specific scalar function.

The generic vector function name is VINT.

VIHNINT

VIHNINT(v;u) is specific vector function that returns the integer that is nearest to each element of the input argument. The input argument must be a vector of type half-precision. The output argument can be a vector of type integer, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of IHNINT(a), where a is the corresponding element in the input argument. IHNINT is a specific scalar function.

The generic vector function name is VNINT.

VINT

VINT(v;u) is a generic vector function that converts each element of the input argument into an integer value. The input argument must be a vector and can be of type integer, real, half-precision, or complex. The output argument can be a vector of type integer, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of INT(a), where a is the corresponding element in the input argument. INT is a generic scalar function.

VINT is also a specific vector function that accepts a real vector input argument and returns an integer vector output argument. The other specific vector function names are VIFIX and VIHINT.

VISIGN

VISIGN(v_1,v_2;u) is a specific vector function that combines the magnitude of each element of the first input argument with the sign of the corresponding element in the second input argument. The

input arguments must be vectors of type integer. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of ISIGN(a_1,a_2), where a_1 and a_2 are the corresponding elements in the input arguments. ISIGN is a specific scalar function.

The generic vector function name is VSIGN.

VLOG

VLOG(v;u) is a generic vector function that returns the natural logarithm of each element of the input argument. The input argument must be a vector and can be of type real, half-precision, or complex. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of LOG(a), where a is the corresponding element in the input argument. LOG is a generic scalar function.

The specific vector function names are VALOG, VHLOG, and VCLOG.

VLOG10

VLOG10(v;u) is a generic vector function that returns the common logarithm of each element of the input argument. The input argument must be a vector and can be of type real or half-precision. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of LOG10(a), where a is the corresponding element in the input argument. LOG10 is a generic scalar function.

The specific vector function names are VALOG10 and VHLOG10.

VMOD

VMOD(v_1,v_2;u) is a generic vector function that returns the result of the elements of first input argument modulo the corresponding elements in the second input argument. The input arguments must be vectors and can be of type integer, real, or half-precision. Both input arguments must be of the same data type. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of MOD(a_1,a_2), where a_1 and a_2 are the corresponding elements in the input arguments. MOD is a generic scalar function.

VMOD is also a specific vector function that accepts an integer vector input argument and returns an integer vector output argument. The other specific vector function names are VAMOD and VHMOD.

VNINT

VNINT(v;u) is a generic vector function that returns the integer that is nearest to each element of the input argument. The input argument must be a vector and can be of type real, or half-precision. The output argument can be a vector of type integer, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of NINT(a), where a is the corresponding element in the input argument. NINT is a generic scalar function.

VNINT is also a specific vector function that accepts a real vector input argument and returns an integer vector output argument. The other specific vector function name is VIHNINT.

VRAND

VRAND (i;u) is a specific vector function that returns a vector of random numbers. The input argument is required, but its value is ignored. The output argument is either a real vector or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the next random number. VRAND, RANF, and VRANF return consecutive random numbers from the same sequence.

VRAND operates most efficiently if called with the same result vector length and RANF, VRANF, and RANSET are not called.

VREAL

VREAL(v;u) is a generic vector function that converts each element of the input argument into a real value. The input argument must be a vector and can be of type integer, real, half-precision, or complex. The output argument can be a vector of type real, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of REAL(a), where a is the corresponding element in the input argument. REAL is a generic scalar function.

VREAL is also a specific vector function that accepts an integer vector input argument and returns a real vector output argument. The other specific vector function names are VFLOAT, VSNGL, and VEXTEND.

VSIN

VSIN(v;u) is a generic vector function that returns the sine of each element of the input argument. The input argument must be a vector and can be of type real, half-precision, or complex. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of SIN(a), where a is the corresponding element in the input argument. SIN is a generic scalar function.

VSIN is also a specific vector function that accepts a real vector input argument and returns a real vector output argument. The other specific vector function names are VHSIN and VCSIN.

VSNGL

VSNGL(v;u) is a specific vector function that converts each element of the input argument into a real value. The input argument must be a vector of type double-precision. The output argument can be a vector of type real, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of SNGL(a), where a is the corresponding element in the input argument. SNGL is a specific scalar function.

The generic vector function name is VREAL.

VSQRT

VSQRT(v;u) is a generic vector function that returns the square root of each element of the input argument. The input argument must be a vector and can be of type real, half-precision, or complex. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of SQRT(a), where a is the corresponding element in the input argument. SQRT is a generic scalar function.

VSQRT is also a specific vector function that accepts a real vector input argument and returns a real vector output argument. The other specific vector function names are VHSQRT and VCSQRT.

VTAN

VTAN(v;u) is a generic vector function that returns the tangent of each element of the input argument. The input argument must be a vector and can be of type real or half-precision. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of TAN(a), where a is the corresponding element in the input argument. TAN is a generic scalar function.

VTAN is also a specific vector function that accepts a real vector input argument and returns a real vector output argument. The other specific vector function name is VHTAN.

VECTOR INTRINSIC FUNCTION EXAMPLES

The following paragraphs describe two sets of examples that use the vector intrinsic functions. The first set of examples demonstrates the bit manipulation functions. The second set of examples uses vector functions to vectorize DO loops.

BIT MANIPULATION FUNCTION EXAMPLES

Table 10-4 shows examples using the bit manipulation functions (Q8VMKZ, Q8VCTRL, Q8VCMPRS, Q8VXPND, Q8VMASK, and Q8MERG).

RESTRUCTURING DO LOOPS AS VECTOR OPERATIONS

The vector functions described in this section can be used to restructure DO loops as vector operations. The following paragraphs provide examples of how this is done. (For background information on vector programming, refer to section 9.)

Using a Bit Vector as a Mask

Suppose a calculation is to be performed using pairs of operands from two arrays, but not on every pair of operands in the arrays. To restructure the calculation as a vector operation, create a bit vector that serves as a mask indicating the pairs of operands on which the calculation is performed.

For example, consider the following DO loop:

```
      DO 10 I=1,N
      IF (TEST(I) .LT. EPSILON) GO TO 10
      A = X(I) * Y(I) + 3.1
      B = X(I) / Y(I) - 2.9
      R(I) = SQRT(A ** 2 - B)
   10 CONTINUE
```

The operand vectors in the DO loop are arrays X and Y; the result vector is array R. The test TEST(I) .LT. EPSILON determines the operand pairs on which the calculation is performed. If the test is true, the calculation is not performed on the operand pair; if the test is false, the calculation is performed on the operand pair.

To restructure the DO loop so that it becomes a vector operation, create a bit vector containing the results of the test for values 1 through N. Each 1 bit in the bit vector marks an operand pair on which the calculation is performed. The restructured code is shown in figure 10-6.

Table 10-4. Bit Manipulation Functions

Function	FORTRAN Statements	Function Input and Output
Q8VMKZ (Create bit mask)	INTEGER X/2/, Y/5/, L/14/ BIT B(14) B(1;14) = Q8VMKZ(X, Y; L)	X: 2 L: 14 Y: 5 B: 00111001110011
Q8VCTRL (Controlled store)	INTEGER A(5)/7,5,6,7,9/, R(5)/4,3,2,3,2/ BIT B(5)/B'01001'/ R(1;5) = Q8VCTRL(A(1;5),B(1;5);R(1;5))	A: 7 5 6 7 9 B: 0 1 0 0 1 R: 4 3 2 3 2 R: 4 5 2 3 9
Q8VCMPRS (Compress)	INTEGER A(5)/1,2,3,4,5/, R(2) BIT B(5)/B'01001'/ R(1;2) = Q8VCMPRS(A(1;5),B(1;5);R(1;2))	A: 1 2 3 4 5 B: 0 1 0 0 1 R: 2 5
Q8VXPND (Expand)	INTEGER A(3)/3,9,7/, R(5) BIT B(5)/B'01011'/ R(1;5) = Q8VXPND(A(1;3),B(1;5);R(1;5))	A: 3 9 7 B: 0 1 0 1 1 R: 0 3 0 9 7
Q8VMASK (Mask)	INTEGER C(5)/1,3,2,3,1/, D(5)/4,6,9,8,7/,R(5) BIT B(5)/B'10011'/ R(1;5) = Q8VMASK(C(1;5),D(1;5),B(1;5);R(1;5))	C: 1 3 2 3 1 D: 4 6 9 8 7 B: 1 0 0 1 1 R: 1 6 9 3 1
Q8VMERG (Merge)	INTEGER C(5)/1,3,2,4,0/, D(5)/5,6,9,8,7/,R(5) BIT B(5)/B'10011'/ R(1;5) = Q8VMERG(C(1;5),D(1;5),B(1;5);R(1;5))	C: 1 3 2 4 0 D: 5 6 9 8 7 B: 1 0 0 1 1 R: 1 5 6 3 2

```
DIMENSION VX(N), VY(N), VA(N), VB(N)      <-- Declares additional arrays required for the vector operation
BIT BITV(N)                                   and the bit vector.

BITV(1;N) = TEST(1;N) .GE. EPSILON        <-- Performs the comparison for each element of TEST and stores
                                              a 1 bit in the corresponding element of BITV if the test is
                                              true.

L = Q8SCNT(BITV(1;N))                     <-- Counts the 1 bits in the bit vector.

VX(1;L) = Q8VCMPRS(X(1;N),BITV(1;N);VX(1;L))  <-- Creates operand vectors containing only those operands on
VY(1;L) = Q8VCMPRS(Y(1;N),BITV(1;N);VY(1;L))      which the calculation is to be performed.

VA(1;L) = VX(1;L) * VY(1;L) + 3.1         <-- Performs the calculation.
VB(1;L) = VX(1;L) / VY(1;L) - 2.9
VA(1;L) = VA(1;L) * VA(1;L) - VB(1;L)
VB(1;L) = VSQRT(VA(1;L);VB(1;L))

R(1;N) = Q8VXPND(VB(1;L),BITV(1;N);R(1;N))  <-- Expands the result vector. For each 1 bit in the bit
                                                vector, a calculation result is stored in the R array. For
                                                each 0 bit in the bit vector, a zero value is stored in the
                                                R array.
```

Figure 10-6. Bit Vector Mask Example

VSIGN

VSIGN(v_1,v_2;u) is a generic vector function that combines the magnitude of each element of the first input argument with the sign of the corresponding element in the second input argument. The input arguments must be vectors and can be of type integer, real, or half-precision. Both input arguments must be of the same data type. The output argument can be a vector of the same data type as the input argument, or an integer expression that specifies the length of the vector function result.

Each element of the output argument is assigned the result of SIGN(a_1,a_2), where a_1 and a_2 are the corresponding elements in the input arguments. SIGN is a generic scalar function.

VSIGN is also a specific vector function that accepts a real vector input argument and returns a real vector output argument. The other specific vector function names are VISIGN and VHSIGN.

Restructuring DO Loops With Nonunit Stride

Another type of DO loop calculation in which not every pair of operands is used is the DO loop for which the index increment is not 1 (nonunit stride). Once again, to restructure the DO loop as a vector operation, create a bit vector to act as a mask indicating the operand pairs on which the calculation is performed.

For example, consider the following DO loop:

```
      DO 10 I=1,N,2
      A(I) = B(I) + 2.0 * C(I)
   10 CONTINUE
```

The operand vectors are arrays B and C; the result vector is array A.

To generate the bit vector for a nonunit stride loop, use the Q8VMKO function. It generates a pattern of 1 and 0 bits beginning with a 1 bit. The restructured code is shown in figure 10-7.

Loop-Dependent Conditional Forward Transfers

The next two examples illustrate DO loops that are restructured such that the calculation is performed on all sets of operands, but a bit vector is used to determine which calculation results are kept.

For example, consider the following DO loop:

```
      DO 10 I=1,1000
      A(I) = B(I) * C(I)
      IF (A(I) .LE. 0.0) GO TO 10
      R(I) = SQRT(A(I) ** 2 + W(I)) * .05 + D(I)
      S(I) = A(I) * D(I)
   10 CONTINUE
```

The operand vectors are arrays A, B, C, D, and W; the result vectors are arrays R and S.

In the restructured code, the calculation is performed on all 1000 sets of operands. However, by controlling the result element assignment using the Q8VCTRL function, only those results that correspond to 1 bits in the bit vector are stored in the result vectors. The restructured code is shown in figure 10-8.

```
DIMENSION VA(N), VB(N), VC(N)          <- Declares additional arrays required for the vector operation
BIT BITV(N)                               and the bit vector.

BITV(1;N) = Q8VMKO(1,2;BITV(1;N))      <- Generates an alternating 1 and 0 bit pattern.

L = Q8SCNT(BITV(1;N))                  <- Counts the 1 bits in the bit vector.

VC(1;L) = Q8VCMPRS(C(1;N),BITV(1;N);VC(1;L))   <- Creates operand vectors containing only those operands on
VB(1;L) = Q8VCMPRS(B(1;N),BITV(1;N);VB(1;L))      which the calculation is to be performed.

VA(1;L) = VB(1;L) + 2.0 * VC(1;L)      <- Performs the calculation.

A(1;N) = Q8VXPND(VA(1;L),BITV(1;N);A(1;N))   <- Expands the result vector. For each 1 bit in the bit
                                                vector, a calculation result is stored in the A array. For
                                                each 0 bit in the bit vector, a zero value is stored in the
                                                A array.
```

Figure 10-7. Nonunit Stride Example

```
DIMENSION T1(1000), T2(1000)           <- Declares additional arrays required for the vector
BIT BITV(1000)                            operation and the bit vector.

A(1;1000) = B(1;1000) * C(1;1000)
BITV(1;1000) = A(1;1000) .GT. 0.0      <- Generates the bit vector.

T1(1;1000) = W(1;1000) + A(1;1000) ** 2   <- Performs the calculation for the R result vector
T2(1;1000) = VSQRT(T1(1;1000);T2(1;1000))    on all 1000 sets of operands.
T1(1;1000) = T2(1;1000) * .05 + D(1;1000)

R(1;1000) = Q8VCTRL(T1(1;1000),BITV(1;1000);R(1;1000))   <- Stores the results corresponding to 1 bits in the
                                                            bit vector in the R result vector.

T1(1;1000) = A(1;1000) * D(1;1000)     <- Performs the calculation for the S result vector
                                          on all 1000 sets of operands.

S(1;1000) = Q8VCTRL(T1(1;1000),BITV(1;1000);S(1;1000))   <- Stores the results corresponding to 1 bits in the
                                                            bit vector in the S result vector.
```

Figure 10-8. Conditional Vector Store - Example 1

The second example restructures the following DO loop:

```
      DO 10 I=1,N
      IF (TEST(I) .EQ. 0.0) GO TO 10
      R(I) = EXP(A(I))
      S(I) = SQRT(B(I) + C(I) ** 2)
   10 CONTINUE
```

The operand vectors are arrays A, B, and C; the result vectors are arrays R and S.

Again, in the restructured code, the calculation is performed on all N sets of operands. The Q8VCTRL function is then used with a bit vector to determine the results that are kept. The restructured code is shown in figure 10-9.

Summing a Vector

The Q8SSUM function is used to restructure a DO loop that sums a vector. For example, consider the following DO loop:

```
      S = 0
      DO 10 I=1,100
   10 S = S + A(I) * B(I) / C(I) + .05 * D(I)
```

The restructured code is shown in figure 10-10.

Finding the Minimum and Maximum Vector Elements

The Q8SMIN and Q8SMAX functions can be used to find the minimum element and the maximum element, respectively, in a vector. For example, consider the following DO loop:

```
      DO 10 I=1,100
      A(I) = C(I) ** 2 + .05 * F(I)
      B(I) = E(I) ** 2 - D(I)
      ASCA = AMIN1(A(I), ASCA)
      BSCA = AMAX1(B(I), BSCA)
   10 CONTINUE
```

The restructured code is shown in figure 10-11.

Gathering and Scattering

The Q8VGATHR and Q8VSCATR functions enable vectorization of DO loops that use indirect indexing. In a loop that uses indirect indexing, the loop index references an element in an index array whose value is used as the index to the operand array. For example, consider the following DO loops:

```
      DO 10 I=1,100
      TEMPA(I) = A(IA(I))
   10 CONTINUE

      DO 20 I=1,100
      A(IA(I)) = TEMPA(I)
   20 CONTINUE
```

```
DIMENSION RC(N), SC(N)  <-- Declares additional arrays required for the vector operation
BIT BITV(N)                 and the bit vector.

BITV(1;N) = TEST(1;N) .NE. 0.0  <-- A 1 bit is stored for each TEST element not equal to 0.0.

SC(1;N) = B(1;N) + C(1;N) ** 2  <-- Performs calculations on all N sets of operands.
RC(1;N) = VEXP(A(1;N);RC(1;N))
SC(1;N) = VSQRT(SC(1;N);SC(1;N))

R(1;N) = Q8VCTRL(RC(1;N),BITV(1;N);R(1;N))  <-- Stores the results corresponding to 1 bits in the bit vector
S(1;N) = Q8VCTRL(SC(1;N),BITV(1;N);S(1;N))      in the result vectors.
```

Figure 10-9. Conditional Vector Store - Example 2

```
DIMENSION TEMP(100)
TEMP(1;100) = A(1;100) * B(1;100) / C(1;100) + .05 * D(1;100)

S = Q8SSUM(TEMP(1;100))  <-- Sums the elements of vector TEMP.
```

Figure 10-10. Vector Summing Example

```
A(1;100) = C(1;100) ** 2 + .05 * F(1;100)
B(1;100) = E(1;100) ** 2 - D(1;100)

TEMP1 = Q8SMIN(A(1;100))  <-- Finds the minimum element in A.
TEMP2 = Q8SMAX(B(1;100))  <-- Finds the maximum element in B.

ASCA = AMIN1(ASCA,TEMP1)
BSCA = AMAX1(BSCA,TEMP2)
```

Figure 10-11. Minimum and Maximum Element Search Example

The same operations can be performed using the following statements:

```
      TEMPA(1;100) =
     + Q8VGATHR(A(1;100),IA(1;100);TEMPA(1;100))
      A(1;100) =
     + Q8VSCATR(TEMPA(1;100),IA(1;100);A(1;100))
```

In this second example, a temporary vector is created for each indirectly indexed operand array using the Q8VGATHR function. The Q8VSCATR function is used to store the result vector because it is also indirectly indexed. This is the original DO loop:

```
      DO 10 I=1,100
   10 A(IA(I)) =
     + EXP(B(IB(I)) * C(I) + D(ID(I)) ** 2 - .5)
```

The restructured code is shown in figure 10-12.

Locating the Greatest Absolute Value

The following DO loop determines the index of the greatest absolute value in array B:

```
      BMAX = B(1)
      DO 10 I=1,100
      IF (ABS(B(I)) .LE. BMAX) GO TO 10
      IMAX = I
   10 CONTINUE
```

The restructured loop uses the VABS function to store the absolute value of each element in the B vector and the Q8SMAXI function to find the index of the greatest value in the absolute value vector. The restructured code is shown in figure 10-13.

```
      VB(1;100) = VABS(B(1;100); VB(1;100))
      IMAX = Q8SMAXI(VB(1;100))
```

Figure 10-13. Greatest Absolute Value Search Example

Multidimensional Arrays

A vector is a sequence of elements stored in contiguous space. Therefore, a portion of a multidimensional array can be referenced as a vector only if it is contiguous space. Otherwise, it must be compressed to contiguous space using a bit vector as described in the nonunit stride example. Contiguous space is referenced in a columnwise array if the dimensions vary in order beginning with the first dimension. (For a rowwise array, the dimensions must vary from last to first.)

Consider the following DO loop using a columnwise array:

```
      DO 10 I=1,1000
   10 A(I) = SQRT(D(I,5) + 3.0)
```

The loop accesses the fifth column of the two-dimensional I array. This loop can be rewritten as the following statement:

```
A(1;1000) = VSQRT(D(1,5;1000) + 3.0; A(1;1000))
```

```
DIMENSION TEMPA(100), TEMPB(100), TEMPD(100)

TEMPB(1;100) = Q8VGATHR(B(1;100),IB(1;100); TEMPB(1;100))
TEMPD(1;100) = Q8VGATHR(D(1;100),ID(1;100); TEMPD(1;100))

TEMPB(1;100) = TEMPB(1;100) * C(1;100) + TEMPD(1;100) ** 2 - .5
TEMPA(1;100) = VEXP(TEMPB(1;100); TEMPA(1;100))

A(1;100) = Q8VSCATR(TEMPA(1;100),IA(1;100); A(1;100))
```

Figure 10-12. Gathering and Scattering Example

The FORTRAN 200 language includes a number of subroutines that are predefined and can be called from a program. The five categories of predefined subroutines are:

Random number subroutines

Concurrent input/output subroutines

Miscellaneous input/output subroutines

Error processing and debugging subroutines

STACKLIB subroutines

A predefined subroutine is called by placing a subroutine call in a program. Subroutine calls are described in section 7. This section describes each of the predefined subroutines.

RANDOM NUMBER SUBROUTINES

Three predefined subroutines are used in generating random numbers. These subroutines return the current value of the seed used by the random number generator, set the value of the seed used by the random number generator, and generate a vector of random numbers. The random number subroutines are RANGET, RANSET, and VRANF.

RANGET

The RANGET subroutine returns the current value of the seed used by the random number generator. See figure 11-1 for the format of a RANGET call.

```
CALL RANGET(n)

n    An integer variable or array element; n is
     assigned the value of the random number
     generator seed
```

Figure 11-1. RANGET Call Format

RANSET

The RANSET subroutine sets the value of the seed used by the random number generator. See figure 11-2 for the format of a RANSET call.

If the argument is a positive odd integer, the seed is set to the value of the argument. If the argument is a positive even integer, the seed is set to the value of the argument plus 1. If the argument is 0 or negative, or if RANSET is not called, the seed is set to the default value X'000054F4A3B933BD'.

```
CALL RANSET(n)

n    An integer constant, symbolic constant,
     expression, variable, or array element; the
     value of n is used to set the random number
     generator seed
```

Figure 11-2. RANSET Call Format

VRANF

The VRANF subroutine generates a vector of random numbers. See figure 11-3 for the format of a VRANF call.

```
CALL VRANF(v,n)

v    A real array that is to contain the
     generated vector of random numbers

n    An integer that specifies the length of v
```

Figure 11-3. VRANF Call Format

CONCURRENT INPUT/OUTPUT SUBROUTINES

FORTRAN 200 defines four subroutines that perform concurrent input/output operations. A concurrent input/output operation is one that is performed while other statements in the program are being executed. The FORTRAN 200 concurrent input/output subroutines are Q7BUFIN, Q7BUFOUT, Q7WAIT, and Q7SEEK.

Normally, execution of a program is suspended until an input/output operation is completed; it then continues with the next executable statement after the input/output statement.

Concurrent input/output operation allows you to initiate an input/output operation, continue executing other statements in the program simultaneously, and return periodically to check the progress of the input/output operation.

Concurrent input/output operations may be used to transfer data between memory and mass storage.

While a file is being used in a concurrent input/output operation, it cannot be used in any other type of input/output operation; it must first be closed.

You as programmer need to provide for a number of functions when using the concurrent input/output operations. FORTRAN 200 will not automatically supply any needed padding to ensure correspondence between a data record size and the physical block from (or to) which the data are transferred. Furthermore, it will not automatically recognize whether there is a logical end-of-file before the end of the physical block of data that is assigned to the file. The concurrent input/output subroutines recognize only the physical end of the file, not the logical end-of-file. FORTRAN 200 will not check the error conditions that result from the data transfer. You do this by calling the Q7WAIT subroutine. No other notification is made of any input/output error that occurs during a concurrent input/output operation.

Concurrent input/output reduces the execution time of a program by overlapping input/output operations with other computations. The greatest efficiency can be achieved by overlapping all input/output operations with other computations.

ARRAY ALIGNMENT

When using the concurrent input/output subroutines, you must properly align and define arrays. Arrays used by Q7BUFIN and Q7BUFOUT calls must be aligned on a block boundary in memory, and be defined as multiples of blocks, where a block is 512 words of memory. Arrays not spanning full blocks must be padded out to full blocks. The program will abort in execution if the arrays used are not block aligned. Alignment can be accomplished by declaring the arrays to reside in one or more labeled common blocks, then using the GRSP or GRLP parameter of the system utility LOAD to load the common blocks. While both parameters are intended for aligning labeled common blocks to page boundaries, the fact that a system page is always a multiple of 512 blocks ensures that the common blocks will start on a 512-word block boundary.

For example, if we have the following declaration in the program:

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

This example declares two arrays, BIGRAY of size 10240 words, and RA2, of size 51200 words, in labeled common block ANAME, and we compile the program unit, and LOAD the program with the following:

```
LOAD,BINARY,CN=XECUTE,GRSP=*ANAME.
```
or
```
LOAD,BINARY,CN=XECUTE,GRLP=*ANAME.
```

The arrays will be block aligned. (Use of GROS and GROL can also accomplish the same purpose).

Q7BUFIN and Q7BUFOUT use the System Interface Language (SIL) routines Q5READ and Q5WRITE to perform the system I/O functions. The Q5 routines handle such I/O in units of system pages. The system recognizes two types of page definitions: a LARGE page (LP), which is 128 512-word blocks, and a SMALL page (SP), which can be defined at system startup time to be either one, four or sixteen blocks. (See the VSOS reference manual, volume one, for further clarification of the page concept). In a single call, the Q5 routines are capable of handling up to a maximum of twenty-four (24) pages of either type. The type of pages that an array is loaded on (LP or SP) can be controlled by the GRSP or GRLP parameter mentioned above. For more details on the use of those parameters, please refer to the LOAD command in the VSOS reference manual, volume one.

As the intent of concurrent I/O is to have maximum processing and I/O overlap, the Q7BUFxx routines attempt to pass the arrays to the Q5 routines in multiples of twenty-four pages. If the array is larger than 24 pages, the Q7BUFxx routines will take care of making multiple calls until all the data in the array is moved. However, the effect of concurrent I/O is mostly lost. The Q7BUFxx routines will retain control in order to issue the multiple I/O requests. After the last I/O has been initiated, control is returned to the user program. User processing and I/O overlap are therefore limited to the last I/O request which completes the transfer of data to/from the array.

Choosing how the arrays should be loaded obviously has significant impact on how well concurrent I/O can perform. If the array involved is larger than 24 SMALL pages (24 x 512 x installation defined SMALL page size), one should consider loading the array on LARGE pages. However, LARGE page I/O could also cause more data to be moved than necessary. Some judgment and experimentation might be needed for optimal results.

As mentioned above, the Q5 routines handle data transfers in pages. They do accept arrays that are only block aligned, but in those cases where the block alignment does not also match up to a page alignment, the maximum quantity of data transferrable is reduced. For example, if the page size is defined as four blocks, an array spanning blocks 4 to 99 (96 blocks, starting on a SP boundary) can be handled in one Q5 call, while the same array spanning block 3 to 98 of memory will have to be handled in two calls, because the array straddles a page boundary.

The MAP parameter on the Q7BUFxx calls is used to inform those routines how the array involved in this I/O operation is loaded. These routines rely on the MAP parameter to be correctly specified. If the MAP parameter does not match the way the array is actually loaded, unpredictable operations could result. If an array is loaded on SMALL pages and is being processed with a MAP parameter of LARGE (refer to description of the MAP parameter in the next section), a system I/O error status may be returned and the request rejected. If an array loaded on LARGE pages is processed with a MAP parameter of SMALL, many calls will be made to the system, as the Q7BUFxx routines will try to pass only up to twenty-four SMALL pages worth of data to the Q5 routines per call.

SUBROUTINE CALLS

The four concurrent input/output subroutines are:

Q7BUFIN

Transfers data from a file on mass storage to an array in memory

Q7BUFOUT

Transfers data from an array in memory to a file on mass storage

Q7WAIT

Determines if an input/output operation is complete and if any input/output errors occurred during the input/output operation

Q7SEEK

Resets the page address at which data is to be transferred

Two Q7BUFIN calls, two Q7BUFOUT calls, or one Q7BUFIN call and one Q7BUFOUT call can be active at one time for a particular file. If two input/output operations are performed on a file at the same time, you must ensure that the portion of the file affected by one input/output operation does not overlap the portion of the file affected by the other input/output operation. If more than two input/output operations are attempted on a file at the same time, the program is aborted.

File positioning can be accomplished in two ways:

By specifying the relative page address as a parameter in the Q7BUFIN or Q7BUFOUT call

By establishing a relative page address using a Q7SEEK call before a Q7BUFIN or Q7BUFOUT call is executed

If a file is not positioned by using either method, the file is scanned sequentially beginning at the first 512 word block of the file when the file is referenced initially. Thereafter, each Q7BUFIN and Q7BUFOUT call moves the current read/write position forward by a specified amount. This amount is the number of 512 word blocks read or written by the Q7BUFIN or Q7BUFOUT call, and is specified in the argument list of the call.

Each of the concurrent input/output subroutines is described in the following paragraphs.

Q7BUFIN

The Q7BUFIN subroutine transfers data from a mass storage file to an array in memory. The Q7BUFIN subroutine initiates the input operation, and then returns control to the program. See figure 11-4 for the format of a Q7BUFIN call.

The array into which the data is input must not be referenced until the subroutine Q7WAIT is called to determine that the input operation is complete and that no input errors occurred.

```
CALL Q7BUFIN(uid,a,len,map,faddr)

uid     A unit identifier.

a       An array or array element that is
        aligned on a 512 word block boundary.
        Data that is read is stored beginning
        at the address of a.

len     An integer expression that indicates
        the number of 512 word blocks to be
        transferred.

map     A character expression; optional. The
        result of map can have one of the
        following values:

            SMALL   The array is mapped in
                    small pages

            LARGE   The array is mapped in
                    large pages

        The default is SMALL.

faddr   An integer expression whose result
        specifies the position of the unit
        before data is transferred; optional.
        If faddr is specified, map must be
        specified. The default is the current
        position.
```

Figure 11-4. Q7BUFIN Call Format

If improper array alignment forces multiple explicit input requests to be issued, concurrent input processing stops after the initial input request is completed. Thus, program execution is suspended until the input operation is completed.

Depending on the value of len, a Q7BUFIN call might transfer data into only part of the array, or it might transfer data into memory locations beyond the end of the array.

Q7BUFOUT

The Q7BUFOUT subroutine transfers data from an array in memory to a mass storage file. The Q7BUFOUT subroutine initiates the output operation, then returns control to the program. See figure 11-5 for the format of a Q7BUFOUT call.

The array from which the data is output must not be referenced until the subroutine Q7WAIT is called to determine that the output operation is complete and that no output errors occurred.

If improper array alignment forces multiple explicit output requests to be issued, concurrent output processing stops after the initial output request is completed. Thus, program execution is suspended until the output operation is completed.

Depending on the value of len, a Q7BUFOUT call might transfer data from only part of the array, or it might transfer data from memory locations beyond the end of the array.

```
CALL Q7BUFOUT(uid,a,len,map,faddr)
```

uid — A unit identifier.

a — An array or array element that is aligned on a 512 word block boundary. Data that starts at the address of a is written to the external unit.

len — An integer expression that indicates the number of 512 word blocks to be transferred.

map — A character expression; optional. The result of map can have one of the following values:

- SMALL — The array is mapped in small pages
- LARGE — The array is mapped in large pages

The default is SMALL.

faddr — An integer expression whose result specifies the position of the unit before data is transferred; optional. If faddr is specified, map must be specified. The default is the current position.

Figure 11-5. Q7BUFOUT Call Format

Q7WAIT

The Q7WAIT subroutine determines if an input/output operation is complete and if any input/output errors occurred during the input/output operation. The Q7WAIT subroutine must be called after each Q7BUFIN or Q7BUFOUT call. See figure 11-6 for the format of a Q7WAIT call.

Each time the Q7WAIT subroutine is called, it returns a status value that indicates the status of the input/output operation. If the input/output operation is still in progress, the Q7WAIT subroutine can cause control to return to the program or the Q7WAIT routine can cause program execution to be suspended until the input/output operation is completed. The arguments in the Q7WAIT call determine which alternative is used.

The status value returned by the Q7WAIT subroutine also indicates if an input/output error occurred, or if the physical end of the file was reached.

An array that is specified in a Q7BUFIN or Q7BUFOUT call must not be referenced until the Q7WAIT subroutine is called and indicates that the input/output operation is complete and that no input/output errors occurred.

```
CALL Q7WAIT(uid,a,stat,ret,len)
```

uid — A unit identifier.

a — An array or array element that appears in a call to Q7BUFIN or Q7BUFOUT.

stat — An integer variable whose value is returned by the Q7WAIT call. The value returned indicates the status of the input/output operation. The values that can be returned are:

- 0 — The input/output operation was completed normally.
- 1 — The physical end of the file was reached during the input/output operation.
- 2 — Hardware failure caused an input/output error.
- 3 — The input/output operation is not yet completed.
- 4 — The MAP parameter is inconsistent with how the array is grouped on the LOAD statement.

ret — An integer constant or variable that specifies the action to be taken after execution of the call to Q7WAIT is completed; optional. The values that can be specified are:

- 0 — If the specified input/output operation is not yet completed (the value 3 was returned for stat), concurrent input/output processing is to stop until the input/output operation is completed.
- 1 — If the specified input/output operation is not yet completed (the value 3 was returned for stat), concurrent input/output processing is to continue.

The default is 0.

len — An integer variable whose value is returned by the Q7WAIT call; optional. The value returned is the number of 512 word blocks actually transferred during the input/output operation. If the physical end of the file was reached during the input/output operation, the number of 512 word blocks actually transferred might be fewer than the number requested.

If len is specified, ret must be specified.

Figure 11-6. Q7WAIT Call Format

Q7SEEK

The Q7SEEK subroutine repositions the mass storage file by resetting the 512 word block address of the file. See figure 11-7 for the format of a Q7SEEK call.

```
CALL Q7SEEK(uid,faddr)

uid     A unit identifier.

faddr   An integer expression whose result
        positions the unit; optional. If faddr
        is 0 or omitted, the file is repositioned
        to the beginning of the file as if a
        REWIND statement had been executed for
        that unit. Otherwise, faddr has the same
        effect as the faddr parameter of a
        Q7BUFIN or Q7BUFOUT call.
```

Figure 11-7. Q7SEEK Call Format

A file can be rewound by using the statement CALL Q7SEEK(u,0) or CALL Q7SEEK(u), where u is the logical unit number.

Q7STOP

The subroutine Q7STOP enables you to specify the program termination level desired. The subroutine calls Q5TERM; program execution is then halted at the Q7STOP call. A character expression is displayed in the job dayfile or at your terminal. See figure 11-7.1 for the format of the Q7STOP call.

```
CALL Q7STOP(string,irval)

string   Character expression to be displayed
         in the job dayfile or at your
         terminal upon termination.

irval    Integer which indicates the level of
         termination desired.

         irval ≤ 0       :  normal termination
         1 ≤ irval ≤ 4   :  error termination,
                            return code = 4.
         5 ≤ irval ≤ 8   :  fatal termination,
                            return code = 8.
         irval > 8       :  abort termination,
                            return code = 8,
                            program dump
                            performed.
```

Figure 11-7.1. Q7STOP Call Format

MISCELLANEOUS INPUT/OUTPUT SUBROUTINES

Two predefined subroutines perform miscellaneous input/output-related operations. The miscellaneous input/output subroutines are Q8WIDTH and Q8NORED.

Q8WIDTH

The subroutine Q8WIDTH sets a fixed record length for an ASCII output file. The default record length is variable, with trailing blanks removed from each line. See figure 11-8 for the format of a Q8WIDTH call.

```
CALL Q8WIDTH(uid,width)

uid      A unit identifier.

width    An integer that specifies the record
         length for subsequent ASCII output to
         the file. If 0 is specified, trailing
         blanks are removed from each line and
         the record length is variable.
```

Figure 11-8. Q8WIDTH Call Format

Q8NORED

The subroutine Q8NORED is a null operation with VSOS Release 2.2 because of the dynamic file allocation feature in this release. With this feature, when your program creates a file, the system allocates the initial amount of file space. If you have specified a file length, your specification is used; otherwise, the system chooses the initial file length. Thereafter, the system reduces the size of your file only if you request it by calling the Q8REDUCE subroutine.

The system does not automatically reduce the size of your file at the completion of program execution. Because this automatic file size reduction no longer takes place, you do not need to halt it with Q8NORED. A Q8NORED is similar to a null operation; there is no abnormal termination and Q8NORED does not stop any automatic file size reductions.

The dynamic file allocation feature allocates additional space to your file as needed. You can advise the system of the allocation unit size to use for these additions; this will reduce the impact on performance of file fragmentation. The system adjusts this number as needed for the particular device selected and the file growth pattern. If necessary, you can stop the additional allocations by specifying NOEXTEND when you create the file. See appendix F for a description of how Q8NORED works in VSOS Release 2.1.6.

ERROR PROCESSING AND DEBUGGING SUBROUTINES

Several predefined subroutines are used to process execution-time errors and to debug a program. The error processing and debugging subroutines are the data flag branch subroutines Q7DFCL1, Q7DFSET, Q7DFLAGS, and Q7DFOFF, the system error processor subroutine SEP, and the debugging subroutine MDUMP.

DATA FLAG BRANCH MANAGER

The data flag branch manager (DFBM) is a subroutine that processes data flag branches when they occur during execution of a program.

A data flag branch is an automatic transfer of control to the data flag branch manager when a particular condition is detected. A data flag branch is a CYBER 200 hardware function. The hardware monitors a register, which is called the data flag branch register, to determine if a particular condition exists and if that condition is to cause a data flag branch.

Some of the conditions that can cause a data flag branch are:

- An attempt to compute the square root of a negative number
- An attempt to divide a number by 0
- An exponent overflow during the computation of a number too large to represent
- An attempt to use an indefinite value in an operation
- The reduction of the job interval timer to 0 (cannot occur unless the program sets the job interval timer)
- The execution of a hardware breakpoint instruction in some cases (cannot occur unless the program uses the system utility DEBUG or the BKP instruction)

You can specify how a data flag branch is to be processed by using the predefined data flag branch subroutines.

Normally, when the hardware detects a particular condition, it causes control to transfer to DFBM, which processes the data flag branch. DFBM then returns control to the program or aborts the program.

You can provide one or more branch-handling routines to process data flag branches. If you provide a branch-handling routine for a particular condition, DFBM transfers control to the branch-handling routine. Your branch-handling routine can process the condition and return control to the program.

The following paragraphs describe the data flag branch register, data flag branch processing, and the data flag branch subroutines that you can use to specify how data flag branches are to be processed.

Data Flag Branch Register

The data flag branch register is a 64-bit register that is located in the CYBER 200 central processor. The data flag branch register consists of data flags, which indicate if a particular condition exists, condition-enable bits, which specify the conditions that can cause a data flag branch, and product bits, which indicate if an enabled condition exists. Several other bits are also contained in the data flag branch register. See figure 11-9 for an illustration of the data flag branch register.

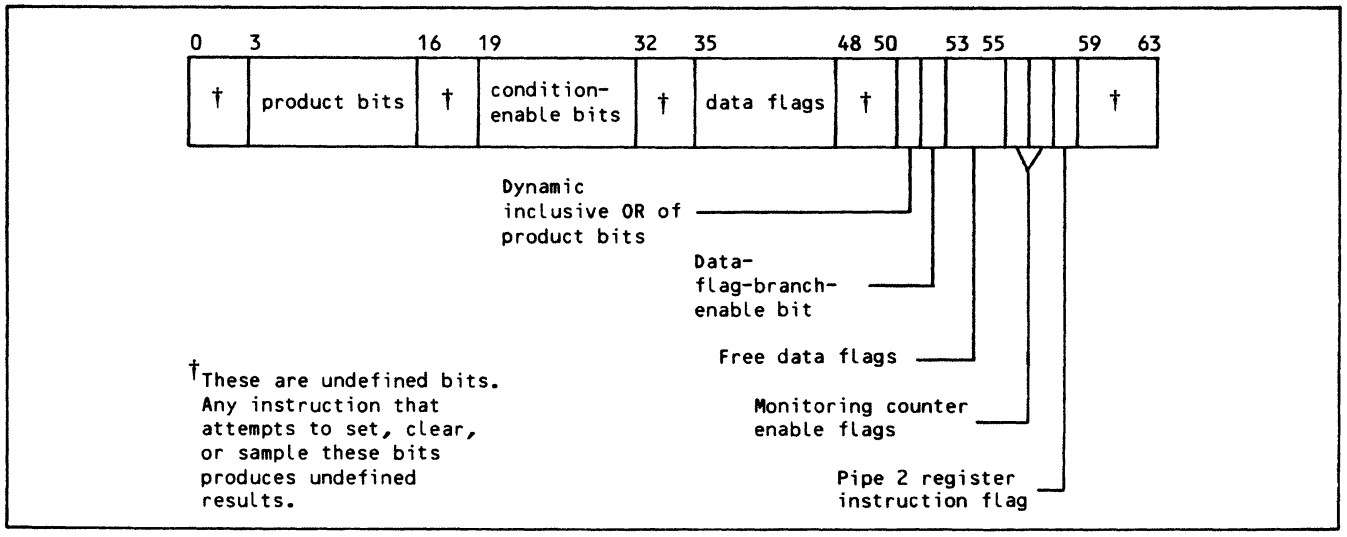


Figure 11-9. Data Flag Branch Register Format

The data flags are bits 35 through 47 of the data flag branch register. Each data flag corresponds to a particular condition. If a condition occurs during execution of a program, the corresponding data flag is set to 1 by the hardware.

For example, if an attempt is made to divide a number by 0, bit 41 is set to 1 by the hardware. Once a data flag is set to 1 by the hardware, it remains set to 1 until the program or DFBM sets it to 0.

The condition-enable bits are bits 19 through 31 of the data flag branch register. Each condition-enable bit corresponds to a bit in the data flag field: bit 19 corresponds to bit 35, bit 20 corresponds to bit 36, and so on. Each condition-enable bit specifies whether the condition to which it corresponds can cause a data flag branch. If a condition-enable bit is set to 1, a data flag branch can occur when that condition exists; if a condition-enable bit is set to 0, a data flag branch cannot occur when that condition exists.

For example, in order for a data flag branch to occur when an attempt is made to divide a number by 0, bit 25 must be set to 1.

You can set the condition-enable bits for some conditions to 1 or 0 by using data flag branch subroutine calls. Some condition-enable bits are initialized by the system at the beginning of program execution.

The product bits are bits 3 through 15 of the data flag branch register. Each product bit corresponds to a bit in the data flag field: bit 3 corresponds to bit 35, bit 4 corresponds to bit 36, and so on. Each product bit specifies whether the condition to which it corresponds exists and is enabled. A product bit is the logical product (logical .AND.) of the corresponding data flag and the corresponding condition-enable bit. If a product bit is set to 1, the condition exists and is enabled. If a product bit is set to 0, the condition does not exist or it is not enabled.

For example, if bit 9 is set to 1, then an attempt to divide a number by 0 occurred and that condition is enabled.

Product bits are set to 1 or 0 by the system during program execution. You can test the bits by using the Q8BADF special call. See section 12 for a description of the Q8BADF special call. A product bit remains set to 1 until the corresponding condition-enable bit is set to 0, or until the corresponding data flag is set to 0.

Bit 51 is the logical sum (logical inclusive .OR.) of all of the product bits. Bit 51 is set to 1 by the hardware when at least one of the product bits is set to 1. Bit 51 remains set to 1 until all of the product bits are set to 0.

Bit 52 is the data-flag-branch-enable bit. Data flag branches can only occur when bit 52 is set to 1. Bit 52 is set to 1 or 0 by the hardware and by DFBM. Bit 52 is set to 0 when a data flag branch is initiated.

When both bit 51 and bit 52 are set to 1, a data flag branch occurs.

Bit 58 is the pipe 2 register instruction data flag. Bit 58 is set to 1 by the hardware when a pipe 2 register instruction sets one of the data flags to 1. Once bit 58 is set to 1, it remains set to 1 until the program or DFBM sets it to 0. See the appropriate hardware reference manual for a description of pipe 2 register instructions.

See table 11-1 for the condition that corresponds to each of the data flags in the data flag branch register. The table also shows the condition-enable bit and the product bit that corresponds to each data flag, the classification of each condition, and the priority of each condition.

The following paragraphs describe the classification of conditions, the default conditions, and the hardware processing of data flag branches.

Condition Classifications

There are two classes of conditions that can cause data flag branches: class I conditions and class III conditions. Class I conditions are conditions that are always enabled. (The condition-enable bits for class I conditions are always set to 1.) The class I conditions are JIT, SFT, and BKP.

A class I condition cannot exist unless the program performs a specific action to cause the condition. For example, a class I condition could be caused by using the breakpoint feature of the system utility DEBUG, or by using the special call Q8WJTIME to set the job interval timer.

TABLE 11-1. DATA FLAG BRANCH CONDITIONS

Class	Designator	Condition Description	Condition-Enable Bit	Data Flag	Product Bit	Priority
I	SFT	(Reserved)	19†	35	3	2
I	JIT	Job interval timer has reduced to zero.	20†	36	4	1
III	SSC	Selected condition has not been met. In search for masked key, there was no match; or count of nonzero translated bytes is greater than 65535.	21	37	5	11
III	DDF	Decimal data fault. A sign was found in a digit position, or vice versa.	22	38	6	12
III	TBZ	Truncation of leading nonzero digits or bits, or decimal or binary division by zero.	23	39	7	13
III	ORD	Dynamic inclusive OR of the preceding three conditions (SSC, DDF, and TBZ). Enabling this condition permits a branch on any of the three conditions.	24	40	8	5
III	FDV	Floating-point divide fault.	25†	41	9	8
III	EXO	Exponent overflow.	26	42	10	9
III	RMZ	Result is machine zero.	27	43	11	10
III	ORX	Dynamic inclusive OR of the preceding three conditions (FDV, EXO, and RMZ). Enabling this condition permits a branch on any of the three conditions.	28	44	12	4
III	SRT	Square root operation on negative operand.	29†	45	13	6
III	IND	Indefinite result or indefinite operand.	30†	46	14	7
I	BKP	Breakpoint flag was set on the breakpoint instruction (instruction #04).	31†	47	15	3

†Set during execution-time initialization.

Class III conditions are conditions that can be enabled or disabled. You can enable and disable a class III condition by using the predefined subroutines Q7DFSET and Q7DFOFF respectively. These subroutines are described later in this section. The class III conditions are ORX, ORD, SRT, IND, FDV, EXO, RMZ, SSC, DDF, and TBZ.

An ORX condition exists if an FDV, EXO, or RMZ condition exists.

An ORD condition exists if an SSC, DDF, or TBZ condition exists.

An SRT condition exists if the program attempts to compute the square root of a negative number. When this condition occurs, the two's complement of the square root of the absolute value of the number is computed. This result is meaningful, but not mathematically correct.

An IND condition exists if an indefinite value is computed and stored into memory or the register file. The condition also occurs if one or more of the operands of a floating-point operation have indefinite values. Both floating-point arithmetic operations and floating-point comparisons can cause the IND condition. Because an indefinite value results from a floating-point operation involving one or more indefinite operands, indefinite values can be propagated easily. The FDV condition and the EXO condition can also cause an IND condition.

An FDV condition exists if a floating-point division operation is attempted with a divisor of 0. A divisor of 0 is either a machine-zero or a floating-point number that has an all-zero coefficient. A divisor that has an indefinite value is not a 0 divisor and does not cause the FDV condition. The result of a division by 0 is an indefinite value, which causes the IND condition. See section 2 for a description of the representations for machine-zero values and indefinite values.

An EXO condition exists if the exponent of a number is too large to be represented. The EXO condition can also cause an IND condition.

An RMZ condition exists if the result of an operation is a machine-zero. See section 2 for a description of the representation of machine-zero.

An SSC condition exists if a selected condition is not satisfied.

A DDF condition exists if a decimal data fault occurred. A decimal data fault occurs when the sign of a number appears in a position that should contain a digit, or vice versa.

A TBZ condition exists if the leading nonzero bits or digits of a number are truncated, or if a decimal or binary division operation is attempted with a divisor of 0.

DFBM processes class I conditions individually, as if each class I condition were caused by a separate event. DFBM processes class III conditions as a group, as if they were all caused by a single event.

A data flag branch that is caused by a class I condition is called a class I branch. A data flag branch that is caused by a class III condition is called a class III branch.

Default Conditions

At the time program execution begins, six condition-enable bits are automatically set in the data flag branch register. This enables a data flag branch to occur if any of these conditions exists during program execution. The conditions that are initially enabled are JIT, SFT, BKP, IND, SRT, and FDV.

Hardware Processing of Data Flag Branches

When an enabled condition occurs during program execution, the following steps are performed by the hardware:

Bit 52 is set to 0. This disables all further data flag branches.

The address of the instruction that would have been executed next is placed in register #1.

Control transfers to the address that is stored in register #2, which is the address of a DFBM entry point. (This address is placed in register #2 at the time program execution begins.)

The processing performed by DFBM depends on the bit settings in the data flag branch register and by any specifications that were made by calls to the predefined subroutines Q7DFCL1, Q7DFSET, and Q7DFOFF.

The address that is stored in register #1 is not necessarily the address of the instruction immediately following the instruction that caused the data flag branch. The hardware initiates a data flag branch only after all currently executing instructions have finished executing. Because instructions might be executing in parallel when the condition that causes a data flag branch exists, the data flag branch can occur up to 35 instructions after the instruction that caused it. Also, the point at which control transfers to DFBM can differ between executions of the same program because the load and store hardware operations can occur at different points as a result of the asynchronous nature of CYBER 200 input/output.

Use of the special calls Q8BADF or Q8LSDFR, or the system utility DEBUG in a program that calls DFBM entry points can effect changes in the data flag branch register that conflict with DFBM. Therefore, you should use caution when using the special calls Q8BADF and Q8LSDFR, and the system utility DEBUG.

Data Flag Branch Processing

When a data flag branch occurs, the hardware transfers control to DFBM. DFBM then checks the product bits of the data flag branch register to determine what condition caused the data flag branch. DFBM checks the product bits beginning with the product bit that corresponds to the highest priority condition; therefore DFBM checks the product bits in the following order:

1. Bit 4 JIT condition
2. Bit 3 SFT condition
3. Bit 15 BKP condition

4. Bit 12 ORX condition

5. Bit 8 ORD condition

6. Bit 13 SRT condition

7. Bit 14 IND condition

8. Bit 9 FDV condition

9. Bit 10 EXO condition

10. Bit 11 RMZ condition

11. Bit 5 SSC condition

12. Bit 6 DOF condition

13. Bit 7 TBZ condition

DFBM then performs one of the following:

If no branch-handling routine is specified for the condition, default processing is performed.

If a branch-handling routine is specified for the condition and if the condition is a class I condition, class I branch processing is performed using the branch-handling routine.

If a branch-handling routine is specified for the condition and if the condition is a class III condition, class III branch processing is performed using the branch-handling routine.

Default Branch Processing

In the absence of a user-written branch-handling routine to process the class I or class III branch, the hardware initiates a data flag branch which DFBM processes. DFBM performs the following processing steps:

Having checked the data flag branch register to determine which condition exists and has the highest priority, DFBM transfers control to a predefined routine that issues an error message for that condition if appropriate.

If the program specified an error exit subroutine by calling the system error processing predefined subroutine SEP before the data flag branch occurred, control is transferred to that error exit subroutine. The predefined subroutine SEP is described later in this section.

If an error exit was not previously specified, and if the error that occurred was nonfatal, DFBM sets one or all of the data flags to 0 and returns control to the program. If a class I condition caused the data flag branch, the data flag corresponding to the highest priority existing condition is set to 0. If a class III condition caused the data flag branch, all data flags are set to 0. Setting a data flag to 0 automatically sets the corresponding product bit to 0 as well.

If an error exit was not previously specified, and if the error that occurred was fatal or catastrophic, DFBM outputs the contents of the data flag branch register and aborts the program. If the program is part of a batch job, a post-mortem dump is also output.

Class I Branch Processing Using Branch-Handling Routine

When a class I condition is detected, the hardware initiates a data flag branch and DFBM processes the data flag branch. If you do not provide a branch-handling routine to process the class I branch, DFBM uses default data flag branch processing to process the data flag branch.

If you provide a branch-handling routine for the class I branch, DFBM performs the following steps:

Having checked the data flag branch register to determine which condition exists and has the highest priority, DFBM sets the data flag for that condition to 0. This automatically sets the corresponding product bit to 0 as well.

DFBM transfers control to the branch-handling routine that was specified in the most recent call to Q7DFCL1 for the particular condition.

In order to process a class I branch using a branch-handling routine, you must supply one or more branch-handling routines and you must call the predefined subroutine Q7DFCL1 in order to specify the address of the branch-handling routine to be used when a particular condition causes a class I branch.

A class I branch-handling routine must be written in a lower-level language, such as CYBER 200 assembly language. A class I branch-handling routine cannot be written in FORTRAN. See the CYBER 200 Assembler reference manual for a description of the CYBER 200 assembly language.

A class I branch-handling routine is responsible for most of the interface between itself and DFBM. DFBM does not use a standard calling sequence to call the branch-handling routine. Instead, DFBM transfers control to the address specified in the most recent call to Q7DFCL1 for the particular condition. Therefore, the address of the data base of the class I branch-handling routine is not available in register #1E.

The branch-handling routine must save the values in register #1 through #FF, and the branch-handling routine must restore the values of those registers before returning control to DFBM.

The address to which the class I branch-handling routine must transfer control is returned in a parameter of the most recent call to Q7DFCL1 for the particular condition. At the time control branches to the class I branch-handling routine, all data flag branches are disabled.

The address of the branch-handling routine used for class I data flag branches must be specified in a call to the predefined subroutine Q7DFCL1 before a class I data flag branch occurs. At least one Q7DFCL1 call must be made for each class I condition processed by a branch-handling routine. The specification of a class I branch-handling routine is effective for the duration of program execution, or until another call to Q7DFCL1 is executed for a particular condition. The branch-handling routine used for class I data flag branches can be changed during execution of the program by using multiple calls to Q7DFCL1. The predefined subroutine Q7DFCL1 is described later in this section.

Class III Branch Processing Using Branch-Handling Routine

When a class III condition is detected, the hardware initiates a data flag branch and DFBM processes the data flag branch. If you do not provide a branch-handling routine to process the class III branch, DFBM uses default data flag branch processing to process the data flag branch.

If you provide a branch-handling routine for the class III branch, DFBM performs the following steps:

Having checked the data flag branch register to determine which condition exists and has the highest priority, DFBM saves a copy of the entire register file of the routine in which the data flag branch was initiated.

DFBM sets all of the data flags in the data flag branch register to 0. This automatically sets all of the product bits to 0 as well.

DFBM sets bit 52 of the data flag branch register to 1, which enables further data flag branches.

DFBM transfers control to the branch-handling routine specified in the call to Q7DFSET that was in effect at the time the data flag branch occurred.

In order to process a class III branch using a branch-handling routine, you must supply one or more branch-handling routines, and you must call the predefined subroutine Q7DFSET in order to specify the name of the branch-handling routine to be used when a particular condition causes a class III branch.

A class III branch-handling routine can be a FORTRAN subroutine, but it must have no arguments. Data communication between the branch-handling routine and higher level routines can be accomplished by using common blocks.

If a class III branch occurs while the branch-handling routine is executing, DFBM causes a catastrophic error message to be issued and aborts the program. The branch-handling routine can disable class III branches while the branch-handling routine is executing; this is done by calling the predefined subroutine Q7DFSET from the branch-handling routine.

If a class I branch occurs while a branch-handling routine is executing, the class I branch is processed immediately.

All data flags are set to 0 before control is transferred from DFBM to a branch-handling routine; however, the branch-handling routine can determine which data flags were set to 1 at the time of the data flag branch by calling the predefined subroutines, Q7DFLAGS or Q7DFBR.

The name of the branch-handling routine used for class III data flag branches must be specified in a call to the predefined subroutine Q7DFSET before a class III data flag branch occurs. Each subroutine in a program can make different specifications of how class III branches are to be handled within that subroutine and in subroutines called by that subroutine. These specifications have no effect on the specifications that were made in higher-level subroutines.

See figure 11-10 for an illustration of the scope of calls to Q7DFSET. The main program in the example begins execution with the default conditions in effect and executes until the first call to Q7DFSET is executed. A new set of conditions is selected by the second Q7DFSET call and remains in effect until the Q7DFSET call in subroutine K is executed. The set of conditions selected by the Q7DFSET call that appears in subroutine K remains in effect throughout execution of subroutine K, which includes execution of subroutine D.

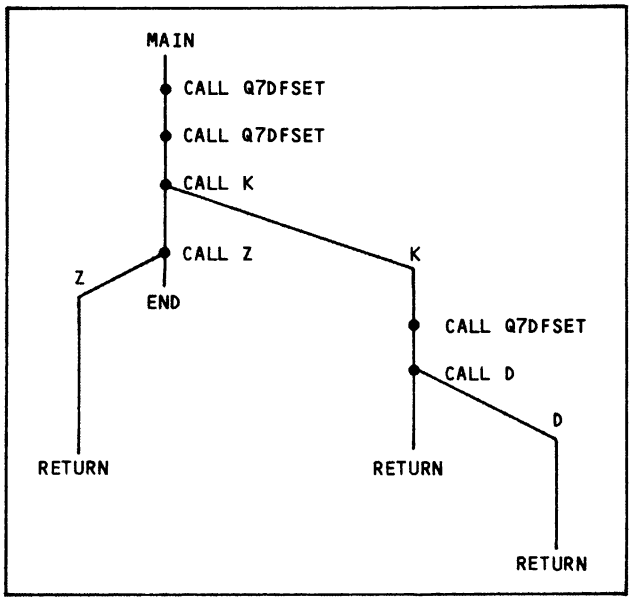


Figure 11-10. Scope of Selected Conditions

When execution of subroutine K is completed and control returns to the main program, the conditions that were in effect when subroutine K was called are reestablished. This set of conditions remains in effect throughout execution of the remainder of the program, which includes execution of subroutine Z.

Multiple Conditions Per Branch

The execution of a single machine language instruction can cause several class I and class III conditions to be flagged at the same time. When several product bits are set to 1 in the data flag branch register, the hardware causes control to transfer to DFBM as described previously. DFBM then processes each class I condition and one class III condition before returning control to the program.

If a data flag branch occurs and more than one product bit is set to 1, DFBM processes any class I conditions first according to the priority of the condition. Then, if processing the class I condition does not cause the program to abort, DFBM processes the highest priority class III condition that exists.

See table 11-2 for the processing that would be performed for a data flag branch involving multiple conditions. This table assumes that any branch-handling routines return control to the program.

TABLE 11-2. MULTIPLE INTERRUPT PROCESSING

Class I Branch-Handling Routine Provided	Class III Branch-Handling Routine Provided	Processing Performed After Control Transfers to DFBM
No	No	Class I error message issued, program aborted
Yes	No	Class I routine executed, class III error message issued, program aborted for fatal message and resumed otherwise
No	Yes	Class I error message issued, program aborted (class III routine not executed although class III condition flagged)
Yes	Yes	Class I routine executed, class III routine then executed, program resumed (no error messages issued by DFBM)

Data Flag Branch Subroutines

Several predefined subroutines enable you to control processing performed by DFBM. These subroutines specify class I and class III branch-handling routines, reference the bit settings of the data flag branch register, and enable/disable class III data flag branches. The data flag branch subroutines are Q7DFCL1, Q7DFSET, Q7DFBR, Q7DFLAGS, and Q7DFOFF.

Q7DFCL1

The Q7DFCL1 subroutine specifies the address of a branch-handling routine to which control transfers when a class I data flag branch occurs. The Q7DFCL1 subroutine also returns the address to which the branch-handling routine must return upon completion. See figure 11-11 for the format of a Q7DFCL1 call.

At least one Q7DFCL1 call must be made for each class I condition that you want to process with a branch-handling routine. More than one Q7DFCL1 call can be made for a particular condition; the most recently executed Q7DFCL1 call for a particular condition is the call that is effective.

Q7DFSET

The Q7DFSET subroutine can be used to do either or both of the following:

- Specify the conditions for which a class III data flag branch is to occur by setting the appropriate condition-enable bits in the data flag branch register
- Specify the name of the branch-handling routine that is to be called when a data flag branch occurs for a particular class III condition

See figure 11-12 for the format of a Q7DFSET call.

More than one Q7DFSET call can be made for a particular condition; the Q7DFSET call that is effective at a particular time depends on the scope of the Q7DFSET calls. The scope of a Q7DFSET call is described previously in this section.

```
CALL Q7DFCL1(bhr,return,'cd')
```

bhr	A fullword variable that contains the virtual bit address of the branch-handling routine to which DFBM is to transfer control if the specified condition occurs.
return	A fullword variable. The Q7DFCL1 call assigns to return the virtual bit address in DFBM to which the branch-handling routine for the specified condition is to transfer control upon completion.
cd	A class I condition designator.

Figure 11-11. Q7DFCL1 Call Format

```
CALL Q7DFSET(bhr)
    or
CALL Q7DFSET(bhr,'NUL')
    or
CALL Q7DFSET(bhr,'cd1', ... ,'cdn')
```

bhr	A zero or the name of the branch-handling routine that is to be called if a class III branch occurs. If a zero is specified, default processing is performed for class III branches (thus, the specification in effect at the time program execution began is reestablished).
'NUL'	Indicates that all class III condition-enable bits are to be set to zero, which disables all class III branches.
cd_i	A class III condition designator or the characters STD. If condition designators are specified, the corresponding condition-enable bits are set to 1. If STD is specified, the condition-enable bits corresponding to the conditions SRT, IND, and FDV are set to 1. Condition designators and STD can be specified in the same Q7DFSET call.

Figure 11-12. Q7DFSET Call Format

Q7DFBR

The Q7DFBR subroutine returns a copy of the data flag branch register contents. If you call Q7DFBR before any interrupts, class I or class III, have occurred, the result is undefined.

If program execution causes an error condition that leads to a data flag condition: The bit in the data flag branch register corresponding to that error condition is set to 1. Otherwise, the bit stays at 0. See table 11-1 for the data flag branch conditions.

See figure 11-13 for the format of a Q7DFBR call.

```
CALL Q7DFBR(ivar)

ivar   An integer variable that gets a copy of
       the data flag branch register contents.
```

Figure 11-13. Q7DFBR Call Format

Q7DFLAGS

The subroutine Q7DFLAGS references the bit settings in the data flag branch register. See figure 11-14 for the format of a Q7DFLAGS call.

The subroutine Q7DFLAGS returns an array of logical values that indicate whether each product bit in the data flag branch register was set to 1 or to 0 at the time the data flag branch occurred. The logical value .TRUE. indicates that the corresponding bit was set to 1; the logical value .FALSE. indicates that the corresponding bit was set to 0.

If Q7DFLAGS is called before any class III branches have occurred, all of the logical values returned are .FALSE. and all other values returned are 0.

Q7DFOFF

The subroutine Q7DFOFF disables specified class III branches at the time control returns to the program from a branch-handling routine. See figure 11-15 for the format of a Q7DFOFF call.

A Q7DFOFF call can appear only in a branch-handling routine; if the call appears in any other routine, the call has no effect.

Class III branches that are disabled by a Q7DFOFF call become disabled only when control returns to the program from the branch-handling routine in which the Q7DFOFF call appears. The conditions that are disabled by a Q7DFOFF call remain disabled until a Q7DFSET call is executed.

The scope of a Q7DFOFF call is the same as the scope of its associated Q7DFSET call. The scope of a Q7DFSET call is described previously in this section.

```
CALL Q7DFLAGS(pb,df,ad,rf)

pb   A 1-dimensional 10-element array of type
     logical; the Q7DFLAGS call assigns logical
     values to the array elements that indicate
     the setting of the class III product
     bits: the value .TRUE. corresponds to the
     bit value 1, and the value .FALSE.
     corresponds to the bit value 0. The first
     element of array pb corresponds to the
     class III condition having the highest
     priority, the second element of array pb
     corresponds to the class III condition
     having the second-highest priority, and so
     on. See table 11-1 for the priorities of
     the conditions.

df   A 1-dimensional 11-element array of type
     logical; the Q7DFLAGS call assigns logical
     values to the array elements that indicate
     the settings of the class III data flags
     and the pipe 2 register instruction flag:
     the value .TRUE. corresponds to the bit
     value 1, and the value .FALSE. corresponds
     to the bit value 0. The first element of
     array df corresponds to the class III
     condition having the highest priority, the
     second element of array df corresponds to
     the class III condition having the
     second-highest priority, and so on; the
     11th element of array df corresponds to
     the pipe 2 register instruction data
     flag. See table 11-1 for the priorities
     of the conditions.

ad   A variable of type integer; the Q7DFLAGS
     call assigns to ad the address contained
     in register #1, which is the address of
     the instruction that would have been
     executed next had the data flag branch not
     occurred.

rf   A 1-dimensional 256-element array or
     symbolic descriptor array of type integer
     or real; optional. The Q7DFLAGS call
     assigns to the elements of rf the contents
     of the register file at the time the data
     flag branch occurred.
```

Figure 11-14. Q7DFLAGS Call Format

```
CALL Q7DFOFF('cd1, ... ,'cdn)

cdi   A class III condition designator,
      or the characters ALL or STD. If
      condition designators are specified,
      the corresponding condition-enable
      bits are set to 0. If ALL is speci-
      fied, all condition-enable bits are
      set to 0. If STD is specified, the
      condition-enable bits corresponding
      to the conditions SRT, IND, and FDV
      are set to 0.
```

Figure 11-15. Q7DFOFF Call Format

SYSTEM ERROR PROCESSOR

The system error processor (SEP) is a subroutine that changes certain attributes of execution-time errors. Some of the attributes of an error that can be changed are the severity of the error, the number of nonfatal errors that can occur during program execution, the printing of the error message, and the content of the error message. See figure 11-16 for the format of a SEP call.

Parameter p_1 and at least one additional parameter must be specified in a SEP call. A parameter must be indicated as 0 if that parameter is not to be specified; however, trailing zero parameters can be omitted.

SEP calls can appear as frequently as required in a program, and the error attributes can be changed any number of times during program execution.

You should use caution when changing the severity of an error from fatal to nonfatal.

CALL SEP(p_1,p_2,p_3,p_4,p_5,p_6,p_7)

p_1 The error number of the run-time error. See appendix B. When p_1 is zero, all other parameters must be zero except p_4, which refers to the global nonfatal error count.

p_2 Indicates the error severity to which p_1 is to be changed. Parameter p_2 can be one of the following:

- 'F' Sets the error severity to fatal. Program execution is terminated abnormally when this condition occurs.
- 'W' Sets the error severity to warning. Execution continues when this nonfatal condition occurs.
- 0 The error severity is not changed.

When a fatal error is changed to a warning error, parameter p_4 should also be specified to change the maximum error count to a nonzero number.

p_3 The error exit subroutine entry point name, which must be included in an EXTERNAL statement in the same program unit. If the error p_1 occurs, entry point p_3 is called and execution continues from there. If p_3 is zero, no error exit is implied and processing continues if the error is nonfatal. If p_1 is a fatal error and the subroutine p_3 executes a RETURN, the program aborts; if p_1 is nonfatal and p_3 executes a RETURN, program execution continues.

p_4 An integer constant indicating the maximum error count for nonfatal errors; if the number of nonfatal error condition occurrences reaches p_4, execution terminates. An infinite error count is indicated by a value of -1. If p_4 is zero, no change for this parameter is indicated (p_4 might have been assigned a value in a previous SEP call). If p_4 is greater than 255, SEP sets p_4 to 255.

The maximum error count for a warning error for which SEP has not been called is 25. The maximum error count for a fatal error for which SEP has not been called is zero. When p_2 changes a fatal error to a warning error, p_4 should also be specified.

p_5 The error display supression argument, applying only to nonfatal errors. The values that can be specified for p_5 are:

- 'S' Indicates that the error message, normally sent to the user's dayfile or to the terminal, is to be supressed.
- 0 No message suppression is to take place.

p_6 The number of characters in p_7, excluding bracketing apostrophes.

p_7 A character string that replaces the standard message associated with p_1. The string must be enclosed by apostrophes to form a character constant. Parameter p_6 must appear when p_7 appears.

Figure 11-16. SEP Call Format

MDUMP

The subroutine MDUMP is a debugging subroutine that outputs the contents of specified areas of virtual memory. See figure 11-17 for the format of an MDUMP call.

```
CALL MDUMP(first,len,dtype,uid)

first    A simple variable, array, or array
         element with which the area to be
         dumped begins.

len      Length in fullwords of area to be
         dumped.

dtype    Dump format:

         'Z'        Hexadecimal dump

         'I'        Integer dump

         'Ew.d'     Floating-point dump, where w
           or       is the field width and d is
         'Fw.d'     the factional decimal digit
                    count

         If dtype has a value other than one of
         the above, a hexadecimal dump is made.

uid      An external unit identifier.
```

Figure 11-17. MDUMP Call Format

The subroutine MDUMP can be called from a FORTRAN program or from assembly language subroutines that are called by a FORTRAN program. MDUMP is called from an assembly language subroutine by using the standard calling sequence conventions. See section 13 for a description of the standard calling sequence conventions. The external unit identifier specified in an MDUMP call that appears in an assembly language subroutine must be defined in the same way as for MDUMP calls that appear in a FORTRAN program unit.

STACKLIB SUBROUTINES

The STACKLIB subroutines are a library of predefined subroutines designed to optimize certain DO loop constructs that do not lend themselves to direct vector calculations. The STACKLIB subroutines optimize these constructs by efficient use of the instruction stack and register file.

If vectorization is requested with the OPTIMIZE=V compilation option, the compiler attempts to transform each DO loop it cannot vectorize into a STACKLIB subroutine call as described under Generation of Calls to STACKLIB Routines in section 9. The compiler may not recognize a DO loop as vectorizable or as convertible to a STACKLIB routine. In that case, you can optimize your program yourself by replacing the unconverted DO loops with the appropriate calls to STACKLIB routines.

NOTE

STACKLIB calls only provide scalar optimization. Whenever possible, restructure DO loop constructs so that they execute as vector operations.

Each STACKLIB call replaces a DO loop. Table 11-3 defines each available STACKLIB subroutine with an example of a DO loop that is the equivalent of the corresponding STACKLIB call.

STACKLIB SUBROUTINE CHARACTERISTICS

The STACKLIB subroutines in table 11-3 are defined in terms of the following characteristics:

Forward count: Indicates that vector elements are processed in increasing subscript order.

Backward count: Indicates that vector elements are processed in decreasing subscript order.

Dyadic operation: Indicates that the equivalent calculation is a DO loop comprising one arithmetic operation performed on two operands.

Triadic operation: Indicates that the equivalent calculation is a DO loop comprising two arithmetic operations performed on three operands.

Broadcasts: Indicates that the subroutine uses one or more scalar (loop invariant) operands. All other operands are vector (loop dependent) operands.

Recursive: Indicates that an element of the result vector is used to compute the next element of the result vector. A call to a recursive STACKLIB routine specifies the same vector as the result vector and as one of the operand vectors.

Normal order: Indicates a triadic operation in which the first arithmetic operation (using the first and second operands) is performed first and the second arithmetic operation (using the third operand and the operation result) is performed second. For example, using normal order, the expression A + B * C is computed as (A + B) * C.

Reverse order: Indicates a triadic operation in which the second arithmetic operation (using the second and third operands) is performed first and the first arithmetic operation (using the first operand and the operation result) is performed second. For example, using reverse order, the expression A + B * C is computed A + (B * C).

TABLE 11-3. STACKLIB ROUTINES

Recursive Add, Normal Count, Forward Order (Q8A0x0)		
Name:	Q8A010	Q8A020
Recursive Operand:	v1	v2
Example:	CALL Q8A010(R(2),A(2),R(1),L-1)	CALL Q8A020(R(2),R(1),B(2),L-1)
DO Loop Equivalent of STACKLIB Call:	DO 10 I=2,L 10 R(I) = A(I) + R(I-1)	DO 10 I=2,L 10 R(I) = R(I-1) + B(I)

Recursive Multiply Add, Normal Count, Forward Order (Q8MA0x0)		
Name:	Q8MA020	Q8MA040
Recursive Operand:	v2	v4
Example:	CALL Q8MA020(R(2),A(1),R(1),C(2),L-1)	CALL Q8MA040(R(2),R(1),B(2),C(2),L-1)
DO Loop Equivalent of STACKLIB Call:	DO 10 I=2,L 10 R(I) =(A(I-1) * R(I-1)) + C(I)	DO 10 I=2,L 10 R(I) =(R(I-1) * B(I))+ C(I)

Recursive Multiply Add, Normal Count, Reverse Order (Q8AM0x1)		
Name:	Q8AM011	Q8AM021
Recursive Operand:	v1	v2
Example:	CALL Q8AM011(R(2),A(2),B(1),R(1),L-1)	CALL Q8AM021(R(2),A(2),R(1),C(1),L-1)
DO Loop Equivalent of STACKLIB Call:	DO 10 I=2,L 10 R(I) = A(I) + (B(I-1) * R(I-1))	DO 10 I=2,L 10 R(I) = A(I) + (R(I-1) * C(I-1))

Recursive Multiply Add, Backwards Count, Reverse Order (Q8AM0x3)		
Name:	Q8AM013	Q8AM023
Recursive Operand:	v1	v2
Example:	CALL Q8AM013(R(L-1),A(L-1),B(L-1),R(L),L-1)	CALL Q8AM023(R(L-1),A(L-1),R(L),C(L-1),L-1)
DO Loop Equivalent of STACKLIB Call:	DO 10 I=2,L J = (L+1) - I 10 R(J) = A(J) + (B(J) * R(J+1))	DO 10 I=2,L J = (L+1) - I 10 R(J) = A(J) + (R(J+1) * C(J))

TABLE 11-3. STACKLIB ROUTINES (Contd)

Recursive Multiply Add, Broadcast, Backwards Count, Forward Order (Q8MAx12)		
Name:	Q8MA212	Q8MA412
Recursive Operand:	v1	v1
Broadcast Operand:	v2	v4
Example:	CALL Q8MA212(R(L-1),A(L-1),S,R(L),L-1)	CALL Q8MA412(R(L-1),S,B(L-1),R(L),L-1)
DO Loop Equivalent of STACKLIB Call:	DO 10 I=2,L J = (L+1) - I 10 R(J) =(A(J) * S)+ R(J+1)	DO 10 I=2,L J = (L+1) - I 10 R(J) =(S * B(J)) + R(J+1)

Recursive Multiply Add, Broadcast, Backwards Count, Reverse Order (Q8AMx43)		
Name:	Q8AM143	Q8AM243
Recursive Operand:	v4	v4
Broadcast Operand:	v1	v2
Example:	CALL Q8AM143(R(L-1),R(L),B(L-1),S,L-1)	CALL Q8AM243(R(L-1),R(L),S,C(L-1),L-1)
DO Loop Equivalent of STACKLIB Call:	DO 10 I=2,L J = (L+1) - I 10 R(J) = R(J+1) + (B(J) * S)	DO 10 I=2,L J = (L+1) - I 10 R(J) = R(J+1) + (S * C(J))

Nonrecursive Multiply Add, Broadcast, Forward Count, Normal Order (Q8MAx00)		
Name:	Q8MA200	Q8MA400
Broadcast Operand:	v2	v4
Example:	CALL Q8MA200(R(2),A(2),S,C(2),L-1)	CALL Q8MA400(R(2),S,B(2),C(2),L-1)
DO Loop Equivalent of STACKLIB Call:	DO 10 I=2,L 10 R(I) = A(I) * S + C(I)	DO 10 I=2,L 10 R(I) = S * B(I) + C(I)

Nonrecursive Multiply Add, Broadcast, Forward Count, Reverse Order (Q8AMx01)		
Name:	Q8AM101	Q8AM201
Broadcast Operand:	v1	v2
Example:	CALL Q8AM101(R(2),A(2),B(2),S,L-1)	CALL Q8AM201(R(2),A(2),S,C(2),L-1)
DO Loop Equivalent of STACKLIB Call:	DO 10 I=2,L 10 R(I) = A(I) + B(I) * S	DO 10 I=2,L 10 R(I) = A(I) + S * C(I)

TABLE 11-3. STACKLIB ROUTINES (Contd)

Recursive Subtract Multiply, Forward Count, Reverse Order (Q8SM0x1)		
Name:	Q8SM011	Q8SM021
Recursive Operand:	v1	v2
Example:	CALL Q8SM011(R(2),A(2),B(2),R(1),L-1)	CALL Q8SM021(R(2),A(2),R(1),C(2),L-1)
DO Loop Equivalent of STACKLIB Call:	DO 10 I=2,L 10 R(I) = A(I) - (B(I) * R(I-1))	DO 10 I=2,L 10 R(I) = A(I) - (R(I-1) * C(I))

Recursive Subtract Multiply, Backwards Count, Reverse Order (Q8SM0x3)		
Name:	Q8SM013	Q8SM023
Recursive Operand:	v1	v2
Example:	CALL Q8SM013(R(L-1),A(L-1),B(L-1),R(L),L-1)	CALL Q8SM023(R(L-1),A(L-1),R(L),C(L-1),L-1)
DO Loop Equivalent of STACKLIB Call:	DO 10 I=2,L J = (L+1) - I 10 R(J) = A(J) - (B(J) * R(J+1))	DO 10 I=2,L J = (L+1) - I 10 R(J) = A(J) - (R(J+1) * C(J))

Nonrecursive Subtract Multiply, Broadcast, Forward Count, Reverse Order (Q8SMx01)		
Name:	Q8SM101	Q8SM201
Broadcast Operand:	v1	v2
Example:	CALL Q8SM101(R(2),A(2),B(2),S,L-1)	CALL Q8SM201(R(2),A(2),S,C(2),L-1)
DO Loop Equivalent of STACKLIB Call:	DO 10 I=2,L 10 R(I) = A(I) - B(I) * S	DO 10 I=2,L 10 R(I) = A(I) - (S * C(I))

Recursive Divide Add, Broadcasts, Backwards Count, Reverse Order (Q8DAxx3)		
Name:	Q8DA523	Q8DA613
Recursive Operand:	v2	v1
Broadcast Operands:	v1,v4	v2,v4
Example:	CALL Q8DA523(R(L-1),S,R(L),T,L-1)	CALL Q8DA613(R(L-1),S,T,R(L),L-1)
DO Loop Equivalent of STACKLIB Call:	DO 10 I=2,L J = (L+1) - I 10 R(J) = S / (R(J+1) + T)	DO 10 I=2,L J = (L+1) - I 10 R(J) = S / (T + R(J+1))

TABLE 11-3. STACKLIB ROUTINES (Contd)

Dot Product		
Name:	Q8DC0000	Q8DC0010
Number of Vectors:	2	1
Example:	CALL Q8DC0000(S,A(2),B(2),L-1)	CALL Q8DC0010(S,A(2),L-1)
DO Loop Equivalent of STACKLIB Call:	DO 10 I=2,L 10 S = S + A(I) * B(I)	DO 10 I=2,L 10 S = S + A(I) * A(I)

Sum of Vector Elements	
Name:	Q8DA0000
Example:	CALL Q8DA0000(S,A(2),L-1)
DO Loop Equivalent of STACKLIB Call:	DO 10 I=2,L 10 S = S + A(I)

STACKLIB SUBROUTINE NAMING CONVENTION

Except for the dot product and sum of vector elements subroutines, each STACKLIB subroutine is named according to the following naming convention:

For dyadic operations:

Q8fbrm

For triadic operations:

Q8fsbrm

f Letter designating the first arithmetic operation (A=add, S=subtract, M=multiply, and D=divide).

s Letter designating the second arithmetic operation (triadic operations only) (A=add, S=subtract, M=multiply, and D=divide).

b Number designating the broadcast (scalar or invariant) operands (0=no broadcast operands, 1=operand v1, 2=operand v2, 3=operands v1 and v2, 4= operand v4, 5=operands v1 and v4, and 6=operands v2 and v4).

r Number designating the recursive operand whose value is replaced by the result vector address offset by 1 (0=no recursive operands, 1=operand v1, 2=operand v2, 3=operands v1 and v2, 4= operand v4, 5=operands v1 and v4, and 6=operands v2 and v4).

m Number designating the count and order for a triadic operation (for a dyadic operation, the value is always 0) (0=forward count, normal order; 1=forward count, reverse order; 2=backwards count, normal order; and 3=backwards count, reverse order).

For example, using this naming convention, the name Q8MA020 indicates that the subroutine has the following characteristics:

Arithmetic operations: Multiply and add

Broadcast operands: None

Recursive operands: v2

Count and order: Forward and normal

Not all STACKLIB routines that could be defined by this naming convention actually exist. Table 11-3 lists the available STACKLIB routines with their characteristics.

STACKLIB CALL FORMATS

A STACKLIB call for a dyadic operation specifies four parameter values as follows:

CALL Q8xnnn(result,v2,v1,length)

result First address of the result vector

v2 First address of the left operand vector

v1 First address of the right operand vector

length Positive integer indicating the number of results to be produced

A STACKLIB call for a triadic operation is the same as for a dyadic call except that it specifies a third operand vector as follows:

```
CALL Q8xxnnn(result,v4,v2,vl,length)
```

result	First address of the result vector
v4	First address of the left operand vector
v2	First address of the middle operand vector
vl	First address of the right operand vector
length	Positive integer indicating the number of results to be produced

STACKLIB ARGUMENT CHECKING AND ERROR PROCESSING

Because the STACKLIB subroutines are implemented for high performance, the subroutines do not check for argument validity. All vectors are assumed to be of type real (an explicit descriptor must be of type real) and the length is assumed to be a positive integer. Standard FORTRAN calling sequence conventions are assumed.

If an operand is designated as recursive, the routine ignores the value specified as the first address of that operand vector. Instead, the routine uses the first address of the result vector offset by 1. For example, if operand vl is designated as recursive, any value specified for vl is ignored.

An error condition that occurs during execution of a STACKLIB routine can only be detected by the Data Flag Branch processor.

The FORTRAN 200 language includes a number of special call statements that directly generate machine language instructions.

Before using special call statements, you should be familiar with the machine language instructions, the assembly language symbolic instructions, the CYBER 200 hardware, and the CYBER 200 Assembler. See the appropriate hardware reference manual and the CYBER 200 Assembler reference manual for more information.

Special call statements are not recommended for most FORTRAN applications. Use them only when absolutely necessary and only to accomplish specific programming tasks.

A special call statement consists of the keyword CALL followed by a special call name and an argument list. A FORTRAN program unit can contain one or more special call statements. See figure 12-1 for the format of a special call statement.

CALL $m(a_1, \ldots, a_n)$

m	One of the special call names beginning with Q8
a_1	An argument corresponding to one of the fields of the instruction format

Figure 12-1. Special Call Statement Format

ARGUMENTS

Arguments that are used in special call statements must be label references, symbolic references, or literals.

The arguments used in the special call statements correspond to the fields of the machine language instructions. The arguments used in the assembly language symbolic instructions can appear differently, but they are functionally the same. For example, the machine language instruction #78 is RTOR R,T in assembly language, but the FORTRAN special call statement is CALL Q8RTOR(R,,T). The extra comma is required because of the missing operand S that can appear between operands R and T in some instructions.

The arguments in a special call statement must rigidly follow the machine language instruction format. Any missing arguments must be indicated by a comma, although trailing missing arguments can be omitted.

Normally, the arguments must appear in the order of the fields of the machine language instruction. An exception is that only one argument is allowed for an entire 8-bit subfunction field (G bits) having 1-bit subfields. Another exception is that for the indexed branch instructions #B0 through #B5, the combined Y and B fields require only one argument in the special call statement; this argument is usually a label reference. If the combined fields represent two register designators, however, you must use a 16-bit hexadecimal constant.

When an argument is a literal, place the value of the literal in the instruction field. When an argument is a variable, place the register number of the variable in the instruction field.

When necessary, the compiler generates a load instruction before the special call statement machine instruction and a store instruction afterward. Only registers #20 through #FF are used for this purpose. The low-order temporary registers may be used, but the generated object code destroys their contents when it reverts to using standard FORTRAN 200 statements.

The bits in the subfunction field (G bits) of machine language instruction formats #1, #2, and #3 are not checked against the operands to ensure the validity of the instruction. There will be no warning messages for any special call statements that transfer control in or out of the range of a DO loop, if-block, else-block, elseif-block, where-block, or otherwise-block.

LABEL REFERENCES

A label reference is designated by prefixing a statement label with an asterisk or ampersand. Label references can appear in the following machine instruction formats:

- In the combined Y and B fields of a format #C machine language instruction
- In the 48-bit I field of a format #5 machine language instruction, except when only 24 bits of the I field are used by certain instructions
- In the 8-bit I field of a format #9 and format #B machine language instruction

If the label reference appears in the combined Y and B fields of a format #C machine language instruction, the label reference is translated into a code halfword offset from the special call statement to the statement in the program unit identified by the label. The labeled statement can appear before or after the special call statement.

If the label reference appears in the 48-bit I field of a format #5 machine language instruction, the label reference is translated into the bit address of the statement tagged by the label. This bit address is a relative bit address with respect to the base address of the code section of the program unit in which the special call statement appears.

If the label reference appears in the 8-bit immediate field of a #2F, #32, or #33 machine language instruction, the label reference is translated into a halfword offset from the special call statement

to the statement tagged by the label. If the resulting halfword offset exceeds a magnitude of 255, a 0 is used to initialize the 8-bit I field; no warning message is issued.

A label reference is the only kind of argument that can appear in the branch field of a relative branch machine language instruction.

SYMBOLIC REFERENCES

A symbolic reference can be a variable, array element, descriptor, descriptor array element, or vector reference of type integer, half-precision, real, or logical. Fullword symbolic references in halfword instruction fields are illegal, and halfword symbolic references in fullword instruction fields are illegal.

Symbolic references can appear in any 8-bit register designator field, except in halfword registers. Registers that are modified by branch instructions must not be referenced symbolically.

LITERALS

A literal can be a decimal, hexadecimal, bit, character, or Hollerith constant, and can be used for any instruction field. Any missing arguments are assumed to be constants whose values are 0. In general, constants are interpreted as register designators rather than as data used by an instruction.

SPECIAL CALL STATEMENT EXAMPLES

The following paragraphs contain two sets of special call statement examples. The first set of examples uses special calls to store information in the register file. The second set of examples uses special call statements to vectorize DO loops.

USING SPECIAL CALLS TO MANIPULATE REGISTERS

See figure 12-2 for the first example of the use of special call statements. The call to Q8BSAVE sets register #3 to the bit address of the next instruction, which has statement label 10. The call to Q8EX in the statement labeled 10 sets register #4 to the bit offset of statement 10 from the base address of the code section. In the next statement, the call to Q8SUBX sets integer variable CB to the base address of the code section. The next call to Q8EX sets variable I to the bit offset of the statement labeled 20. The next statement sets variable L20 to the actual address of the statement labeled 20. This information is then used in the call to Q8BGE.

See figure 12-3 for the second example of the use of special call statements. Each special call statement places the character string AB in register #41. These examples demonstrate how literals can be used as arguments; however, the use of register #41 would probably cause an error because registers #20 to #FF are assigned by the compiler.

```
      .
      .
      .
      INTEGER CB,L20
      CALL Q8BSAVE(3,,3)
10    CALL Q8EX(4,&10)
      CALL Q8SUBX(3,4,CB)
      CALL Q8EX(I,&20)
      L20=I+CB
      .
      .
      .
      CALL Q8BGE(A,B,L20)
      .
      .
      .
20
```

Figure 12-2. Special Call Statement Example #1

```
      .
      .
      .
      CALL Q8ES(65,'AB')
      CALL Q8ES(X'41',X'4142')
      CALL Q8ES(B'1000001','AB')
      CALL Q8ES('A','AB')
      .
      .
      .
```

Figure 12-3. Special Call Statement Example #2

See figure 12-4 for the third example of the use of special call statements. If J is assigned to register #22 by the compiler, this example generates machine language instructions; see figure 12-5 for the assembly language representation of the machine language instructions generated.

```
      .
      .
      .
      CALL Q8ES(3,1)
      CALL Q8ES(4,2)
      CALL Q8ADDX(3,4,J)
      .
      .
      .
```

Figure 12-4. Special Call Statement Example #3

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

Figure 12-5. Generated Code

If J is not assigned to register #22 by the compiler, different machine language instructions would be generated; see figure 12-6 for the alternate assembly language representation of the machine language instructions generated.

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

Figure 12-6. Alternate Generated Code

USING SPECIAL CALLS TO VECTORIZE DO LOOPS

DO loops often perform the loop calculation selectively. Many contain tests that determine the operand pairs on which the loop calculation is performed. For other DO loops, the index increment is not 1 (nonunit stride). These types of DO loops can often be transformed into vector arithmetic operations controlled by a bit vector. This transformation can be done using either the WHERE statement (described in section 9) or special calls. (Use of the WHERE statement is recommended because it produces more readable code.)

NOTE

The automatic vectorizer can vectorize nonunit stride DO loops.

Figure 12-6.1 contains examples of DO loop transformations using special calls. These special calls are used:

Q8ADDNV	Vector addition
Q8SUBNV	Vector subtraction
Q8MPYSV	Vector multiplication
Q8DIVSV	Vector division

Example 1:

The following DO loop performs a multiplication operation for each 1 bit in the BITV vector. It uses the BTOL function to convert each bit in the vector to its corresponding logical value (false for each 0 bit; true for each 1 bit).

```
      DO 10 I=1,N
10 IF (BTOL(BITV(I))) R(I) = A(I) * B(I)
```

The same loop can be written as the following special call statement:

```
CALL Q8MPYSV(X'00',,A(1;N),,B(1;N),BITV(1;N),R(1;N))
```

Example 2:

The following DO loop has a nonunit stride (the loop index is incremented by 2):

```
      DO 10 I=1,N,2
10 A(I) = B(I) + 2.0 * C(I)
```

The DO loop can be rewritten using a bit vector (BITV), a temporary vector (TEMP), and a special call statement, as follows:

```
BITV(1;N) = Q8VMKO(1,2;BITV(1;N))   Stores alternate 1 and 0 bits beginning with a 1 bit.
TEMP(1;N) = 2.0 * C(1;N)
CALL Q8ADDNV(X'00',,B(1;N),,TEMP(1;N),BITV(1;N),A(1;N))
```

Example 3:

The following DO loop is the same as the DO loop in example 2 except it performs a subtraction operation instead of an addition operation:

```
      DO 10 I=1,N,2
10 A(I) = B(I) - 2.0 * C(I)
```

The DO loop can be rewritten using a bit vector (BITV), a temporary vector (TEMP), and a special call statement, as follows:

```
BITV(1;N) = Q8VMKO(1,2;BITV(1;N))   Stores alternate 1 and 0 bits beginning with a 1 bit.
TEMP(1;N) = 2.0 * C(1;N)
CALL Q8SUBNV(X'00',,B(1;N),,TEMP(1;N),BITV(1;N),A(1;N))
```

Figure 12-6.1. Special Call Examples That Vectorize Do Loops (Sheet 1 of 2)

Example 4:

The following DO loop performs an addition operation when a test is evaluated true:

```
      DO 10 I=1,100
      A(I) = B(I) + C(I)
      IF (A(I) .EQ. 0.0) D(I) = E(I) + F(I)
   10 G(I) = A(I) + D(I)
```

The DO loop can be restructured using a bit vector (BITV) and a special call statement, as follows:

```
   A(1;100) = B(1;100) + C(1;100)
   BITV(1;100) = A(1;100) .EQ. 0.0     A 1 bit is stored corresponding to each 0.0 element in A.

   CALL Q8ADDNV(X'00',,E(1;100),,F(1;100),BITV(1;100),D(1;100))
   G(1;100) = A(1;100) + D(1;100)
```

Example 5:

The following DO loop performs a division operation when a test is evaluated true:

```
      DO 10 I=1,100
   10 IF (C(I) .NE. 0.0) A(I) = B(I) / C(I)
```

The DO loop can be restructured using a bit vector (BITV) and a special statement, as follows:

```
   BITV(1;100) = C(1;100) .NE. 0.0     A 1 bit is stored corresponding to each nonzero element in C.

   CALL Q8DIVSV(X'00',,B(1;100),,C(1;100),BITV(1;100),A(1;100))
```

Example 6:

The following DO loop performs a multiplication operation and a constant assignment only when a test is evaluated false:

```
      DO 10 I=1,100
      A(I) = B(I) * C(I)
      IF (A(I) .NE. 0.0) GO TO 20
      A(I) = D(I) * E(I)
      B(I) = 1.0
   20 CONTINUE
      F(I) = A(I) + B(I)
   10 CONTINUE
```

The DO loop can be restructured by reversing the test so that a 1 bit is stored in the bit vector BITV corresponding to each case where the multiplication operation and constant assignment should be performed. The Q8VCTRL function is used to perform the constant assignment; the value 1.0 is stored in each element in B that corresponds to a 1 bit in BITV.

```
   A(1;100) = B(1;100) * C(1;100)
   BITV(1;100) = A(1;100) .EQ. 0.0
   CALL Q8MPYSV(X'00',,D(1;100),,E(1;100),BITV(1;100),A(1;100))
   B(1;100) = Q8VCTRL(1.0,BITV(1;100);B(1;100))
   F(1;100) = A(1;100) + B(1;100)
```

Figure 12-6.1. Special Call Examples That Vectorize Do Loops (Sheet 2 of 2)

WARNING ABOUT USING Q8 SPECIAL CALLS

Because the Q8 special calls directly access the CYBER 205 hardware functions, they are extremely powerful. However, the proper use of such calls requires a thorough understanding of the hardware specifics. The compiler typically translates the calls with their arguments directly into machine instructions. It does very little parameter validity checking and knows very little, if anything, about hardware exceptional conditions. Given that the compiler has no provisions to handle hardware idiosyncrasies, the translated code might not perform as intended.

Following are two situations that we know can cause problems. When you write machine level code by writing Q8 special calls, you must be aware of these situations and code around them accordingly.

Q8LINKV Special Call Warning

One such case is the Q8LINKV special call. The LINKV machine instruction enables efficient processing of two sequential vector instructions, where the second instruction needs as input the output from the first instruction. Two of the hardware requirements of using LINKV are the following:

The two vector instructions are linkable.

The two vector instructions immediately follow the LINKV instruction. If there are any scalar instructions between LINKV and the two vector instructions that the LINKV is linking, a hardware illegal instruction fault results.

In the following example, A, B, C, and D are vector descriptors:

```
CALL Q8LINKV(X'10')
CALL Q8MPYSV(X'10',,A,,B,,C)
CALL Q8ADDNV(X'08',,C,,1.0,,D)
```

All the descriptors must be assigned to registers before compilation. However, if any descriptor is stored in memory, the compiler generates code to load that descriptor into a register. The compiler inserts the code, a scalar instruction, between the LINKV and MPYSV. The position of this scalar instruction makes it an illegal hardware instruction.

Overlapping Scalar Instruction Warnings

Another situation that can be a problem is shown in the following example:

```
IMPLICIT HALF PRECISION (A-Z)
CALL Q8DIVSH(R1,S1,T1)
CALL Q8ADDNH(R2,S2,T1)
```

In this example, the half-precision variable T1 should end up with the result of R2 + S2. However, if R2 or S2 happens to form an odd/even half-word register pair with T1, a problem occurs. In this situation, the hardware allows the result of R1/S1 to be inserted into T1 after the result of R2 + S2 was inserted into T1. Thus, the final value in T1 is the result of the divide rather than the result of the add. This example shows that T1 might not contain the expected result.

This kind of problem is not limited to only the two instructions, Q8DIVSH and Q8ADDNH, in the above example. It can occur whenever all of the following conditions hold:

Two 32-bit scalar instructions overlap in execution; that is, the hardware performs them in parallel.

There is a T to T (output register designation) conflict; that is, both of the 32-bit scalar instructions put their results in the same output register.

R2 or S2 forms an odd/even half-word register pair with T1; that is, one of the input variables in the second 32-bit scalar instruction forms an odd/even half-word register pair with the output register in that instruction.

The second 32-bit scalar instruction takes less time to complete than the first.

SPECIAL CALL FORMATS

Each special call name is a mnemonic preceded by Q8. The mnemonics are identical or similar to the CYBER 200 assembly language mnemonics.

The first field of each machine instruction is the operation code field (F), which indicates the function to be performed. The special call name supplies the operation code in the generated instruction. Other operands are specified as arguments in the special call. See table 12-1 for a description of the operand designators.

TABLE 12-1. OPERAND DESIGNATORS

Designator	Format Type	Definition
A	1 and 3	Specifies a register that contains a field length and base address for the corresponding source vector or string field.
	2	Specifies a register that contains the base address for a source sparse vector field.
	C	Specifies a fullword or halfword register, the length and type of which is determined by G field bits.
B	1 and 3	Specifies a register that contains a field length and base address for the corresponding source vector or string field.
	2	Specifies a register that contains the base address for a source sparse vector field.
	C	Specifies a register that contains the branch base address in the rightmost 48 bits, or must be set to zero, depending on G bit 2.
C	1, 2, and 3	Specifies a register that contains the field length and base address for storing the result vector or string field.
	C	Specifies a fullword or halfword register that contains the sum of (A) + (X) for indexed branch instructions, but must be set to zero for compare floating-point instructions.
C + 1	1	Specifies a register containing the offset for C and Z vector fields. If the C + 1 designator is used, the C designator must specify an even-numbered register.
G	1, 2, 3, 9, B and C	8-bit designator specifies certain subfunction conditions. Subfunctions include length of operands (32- or 64-bit), normal or broadcast source vectors, and so on. The number of bits used in the G designator varies with instructions. For some format 3 instructions, used as an immediate byte I8.
I	5	48-bit index used to form the branch address in a B6 branch instruction. In BE and BF index instructions, I is a 48-bit operand.
	6	In 3E and 3F index instructions, I is a 16-bit operand.
	B	In the 33 branch instruction, the 6-bit I is the number of the DFB object bits used in the branching operation.
R	4	In the register and 3D instructions, R is the register containing an operand to be used in an arithmetic operation.
	5 and 6	In the 3E, 3F, BE, and BF index instructions, R is a destination register for the transfer of an operand or operand sum. In the B6 branch instruction, this register contains an item count used to form the branch address.
	7, 8, and A	R specifies registers and branching conditions given in the individual instruction descriptions.
S	4	In the register and 3D instructions, S is a register containing an operand to be used in an arithmetic operation.
	7, 8, and 9	S specifies registers and branching conditions given in the individual instruction descriptions.
T	4	T specifies a destination register for the transfer of the arithmetic results.
	7, 8, 9 and B	T specifies a register that contains the base address and, in some cases, the field length of the corresponding result field or branch address.
	A	T specifies a register containing the old state of a register, DFB register, and so on; in an index, branch, or inter-register transfer operation.

TABLE 12-1. OPERAND DESIGNATORS (Contd)

Designator	Format Type	Definition
X	1 and 3	Specifies a register that contains the offset or index for vector or string source field A.
	2	Specifies a register that contains length and base address for order vector corresponding to source sparse vector field A.
	C	In indexed branch or compare floating-point instructions (B0 - B5), specifies a fullword or halfword register that contains an operand, the length and type of which is determined by G field bits.
Y	1 and 3	Specifies a register that contains the offset or index for vector or string field B.
	2	Specifies a register that contains the length and base address for the order vector corresponding to source sparse vector field B.
	C	In indexed branch or compare floating-point instructions (B0 - B5), Y specifies one of the following: a register that contains an index used to form the branch address; part of the halfword item count in a relative branch; or a destination register for storing a one if the condition is met, and zero otherwise.
Z	1	Z specifies a register that contains the base address for the order vector used to control the result vector in field C.
	2	Z specifies a register that contains the length and base address for the order vector corresponding to result sparse vector field C.
	3	Z specifies a register that contains the index for result field C.
	C	In indexed branch or compare floating-point instructions (B0 - B5), contains a two's complement or unsigned integer that determines whether the condition is met.

See table 12-2 for the special call formats. The bits of the subfunction field (G bits) that can be set to 0 or 1 are indicated with an x. In table 12-2, the following notations are used:

f Indicates a fullword register containing an operand

h Indicates a halfword register containing an operand

a Indicates a fullword register containing an address; the length field is ignored

i Indicates a fullword register containing an index

d Indicates a fullword register containing a descriptor

e Indicates a fullword register with an exponent field that contains a length operand

eh Indicates a halfword register with an exponent field that contains a length operand

FP Is an abbreviation for floating-point

OV Is an abbreviation for order vector

RJ Is an abbreviation for right-justified

SE	Is an abbreviation for sign-extended
YB	Indicates a combined Y and B field
.OP.	Indicates one of the logical operators .EQ., .NE., .GE., or .LT.
U	Indicates upper result
L	Indicates lower result
N	Indicates normalized upper result
S	Indicates significant result

See table 12-3 for the special call names listed by operation code.

See figure 12-7 for an illustration of the 12 machine instruction formats.

TABLE 12-2. SPECIAL CALL FORMATS

Special Call	Op Code (Hex)	Instruction Format	Description	G Bits
CALL Q8ABS(R_f,,T_f)	79	A	Absolute, fullword FP: ABS(R_f) → T_f	
CALL Q8ABSH(R_h,,T_h)	59	A	Absolute, halfword FP: ABS(R_h) → T_h	
CALL Q8ABSV(G,X,A,,,Z,C)	99	1	Absolute, vector: ABS(A) → C	xxxx oooo
CALL Q8ACPS(G,X,A,Y,B,Z,C)	CF	1	A_n.GE.B_n → C_n,set Z_n,OV length → Z_{0-15}	xooo xxxx
CALL Q8ADDL(R_f,S_f,T_f)	61	4	Add lower, fullword FP: $((R_f)+(S_f))_L$ → T_f	
CALL Q8ADDLEN(R_e,S_f,T_e)	2B	4	Add to length, $R_{0-15}+S_{48-63}$ → T_{0-15},R_{16-63} → T_{16-63}	
CALL Q8ADDLH(R_h,S_h,T_h)	41	4	Add lower, halfword FP: $((R_h)+(S_h))_L$ → T_h	
CALL Q8ADDLS(G,X,A,Y,B,Z,C)	A1	2	Add lower, sparse vector: $(A+B)_L$ → C	xxxx xxxx
CALL Q8ADDLV(G,X,A,Y,B,Z,C)	81	1	Add lower, vector: $(A+B)_L$ → C	xxxx xxxx
CALL Q8ADDN(R_f,S_f,T_f)	62	4	Add normalized, fullword FP: $((R_f)+S_f))_N$ → T_f	
CALL Q8ADDNH(R_h,S_h,T_h)	42	4	Add normalized, halfword FP: $((R_h)+(S_h))_N$ → T_h	
CALL Q8ADDNS(G,X,A,Y,B,Z,C)	A2	2	Add normalized, sparse vector: $(A+B)_N$ → C	xxxx xxxx
CALL Q8ADDNV(G,X,A,Y,B,Z,C)	82	1	Add normalized, vector: $(A+B)_N$ → C	xxxx xxxx
CALL Q8ADDU(R_f,S_f,T_f)	60	4	Add upper, fullword FP: $((R_f)+(S_f))_U$ → T_f	
CALL Q8ADDUH(R_h,S_h,T_h)	40	4	Add upper, halfword FP: $((R_h)+(S_h))_U$ → T_h	
CALL Q8ADDUS(G,X,A,Y,B,Z,C)	A0	2	Add upper, sparse vector: $(A+B)_U$ → C	xxxx xxxx
CALL Q8ADDUV(G,X,A,Y,B,Z,C)	80	1	Add upper, vector: $(A+B)_U$ → C	xxxx xxxx
CALL Q8ADDX(R_f,S_f,T_f)	63	4	Add index, fullword: $R_{16-63}+S_{16-63}$ → T_{16-63},R_{0-15} → T_{0-15}	
CALL Q8ADDXV(G,X,A,Y,B,Z,C)	83	1	Add index, vector: $A_{16-63}+B_{16-63}$ → C_{16-63},A_{0-15} → C_{0-15}	oxxx xooo

TABLE 12-2. SPECIAL CALL FORMATS (Contd)

Special Call	Op Code (Hex)	Instruction Format	Description	G Bits
CALL Q8ADJE(R_f,S_f,T_f)	75	4	Adjust exponent, fullword FP: (R_f) per S → T_f	
CALL Q8ADJEH(R_h,S_h,T_h)	55	4	Adjust exponent, halfword FP: (R_h) per S → T_h	
CALL Q8ADJEV(G,X,A,Y,B,Z,C)	95	1	Adjust exponent, vector: A per B → C	xxxx xooo
CALL Q8ADJM(G,X,A,,,Z,C)	D1	1	Adjacent mean: $(A_{n+1}+A_n)/2 \rightarrow C_n$	xxxo oooo
CALL Q8ADJS(R_f,S_f,T_f)	74	4	Adjust significance, fullword FP: (R_f) per S → T_f	
CALL Q8ADJSH(R_h,S_h,T_h)	54	4	Adjust significance, halfword FP: (R_h) per S → T_h	
CALL Q8ADJSV(G,X,A,Y,B,Z,C)	94	1	Adjust significance, vector: A per B → C	xxxx xooo
CALL Q8AND(,X,A,Y,B,Z,C)	F1	3	Logical AND: $A \cdot B \rightarrow C$	
CALL Q8ANDN(,X,A,Y,B,Z,C)	F6	3	Logical AND NOT: $A \cdot \overline{B} \rightarrow C$	
CALL Q8ANDNV(G,X,A,Y,B,Z,C)	9D	1	Logical AND NOT: $A \cdot B \rightarrow C$, vector	xxxx x11o
CALL Q8ANDV(G,X,A,Y,B,Z,C)	9D	1	Logical AND: $A \cdot B \rightarrow C$, vector	xxxx xoo1
CALL Q8AVG(G,X,A,,,Z,C)	D0	1	Vector average: $(A_n+B_n)/2 \rightarrow C_n$	xxxx xooo
CALL Q8AVGD(G,X,A,,,Z,C)	D4	1	Vector average difference: $(A_n-B_n)/2 \rightarrow C_n$	xxxx xooo
CALL Q8BAB(G,S_a,T_a)	32	9	Branch and alter bit: (S_a) is bit to be altered, (T_a) is branch address	xxxx oxxo
CALL Q8BADF(G,I6,T_a)	33	B	D.F. reg. bit branch and alter: I6 is bit altered, (T_a) is branch address	xxxx oxxo
CALL Q8BARB(G,S,T)	2F	9	Branch to [S] on condition of bit 63 of register T	xxxx oooo
CALL Q8BEQ(R_f,S_f,T_a)	24	8	Branch to (T_a) if (R_f).EQ.(S_f), fullword FP compare	
CALL Q8BGE(R_f,S_f,T_a)	26	8	Branch to (T_a) if (R_f).GE.(S_f), fullword FP compare	
CALL Q8BHEQ(R_h,S_h,T_a)	20	8	Branch to (T_a) if (R_h).EQ.(S_h), halfword FP compare	
CALL Q8BHGE(R_h,S_h,T_a)	22	8	Branch to (T_a) if (R_h).GE.(S_h), halfword FP compare	
CALL Q8BHLT(R_h,S_h,T_a)	23	8	Branch to (T_a) if (R_h).LT.(S_h), halfword FP compare	
CALL Q8BHNE(R_h,S_h,T_a)	21	8	Branch to (T_a) if (R_h).NE.(S_h), halfword FP compare	
CALL Q8BIM(R_i,I48)	B6	5	Branch immediate to (R_i)+I48	

TABLE 12-2. SPECIAL CALL FORMATS (Contd)

Special Call	Op Code (Hex)	Instruction Format	Description	G Bits
CALL Q8BKPT(R_a)	04	4	Breakpoint: R_{16-63} → breakpoint register	
CALL Q8BLT(R_f,S_f,T_a)	27	8	Branch to (T_a) if (R_f).LT.(S_f), fullword FP compare	
CALL Q8BNE(R_f,S_f,T_a)	25	8	Branch to (T_a) if (R_f).NE.(S_f), fullword FP compare	
CALL Q8BSAVE(R_f,S_i,T_a)	36	7	Set (R_f) to next instruction address, branch to [T_a+S_i]	
CALL Q8BTOD(R_f,,T_f)	11	A	Convert binary R to packed BCD T, fixed length	
CALL Q8CFPEQ(G,X,A,YB)	B0	C	Compare FP and branch if (A).OP. (X) then branch to (Y) + (B) or relative from current location	xlox xxxx
CALL Q8CFPGE(G,X,A,YB)	B2	C		xlox xxxx
CALL Q8CFPGT(G,X,A,YB)	B5	C		xlox xxxx
CALL Q8CFPLE(G,X,A,YB)	B4	C		xlox xxxx
CALL Q8CFPLT(G,X,A,YB)	B3	C		xlox xxxx
CALL Q8CFPNE(G,X,A,YB)	B1	C		xlox xxxx
CALL Q8CFPEQ(G,X,A,Y)	B0	C	Compare FP and set condition if (A).OP.(X) then 1 → Y else 0 → Y	xllx xxxx
CALL Q8CFPGE(G,X,A,Y)	B2	C		xllx xxxx
CALL Q8CFPGT(G,X,A,Y)	B5	C		xllx xxxx
CALL Q8CFPLE(G,X,A,Y)	B4	C		xllx xxxx
CALL Q8CFPLT(G,X,A,Y)	B3	C		xllx xxxx
CALL Q8CFPNE(G,X,A,Y)	B1	C		xllx xxxx
CALL Q8CLG(R_f,,T_f)	72	A	Ceiling, fullword FP: nearest integer .GE.(R_f) → T_f	
CALL Q8CLGH(R_h,,T_h)	52	A	Ceiling, halfword FP: nearest integer .GE.(R_h) → T_h	
CALL Q8CLGV(G,X,A,,,Z,C)	92	1	Ceiling, vector: nearest integer .GE.A → C	xxxx oooo
CALL Q8CLOCK(,,T_f)	39	A	Transmit (real time clock) → T_{16-63},0 → T_{0-15}	
CALL Q8CMPEQ(G,X,A,Y,B,Z)	C4	1	Vector compare, form order vector: if (A_n).OP.(B_n), set bit Z_n in order vector	xoox xooo
CALL Q8CMPGE(G,X,A,Y,B,Z)	C6	1		xoox xooo
CALL Q8CMPLT(G,X,A,Y,B,Z)	C7	1		xoox xooo
CALL Q8CMPNE(G,X,A,Y,B,Z)	C5	1		xoox xooo
CALL Q8CNTEQ(R_d,S_i,T_f)	1E	7	Count: # of leading bits equal to bit at [R+S] → T_{48-63}	
CALL Q8CNTO(R_d,S_i,T_f)	1F	7	Count 1's in field R: # of 1's in field [R+S] → T_{48-63}	
CALL Q8CON(R_f,,T_h)	76	A	Contract, fullword FP: R_{64} → T_{32}	
CALL Q8CONV(G,X,A,,,Z,C)	96	1	Contract, vector: A_{64} → C_{32}	oxxx oooo
CALL Q8CPSB(R_d,S_e,T_d)	14	7	Compress bit string: every R_n sub-string from R_n+S_n pattern → T	
CALL Q8CPSV(G,,A,,,Z,C)	BC	2	Compress vector: vector A → sparse C, controlled by OV Z	xxoo oooo
CALL Q8DBNZ(R_f,S_i,T_a)	35	7	(R_f)-1 → (R_f), if (R_f) ≠0 branch to [T_a+S_i]	
CALL Q8DELTA(G,X,A,,,Z,C)	D5	1	Vector delta: (A_{n+1}-A_n) → C_n	xxxo oooo

TABLE 12-2. SPECIAL CALL FORMATS (Contd)

Special Call	Op Code (Hex)	Instruction Format	Description	G Bits
CALL Q8DIVS(R_f,S_f,T_f)	6F	4	Divide significant, fullword FP: $((R_f)/(S_f))_S \rightarrow T_f$	
CALL Q8DIVSH(R_h,S_h,T_h)	4F	4	Divide significant, halfword FP: $((R_h)/(S_h))_S \rightarrow T_h$	
CALL Q8DIVSS(G,X,A,Y,B,Z,C)	AF	2	Divide significant, sparse vector: $(A/B)_S \rightarrow C$	xxxx xxxx
CALL Q8DIVSV(G,X,A,Y,B,Z,C)	8F	1	Divide significant, vector: $(A/B)_S \rightarrow C$	xxxx xxxx
CALL Q8DIVU(R_f,S_f,T_f)	6C	4	Divide upper, fullword FP: $((R_f)/(S_f))_U \rightarrow T_f$	
CALL Q8DIVUH(R_h,S_h,T_h)	4C	4	Divide upper, halfword FP: $((R_h)/(S_h))_U \rightarrow T_h$	
CALL Q8DIVUS(G,X,A,Y,B,Z,C)	AC	2	Divide upper, sparse vector: $(A/B)_U \rightarrow C$	xxxx xxxx
CALL Q8DIVUV(G,X,A,Y,B,Z,C)	8C	1	Divide upper, vector: $(A/B)_U \rightarrow C$	xxxx xxxx
CALL Q8DOTV(G,X,A,Y,B,Z,C)	DC	1	Dot product vector: $A \cdot B \rightarrow C,C+1$	xxoo oooo
CALL Q8DTOB(R_f,,T_f)	10	A	Convert packed BCD to binary T, fixed length	
CALL Q8ELEN(R_e,I16)	2A	6	Enter length: I16 → R_{0-15},R_{16-63} unchanged	
CALL Q8ES(R_fI16)	3E	6	Enter short, fullword: I16 → R_{16-63},RJ,SE,0 → R_{0-15}	
CALL Q8ESH(R_h,I16)	4D	6	Enter short, halfword: I16 → R_{8-31},RJ,SE,0 → R_{0-7}	
CALL Q8EX(R_f,I48)	BE	5	Enter index, fullword: I48 → R_{16-63},0 → R_{0-15}	
CALL Q8EXH(R_h,I24)	CD	5	Enter index, halfword: I24 → R_{8-31},0 → R_{0-7}	
CALL Q8EXIT	09	4	Exit force, job mode to monitor mode	
CALL Q8EXP(R_e,,T_f)	7A	A	Exponent, fullword: R_{0-15} → T_{16-63},SE,0 → T_{0-15}	
CALL Q8EXPHR$_{eh}$,,T_h)	5A	A	Exponent, halfword: R_{0-7} → T_{8-31},SE,0 → T_{0-7}	
CALL Q8EXPV(G,X,A,,,Z,C)	9A	1	Exponent vector: A_{0-15} → C_{48-63},SE,0 → C_{0-15}	xxxx oooo
CALL Q8EXTB(R_f,S_d,T_f)	6E	4	Extract bits from R_f to T_f per S_d	
CALL Q8EXTH(R_h,,T_f)	5C	A	Extend halfword FP: $R_{32} \rightarrow T_{64}$	
CALL Q8EXTV(G,X,A,,,Z,C)	9C	1	Extend vector: $A_{32} \rightarrow C_{64}$	oxxx oooo
CALL Q8EXTXH(R_h,,T_f)	5D	A	Extend index, halfword FP: $R_{8-31} \rightarrow T_{16-63}$SE,$R_{0-7} \rightarrow T_{0-15}$,SE	
CALL Q8FAULT(G)	06	7	Simulate fault	oooo xxxx

TABLE 12-2. SPECIAL CALL FORMATS (Contd)

Special Call	Op Code (Hex)	Instruction Format	Description	G Bits
CALL Q8FLR(R_f,,T_f)	71	A	Floor, fullword FP: nearest integer .LE.(R_f) → T_f	
CALL Q8FLRH(R_h,,T_h)	51	A	Floor, halfword FP: nearest integer .LE.(R_h) → T_h	
CALL Q8FLRV(G,X,A,,,Z,C)	91	1	Floor, vector: nearest integer .LE.A → C	xxxx oooo
CALL Q8IBNZ(R_f,S_i,T_a)	31	7	(R_f)+1 → (R_f), if (R_f) ≠0 branch to [T_a,S_i]	
CALL Q8IBXEQ(G,X,A,YB,Z,C)	B0	C	Increment and branch index:	xoox xxxx
CALL Q8IBXGE(G,X,A,YB,Z,C)	B2	C	(A)+(X) → C,A_{len} → C_{len};	xoox xxxx
CALL Q8IBXGT(G,X,A,YB,Z,C)	B5	C	if (A)+(X).OP.(Z) then branch to	xoox xxxx
CALL Q8IBXLE(G,X,A,YB,Z,C)	B4	C	(Y)+(B) or YB halfwords from	xoox xxxx
CALL Q8IBXLT(G,X,A,YB,Z,C)	B3	C	current location	xoox xxxx
CALL Q8IBXNE(G,X,A,YB,Z,C)	B1	C		xoox xxxx
CALL Q8IBXEQ(G,X,A,Y,,Z,C)	B0	C	Increment index and set condition:	xolx xxxx
CALL Q8IBXGE(G,X,A,Y,,Z,C)	B2	C	(A)+(X) → C,A_{len} → C_{len};	xolx xxxx
CALL Q8IBXGT(G,X,A,Y,,Z,C)	B5	C	if (A)+(X).OP.(Z) then 1 → Y else	xolx xxxx
CALL Q8IBXLE(G,X,A,Y,,Z,C)	B4	C	0 → Y	xolx xxxx
CALL Q8IBXLT(G,X,A,Y,,Z,C)	B3	C		xolx xxxx
CALL Q8IBXNE(G,X,A,Y,,Z,C)	B1	C		xolx xxxx
CALL Q8IDLE	00	4	Idle: enable external interrupts and idle	
CALL Q8INSB(R_f,S_d,T_f)	6D	4	Insert bits from R_f to T_f per S_d	
CALL Q8INTVAL(G,,A,,B,Z,C)	DF	1	Interval vector: A+((n-2)*B) → C	xxxo oooo
CALL Q8IOR(,X,A,Y,B,Z,C)	F2	3	Logical inclusive OR: A+B → C	
CALL Q8IORV(G,X,A,Y,B,Z,C)	9D	1	Logical vector inclusive OR: A+B → C	xxxx xolo
CALL Q8IS(R_f,I16)	3F	6	Increase short, fullword: R_{16-63}+I16 → R_{16-63},R_{0-15} unchanged	
CALL Q8ISH(R_h,I16)	4E	6	Increase short, halfword: R_{8-31}+I16 → R_{8-31},R_{0-7} unchanged	
CALL Q8IX(R_f,I48)	BF	5	Increase index, fullword: I48+R → R	
CALL Q8IXH(R_h,I24)	CE	5	Increase index, halfword: I24+R → R	
CALL Q8LINKV(G)	56	7	Link next two vector instructions	ooox xooo
CALL Q8LOD(R_a,S_i,T_f)	7E	7	Load fullword: load [R_a+S_i] → T_f	
CALL Q8LODAR	0D	4	Load associative registers: beginning at $400xx_8$ → AR	
CALL Q8LODC(R_a,S_i,T_f)	12	7	Load byte: [R_a+S_i] → T_{56-63},0 → T_{0-55}	
CALL Q8LODH(R_a,S_i,T_h)	5E	7	Load halfword: load [R_a+S_i] → T_h	
CALL Q8LODKEY(R_f,S_a,T_a)	0F	4	Load key from (R_f), translate virtual (S_a) to absolute T_a	

TABLE 12-2. SPECIAL CALL FORMATS (Contd)

Special Call	Op Code (Hex)	Instruction Format	Description	G Bits
CALL Q8LSDFR(R_f,,T_f)	3B	A	Load and store data flag register: (DFR) → T_f,(R_f) → DFR	
CALL Q8LTOL(R_e,,T_e)	38	A	Transmit length R_{0-15} to length T_{0-15}, T_{6-63} unchanged	
CALL Q8LTOR(R_e,,T_f)	7C	A	Length to register, fullword FP: R_{0-15} → T_{48-63},0 → T_{0-47}	
CALL Q8MASKB(R_d,S_d,T_d)	16	7	Mask bit strings: alternate (R_d) string and (S_d) string → T string	
CALL Q8MASKO(R_e,S_e,T_d)	1D	7	Form bit mask: repeat (R_n) ones and (S_n)-(R_n) zeros → T string	
CALL Q8MASKV(G,,A,,B,Z,C)	BB	2	If Z_n=1,A_n → C_n; if Z_n=0, B_n → C_n; result length → C_{0-15}	xoox xooo
CALL Q8MASKZ(R_e,S_e,T_d)	1C	7	Form mask: repeat (R_n) zeros and (S_n)-(R_n) ones → T string	
CALL Q8MAX(G,X,A,,B,Z,C)	D8	1	Vector maximum: A_{max} → C, item count → B	xxoo oxoo
CALL Q8MCMPW(G,X,A,,B,,C)	CC	3	Find A=B per maskword C, A index incremented by number of words	oooo ooox
CALL Q8MIN(G,X,A,,B,Z,C)	D9	1	Vector minimum: A_{min} → C, item count → B	xxoo oxoo
CALL Q8MOVL(G,X,A,,B,Z,C)	F8	3	Move bytes left: A → C (left to right)	xxxx oxox
CALL Q8MPYL(R_f,S_f,T_f)	69	4	Multiply lower, fullword FP: $((R_f)*(S_f))_L$ → T_f	
CALL Q8MPYLH(R_h,S_h,T_h)	49	4	Multiply lower, halfword FP: $((R_h)*(S_h))_L$ → T_h	
CALL Q8MPYLS(G,X,A,Y,B,Z,C)	A9	2	Multiply lower, sparse vector: $(A*B)_L$ → C	xxxx xxxx
CALL Q8MPYLV(G,X,A,Y,B,Z,C)	89	1	Multiply lower, vector: $(A*B)_L$ → C	xxxx xxxx
CALL Q8MPYS(R_f,S_f,T_f)	6B	4	Multiply significant, fullword FP: $((R_f)*(S_f))_S$ → T_f	
CALL Q8MPYSH(R_h,S_h,T_h)	4B	4	Multiply significant, halfword FP: $((R_h)*(S_h))_S$ → T_h	
CALL Q8MPYSS(G,X,A,Y,B,Z,C)	AB	2	Multiply significant, sparse vector: $(A*B)_S$ → C	xxxx xxxx
CALL Q8MPYSV(G,X,A,Y,B,Z,C)	8B	1	Multiply significant, vector: $(A*B)_S$ → C	xxxx xxxx
CALL Q8MPYU(R_f,S_f,T_f)	68	4	Multiply upper, fullword FP: $((R_f)*(S_f))_U$ → T_f	
CALL Q8MPYUH(R_h,S_h,T_h)	48	4	Multiply upper, halfword FP: $((R_h)*(S_h))_U$ → T_h	
CALL Q8MPYUS(G,X,A,Y,B,Z,C)	A8	2	Multiply upper, sparse vector: $(A*B)_U$ → C	xxxx xxxx
CALL Q8MPYUV(G,X,A,Y,B,Z,C)	88	1	Multiply upper, vector: $(A*B)_U$ → C	xxxx xxxx

TABLE 12-2. SPECIAL CALL FORMATS (Contd)

Special Call	Op Code (Hex)	Instruction Format	Description	G Bits
CALL Q8MPYX(R_f,S_f,T_f)	3D	4	Multiply index, fullword: R_{16-63}*S_{16-63} → T_{16-63},0 → T_{0-15}	
CALL Q8MPYXH(R_h,S_h,T_h)	3C	4	Multiply index, halfword: R_{8-31}*S_{8-31} → T_{8-31},0 → T_{0-7}	
CALL Q8MRGB(R_d,S_d,T_d)	15	7	Merge bit strings: interleave (R_d) string with (S_d) string → T_d string	
CALL Q8MRGV(G,,A,,B,Z,C)	BD	2	Merge vector: if Z_n=1, A_n → C_n; if Z_n=0, B_n → C_n; result length → C_{0-15}	xoox xoox
CALL Q8MTIME(R_f)	0A	4	Transmit (R_f) → monitor interval timer	
CALL Q8NAND(,X,A,Y,B,Z,C)	F3	3	Logical NAND: $\overline{A \cdot B}$ → C	
CALL Q8NANDV(G,X,A,Y,B,Z,C)	9D	1	Logical NAND: $\overline{A \cdot B}$ → C, vector	xxxx xo11
CALL Q8NOR(,X,A,Y,B,Z,C)	F4	3	Logical NOR: $\overline{A+B}$ → C	
CALL Q8NORV(G,X,A,Y,B,Z,C)	9D	1	Logical NOR: $\overline{A+B}$ → C, vector	xxxx xloo
CALL Q8ORN(,X,A,Y,B,Z,C)	F5	3	Logical OR NOT: $A+\overline{B}$ → C	
CALL Q8ORNV(G,X,A,Y,B,Z,C)	9D	1	Logical OR NOT: $A+\overline{B}$ → C, vector	xxxx xlo1
CALL Q8ORV(G,X,A,Y,B,Z,C)	9D	1	Logical inclusive OR: A+B → C, vector	xxxx xolo
CALL Q8PACK(R_f,S_f,T_f)	7B	4	Pack, fullword FP: R_{48-63} and S_{16-63} → T_f	
CALL Q8PACKH(R_h,S_h,T_h)	5B	4	Pack, halfword FP: R_{24-31} and S_{8-31} → T_h	
CALL Q8PACKV(G,X,A,Y,B,Z,C)	9B	1	Pack, vector: A_{48-63} and B_{16-63} → C	xxxx xooo
CALL Q8POLYEV(G,X,A,Y,B,Z,C)	DE	1	Polynomial evaluation: A_n per B → C_n	xxxx oooo
CALL Q8PRODCT(G,X,A,,,Z,C)	DB	1	Vector product: Product (A_0,A_1,...A_n) → C	xxoo oooo
CALL Q8RAND(R_f,S_f,T_f)	2D	4	Logical AND: R,S → T	
CALL Q8RCON(R_f,,T_h)	77	A	Rounded contract, fullword FP: R_{64} → T_{32}	
CALL Q8RCONV(G,X,A,,,Z,C)	97	1	Rounded contract, vector: A_{64} rounded → 32	oxxx oooo
CALL Q8RIOR(R_f,S_f,T_f)	2E	4	Logical inclusive OR: R,S → T	
CALL Q8RJTIME(,,T_f)	37	A	Read job interval timer to (T_f)	
CALL Q8RTOR(R_f,,T_f)	78	A	Register to register fullword transmit: (R_f) → T_f	
CALL Q8RTORH(R_h,,T_h)	58	A	Register to register halfword transmit: (R_h) → T_h	
CALL Q8RXOR(R_f,S_f,T_f)	2C	4	Logical exclusive OR: R,S → T	

TABLE 12-2. SPECIAL CALL FORMATS (Contd)

Special Call	Op Code (Hex)	Instruction Format	Description	G Bits
CALL Q8SCNLEQ(I8,S_i,T_d)	28	7	Scan left to right from $[T_d,S_i]$ for byte equal to I8, index S_i	
CALL Q8SELEQ(G,X,A,Y,B,Z,C)	C0	1	Vector select: if A_n.OP.B_n,	xxox xooo
CALL Q8SELGE(G,X,A,Y,B,Z,C)	C2	1	then count up to the condition	xxox xooo
CALL Q8SELLT(G,X,A,Y,B,Z,C)	C3	1	met → C	xxox xooo
CALL Q8SELNE(G,X,A,Y,B,Z,C)	C1	1		xxox xooo
CALL Q8SETCF(R_f)	08	4	Input/output: set channel (R_f) channel flag	
CALL Q8SHIFT (R_f,S_f,T_f)	34	4	Shift R_f by (S_f) → T_f	
CALL Q8SHIFTI(R_f,I8,T_f)	30	7	Shift R_f by I8 → T_f	
CALL Q8SHIFTV(G,X,A,Y,B,Z,C)	8A	1	Shift A by B → C, vector	xxxx xxxx
CALL Q8SQRT(R_f,,T_f)	73	A	Significant square root, fullword FP: $(SQRT(R_f))_S \rightarrow T_f$	
CALL Q8SQRTH(R_h,,T_h)	53	A	Significant square root, halfword FP: $(SQRT(R_h))_S \rightarrow T_h$	
CALL Q8SQRTV(G,X,A,,,Z,C)	93	1	Significant square root, vector: $SQRT(A)_S \rightarrow C$	xxxx oxxo
CALL Q8SRCHEQ(G,,A,,B,Z,C)	C8	1	Vector search from indexed list:	xxxo oooo
CALL Q8SRCHGE(G,,A,,B,Z,C)	CA	1	each $(A_n).OP.(B_n)$, count → C_n	xxxo oooo
CALL Q8SRCHLT(G,,A,,B,Z,C)	CB	1		xxxo oooo
CALL Q8SRCHNE(G,,A,,B,Z,C)	C9	1		xxxo oooo
CALL Q8STO(R_a,S_i,T_f)	7F	7	Store, fullword: store (T_f) address $[R_a+S_i]$	
CALL Q8STOAR	0C	4	Store associative registers: AR → $400xx_8$ and higher addresses	
CALL Q8STOC(R_a,S_i,T_f)	13	7	Store byte (character): T_{56-63} → address $[R_a+S_i]$	
CALL Q8STOH(R_a,S_i,T_h)	5F	7	Store, halfword: (T_h) → address R_a+S_i	
CALL Q8SUBL(R_f,S_f,T_f)	65	4	Subtract lower, fullword FP: $((R_f)-(S_f))_L \rightarrow T_f$	
CALL Q8SUBLH(R_h,S_h,T_h)	45	4	Subtract lower, halfword FP: $((R_h)-(S_h))_L \rightarrow T_f$	
CALL Q8SUBLS(G,X,A,Y,B,Z,C)	A5	2	Subtract lower, sparse vector: $(A-B)_L \rightarrow C$	xxxx xxxx
CALL Q8SUBLV(G,X,A,Y,B,Z,C)	85	1	Subtract lower, vector: $(A-B)_L \rightarrow C$	xxxx xxxx
CALL Q8SUBN(R_f,S_f,T_f)	66	4	Subtract normalized, fullword FP: $((R_f)-(S_f))_N \rightarrow T_f$	
CALL Q8SUBNH(R_h,S_h,T_h)	46	4	Subtract normalized, halfword FP: $((R_h)-(S_h))_N \rightarrow T_f$	
CALL Q8SUBNS(G,X,A,Y,B,Z,C)	A6	2	Subtract normalized, sparse vector: $(A-B)_N \rightarrow C$	xxxx xxxx
CALL Q8SUBNV(G,X,A,Y,B,Z,C)	86	1	Subtract normalized, vector: $(A-B)_N \rightarrow C$	xxxx xxxx

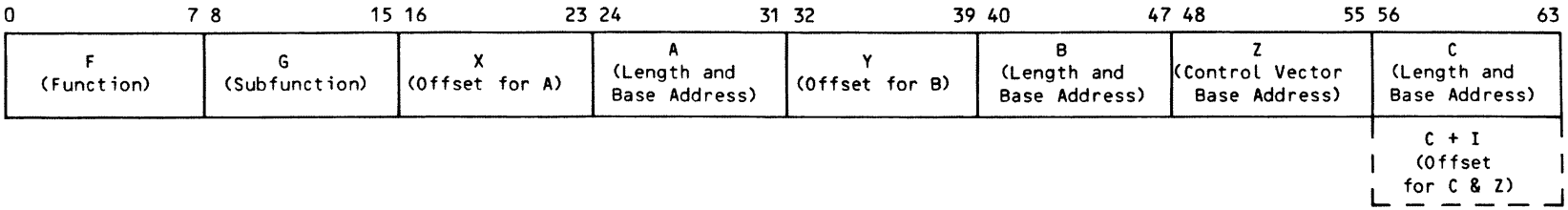
TABLE 12-2. SPECIAL CALL FORMATS (Contd)

Special Call	Op Code (Hex)	Instruction Format	Description	G Bits
CALL Q8SUBU(R_f,S_f,T_f)	64	4	Subtract upper, fullword FP: $((R_f)-(S_f))_U \rightarrow T_f$	
CALL Q8SUBUH(R_h,S_h,T_h)	44	4	Subtract upper, halfword FP: $((R_h)-(S_h))_U \rightarrow T_h$	
CALL Q8SUBUS(G,X,A,Y,B,Z,C)	A4	2	Subtract upper, sparse vector: $(A-B)_U \rightarrow C$	xxxx xxxx
CALL Q8SUBUV(G,X,A,Y,B,Z,C)	84	1	Subtract upper, vector: $(A-B)_U \rightarrow C$	xxxx xxxx
CALL Q8SUBX(R_f,S_f,T_f)	67	4	Subtract index: $R_{16-63}-S_{16-63} \rightarrow T_{16-63}$, $R_{0-15} \rightarrow T_{0-15}$	
CALL Q8SUBXV(G,X,A,Y,B,Z,C)	87	1	Subtract index, vector: $A_{16-63}-B_{16-63} \rightarrow C_{16-63}$, $A_{0-15} \rightarrow C_{0-15}$	oxxx xooo
CALL Q8SUM(G,X,A,,,Z,C)	DA	1	Vector sum: $Sum(A_0,A_1,...A_n) \rightarrow C,C+1$	xxoo oooo
CALL Q8SWAP(R_d,S_f,T_d)	7D	7	Swap registers: start with S_f, storing at T_d and loading from R_d	
CALL Q8TLXI(R_a,S_i,T_f)	0E	4	Translate external interrupt: (T_f)=priority, branch to R_a $[S_i]$	
CALL Q8TRU(R_f,,T_f)	70	A	Truncate, fullword FP: nearest integer .LE.$(R_f) \rightarrow T_f$	
CALL Q8TRUH(R_h,,T_h)	50	A	Truncate, halfword FP: nearest integer .LE.$(R_h) \rightarrow T_h$	
CALL Q8TRUV(G,X,A,,,Z,C)	90	1	Truncate, vector: nearest integer .LE.$(A) \rightarrow C$	xxxx oooo
CALL Q8VREVV(G,X,A,,,Z,C)	B8	1	Transmit vector reversed to vector: $A_{rev} \rightarrow C$	xxxo oooo
CALL Q8VSB(,,T_a)	03	4	Void instruction stack and branch to (T_a)	
CALL Q8VTOV(G,X,A,,,Z,C)	98	1	Vector to vector transmit: $A \rightarrow C$	xxxx oooo
CALL Q8VTOVX(G,,A,,B,,C)	B7	1	Vector to vector indexed transmit: $B \rightarrow C$ indexed by A	xooo xxxx
CALL Q8VXTOV(G,,A,,B,,C)	BA	1	Vector to vector indexed transmit: B indexed by $A \rightarrow C$	xooo oxxx
CALL Q8WJTIME(R_f)	3A	A	Transmit $(R_f) \rightarrow$ job interval timer	
CALL Q8XOR(,X,A,Y,B,Z,C)	F0	3	Logical exclusive OR: $A-B \rightarrow C$	
CALL Q8XORN(,X,A,Y,B,Z,C)	F7	3	Logical equivalence (exclusive OR NOT): $A-\overline{B} \rightarrow C$	
CALL Q8XORNV(G,X,A,Y,B,Z,C)	9D	1	Logical exclusive OR NOT: (equivalence) $A-\overline{B} \rightarrow C$, vector	xxxx x111
CALL Q8XORV(G,X,A,Y,B,Z,C)	9D	1	Logical exclusive OR: $A-B \rightarrow C$, vector	xxxx xooo

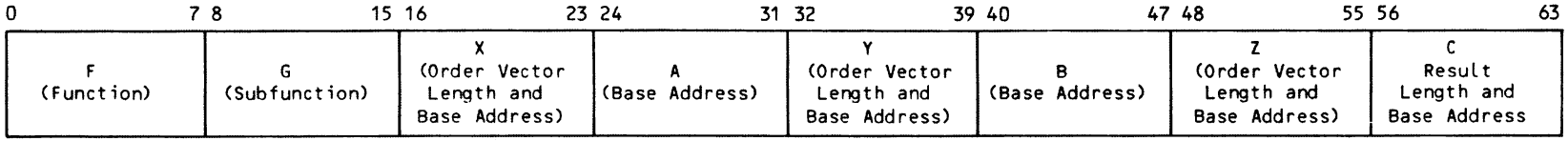
TABLE 12-3. SPECIAL CALLS LISTED BY OP CODE

Op Code (hex)	Special Call	Op Code (hex)	Special Call	Op Code (hex)	Special Call	Op Code (hex)	Special Call
00	Q8IDLE	41	Q8ADDLH	7C	Q8LTOR	B2	Q8IBXGE
03	Q8VSB	42	Q8ADDNH	7D	Q8SWAP		Q8CFPGE
04	Q8BKPT	44	Q8SUBUH	7E	Q8LOD	B3	Q8IBXLT
06	Q8FAULT	45	Q8SUBLH	7F	Q8STO		Q8CFPLT
08	Q8SETCF	46	Q8SUBNH	80	Q8ADDUV	B4	Q8IBXLE
09	Q8EXIT	48	Q8MPYUH	81	Q8ADDLV		Q8CFPLE
0A	Q8MTIME	49	Q8MPYLH	82	Q8ADDNV	B5	Q8IBXGT
0C	Q8STOAR	4B	Q8MPYSH	83	Q8ADDXV		Q8CFPGT
0D	Q8LODAR	4C	Q8DIVUH	84	Q8SUBUV	B6	Q8BIM
0E	Q8TLXI	4D	Q8ESH	85	Q8SUBLV	B7	Q8VTOVX
0F	Q8LODKEY	4E	Q8ISH	86	Q8SUBNV	B8	Q8VREVV
10	Q8DTOB	4F	Q8DIVSH	87	Q8SUBXV	BA	Q8VXTOV
11	Q8BTOD	50	Q8TRUH	88	Q8MPYUV	BB	Q8MASKV
12	Q8LODC	51	Q8FLRH	89	Q8MPYLV	BC	Q8CPSV
13	Q8STOC	52	Q8CLGH	8A	Q8SHIFTV	BD	Q8MRGV
14	Q8CPSB	53	Q8SQRTH	8B	Q8MPYSV	BE	Q8EX
15	Q8MRGB	54	Q8ADJSH	8C	Q8DIVUV	BF	Q8IX
16	Q8MASKB	55	Q8ADJEH	8F	Q8DIVSV	C0	Q8SELEQ
1C	Q8MASKZ	56	Q8LINKV	90	Q8TRUV	C1	Q8SELNE
1D	Q8MASKO	58	Q8RTORH	91	Q8FLRV	C2	Q8SELGE
1E	Q8CNTEQ	59	Q8ABSH	92	Q8CLGV	C3	Q8SELLT
1F	Q8CNTO	5A	Q8EXPH	93	Q8SQRTV	C4	Q8CMPEQ
20	Q8BHEQ	5B	Q8PACKH	94	Q8ADJSV	C5	Q8CMPNE
21	Q8BHNE	5C	Q8EXTH	95	Q8ADJEV	C6	Q8CMPGE
22	Q8BHGE	5D	Q8EXTXH	96	Q8CONV	C7	Q8CMPLT
23	Q8BHLT	5E	Q8LODH	97	Q8RCONV	C8	Q8SRCHEQ
24	Q8BEQ	5F	Q8STOH	98	Q8VTOV	C9	Q8SRCHNE
25	Q8BNE	60	Q8ADDU	99	Q8ABSV	CA	Q8SRCHGE
26	Q8BGE	61	Q8ADDL	9A	Q8EXPV	CB	Q8SRCHLT
27	Q8BLT	62	Q8ADDN	9B	Q8PACKV	CC	Q8MCMPW
28	Q8SCNLEQ	63	Q8ADDX	9C	Q8EXTV	CD	Q8EXH
2A	Q8ELEN	64	Q8SUBU	9D	Q8ANDNV	CE	Q8IXH
2B	Q8ADDLEN	65	Q8SUBL		Q8ANDV	CF	Q8ACPS
2C	Q8RXOR	66	Q8SUBN		Q8NANDV	D0	Q8AVG
2D	Q8RAND	67	Q8SUBX		Q8NORV	D1	Q8ADJM
2E	Q8RIOR	68	Q8MPYU		Q8ORNV	D4	Q8AVGD
2F	Q8BARB	69	Q8MPYL		Q8ORV	D5	Q8DELTA
30	Q8SHIFTI	6B	Q8MPYS		Q8XORNV	D8	Q8MAX
31	Q8IBNZ	6C	Q8DIVU		Q8XORV	D9	Q8MIN
32	Q8BAB	6D	Q8INSB	A0	Q8ADDUS	DA	Q8SUM
33	Q8BADF	6E	Q8EXTB	A1	Q8ADDLS	DB	Q8PRODCT
34	Q8SHIFT	6F	Q8DIVS	A2	Q8ADDNS	DC	Q8DOTV
35	Q8DBNZ	70	Q8TRU	A4	Q8SUBUS	DE	Q8POLYEV
36	Q8BSAVE	71	Q8FLR	A5	Q8SUBLS	DF	Q8INTVAL
37	Q8RJTIME	72	Q8CLG	A6	Q8SUBNS	F0	Q8XOR
38	Q8LTOL	73	Q8SQRT	A8	Q8MPYUS	F1	Q8AND
39	Q8CLOCK	74	Q8ADJS	A9	Q8MPYLS	F2	Q8IOR
3A	Q8WJTIME	75	Q8ADJE	AB	Q8MPYSS	F3	Q8NAND
3B	Q8LSDFR	76	Q8CON	AC	Q8DIVUS	F4	Q8NOR
3C	Q8MPYXH	77	Q8RCON	AF	Q8DIVSS	F5	Q8ORN
3D	Q8MPYX	78	Q8RTOR	B0	Q8IBXEQ	F6	Q8ANDN
3E	Q8ES	79	Q8ABS		Q8CFPEQ	F7	Q8XORN
3F	Q8IS	7A	Q8EXP	B1	Q8IBXNE	F8	Q8MOVL
40	Q8ADDUH	7B	Q8PACK		Q8CFPNE	FB	Q8ZTOD

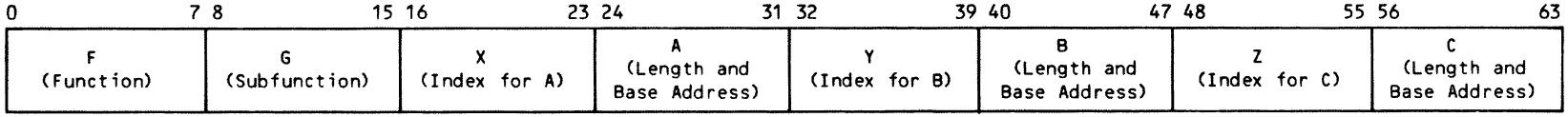
FORMAT 1 - Used for Vector, Vector Macro, and Some Nontypical Instructions:



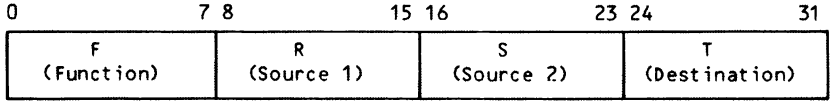
FORMAT 2 - Used for Sparse Vector and Some Nontypical Instructions:



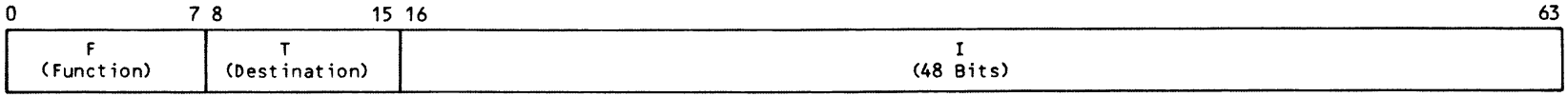
FORMAT 3 - Used for Logical String and String Instructions:



FORMAT 4 - Used for Some Register, for all Monitor instructions, and for the #3D, and #04 Nontypical Instructions:



FORMAT 5 - Used for the #BE, #BF, #CD, and #CE Index Instructions and for the #B6 Branch Instruction:



FORMAT 6 - Used for the #3E, #3F, #4D, and #4E Index Instructions and the #2A Register Instruction:

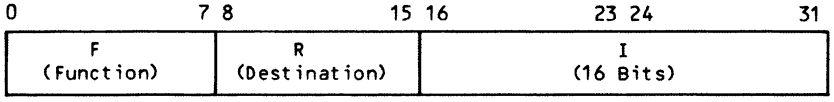
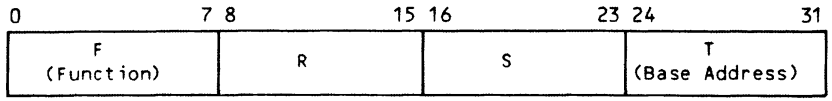
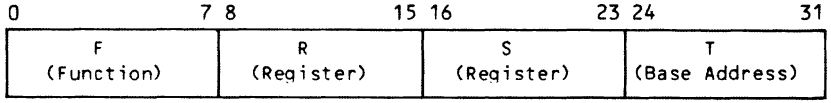


Figure 12-7. Instruction Formats (Sheet 1 of 2)

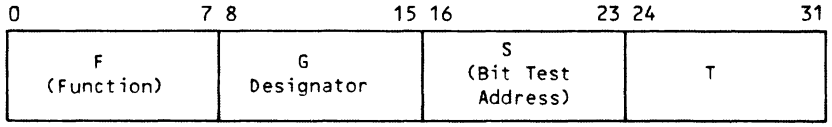
FORMAT 7 - Used for Some Branch and Nontypical Instructions:



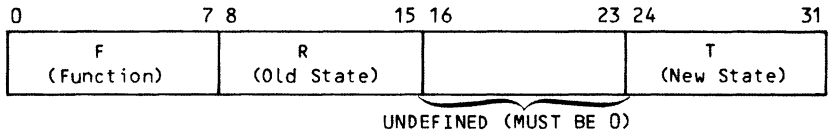
FORMAT 8 - Used for Some Branch Instructions:



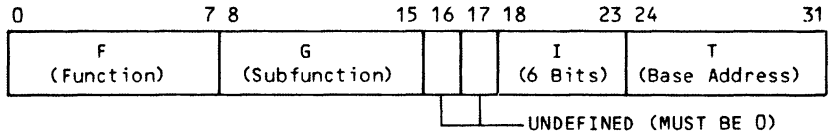
FORMAT 9 - Used for the #32 Branch Instruction:



FORMAT A - Used for Some Index, Branch, and Register Instructions:



FORMAT B - Used for the #33 Branch Instruction:



FORMAT C - Used for the #B0 through #B5 Branch Instructions:

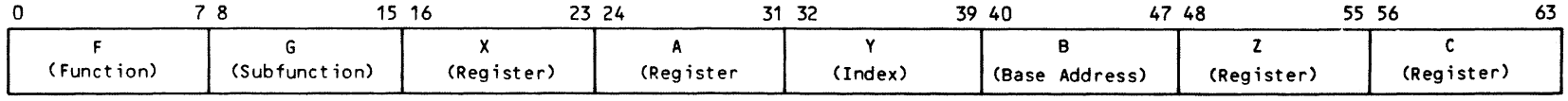


Figure 12-7. Instruction Formats (Sheet 2 of 2)

This section describes the relationship between FORTRAN programs and other CYBER 200 software products. This section includes general descriptions of:

The basic procedure for compiling and executing a FORTRAN program

The major operating system features that can be accessed from a FORTRAN program using the System Interface Language (SIL) routines and the debugging facilities

The conventions used in linking the program units of a FORTRAN program

PROGRAM COMPILATION, LOADING, AND EXECUTION

There are two ways to compile, load, and execute a FORTRAN 200 program on the CYBER 200:

1. Execute the following three CYBER 200 control statements in a CYBER 200 batch job or interactive session:

   a. A FTN200 statement (described in section 14) to compile the program and create a relocatable binary.

   b. A LOAD statement (described in the VSOS Reference manual, Volume 1) to load the relocatable binary and to build a controlee file.

   c. A control statement naming the controlee file (The default file name is GO.) This control statement executes the FORTRAN 200 program.

   Future runs of the program can execute the stored object code. This is useful when a program needs to be executed several times.

2. Execute a FTN200 control statement with the GO parameter set to 1. When you specify GO=1 in a CYBER 200 batch job or interactive session, your FORTRAN 200 program is loaded and executed if it compiles with no fatal errors. The LOAD utility is not used which means that the operating system overhead of bringing in the LOAD is eliminated. However, the object code is not stored for future use, so each run of the program requires a compile. This method is good for short programs that will not be executed often, or for debugging running programs.

   You can use the GO parameter only if the system shared library is active in your installation. Appendix F gives the procedure to compile, load, and execute a FORTRAN 200 program, when the system shared library is not active in your installation.

The input file read by the FORTRAN 200 compiler can contain more than one subprogram. Although subprograms can be compiled without a main program, a program cannot be loaded or executed without a main program.

Program compilation, loading, and execution are possible only within a CYBER 200 job or interactive session. For a full description of CYBER 200 jobs and interactive sessions, refer to the VSOS Reference Manual, volume 1.

CYBER 200 JOB SUBMITTAL

To submit a CYBER 200 job, you login to a front-end system, create a CYBER 200 job file, and then submit the CYBER 200 job file to the CYBER 200 system for execution. The actual statement used to submit the job differs depending on the front-end operating system. Ask site personnel for the appropriate statement for your site. Appendix F gives an example of a batch CYBER 200 job submittal on a VSOS release 2.1.6 installation.

Figure 13-1 shows a NOS 2 interactive session in which a batch job is submitted to a CYBER 200 system. The following paragraphs describe the session shown in the figure.

After logging in to the NOS 2 system and specifying NORMAL and BATCH modes, the user gets and displays the contents of the CYBER 200 job file.

The first statement in the job file is the job statement (JOB1,STTY3.). The parameter STTY3 specifies the CYBER 200 mainframe identifier TY3. (You must ask site personnel for the mainframe identifier effective at your site.)

The second statement reads the file ACCT205 which contains the CY205 accounting information. You can also use the USER statement which specifies the user name, account name, and user password.

As described earlier, the three control statements: (FTN200, LOAD, and GO) will compile, load, and execute a FORTRAN 200 program. If you use the GO parameter, the one control statement FTN200, will execute the program if there are no fatal errors from the compilation.

Figure 13-1, a FORTRAN 200 program source is listed between two --EOR-- indicators. The XEDIT editor uses --EOR-- to display an end-of-record delimiter. One end-of-record delimiter separates the program from the control statements, and another separates the program from the input data (here, a sequence of integers 1,2,3,4,5).

```
(date)   (time)
(NOS 2 logon banner)
FAMILY: ,user1,passwd <-- Entry to logon to NOS 2.
JSN: ABGU,NAMIAF <-- IAF logon is successful.

 READY.
normal <-- Ensures the terminal is in
 READY.    normal mode.
batch <-- Switches NOS 2 to batch mode.
RFL,0.
/get,job1 <-- Gets the file JOB1.
/xedit,job1 <-- Calls a text editor to
 XEDIT 3.1.00    display the job file
?? p*            contents.
/JOB
/NOSEQ
JOB1,STTY3.
/READ,ACCT205
COMMENT.  JOB EXAMPLE TO DEMONSTRATE STANDARD 2.2 RUN
COMMENT.  FILE ACCT205 ON THE NOS SYSTEM CONTAINS THE
COMMENT.  CY205 ACCOUNTING INFORMATION.  AN EXAMPLE
COMMENT.  OF THE CONTENT OF THIS FILE MIGHT BE
COMMENT.
COMMENT.  USER(U=205_USERNUM,PA=205_PASSWORD,AC=205_ACCOUNT)
COMMENT.  RESOURCE(TL=10,JCAT=JDEFAULT)
COMMENT.
FTN200. <-- Calls the compiler.
LOAD. <-- LOAD to write controlee.
GO.       Execute.
/EOR
      PROGRAM LOOP
      K=0
      DO 10 I = 1,5
      READ 100,J
  100 FORMAT(I1) <-- FORTRAN source code.
   10 K = J + K
      PRINT 200
  200 FORMAT(' THE SUM IS')
      PRINT 300,K
  300 FORMAT(1X,I2)
      STOP
      END
/EOR
1 <-- Input data to the FORTRAN
2     program.
3
4
5
/EOF
END OF FILE
```

Figure 13-1. Example of a NOS 2 Interactive Session
Submitting a CYBER 200 Job (Sheet 1 of 3)

```
?? end
JOB1 IS A LOCAL FILE
/submit,job1,e <-------------------------- Submits job to be executed
 12.29.00  SUBMIT COMPLETE.  JSN IS AAKZ.   on 205.
/enquire,jsn <---------------------------- Displays job status.

 JSN SC CS DS LID STATUS         JSN SC CS DS LID STATUS

 AAKZ.B.  .RB.TY3.INPUT QUEUE    AAKM.T.ON.BC.   .EXECUTING
 AAKK.B.  .RB.   .PRINT QUEUE
/enquire,jsn

 JSN SC CS DS LID STATUS         JSN SC CS DS LID STATUS

 AAKZ.B.  .RB.TY3.INPUT QUEUE    AAKM.T.ON.BC.   .EXECUTING
 AAKK.B.  .RB.   .PRINT QUEUE
/enquire,jsn

 AAKM.T.ON.BC.   .EXECUTING      AALD.R.  .RB.M10.PRINT QUEUE
/qget,aald,pr <--------------------------- Output file becomes local
 QGET COMPLETE.                            file.
/xedit,aald <----------------------------- Calls a text editor to
 XEDIT 3.1.00                              display the output file.
?? l-rsys- <------------------------------ Locates the dayfile.
--EOR--
--EOR--
--EOR--
1 12.17.10 RSYSK11   VS22DBC   013428   DEVSYS   G   JOB1AABC
   08/28/85
?? p* <----------------------------------- Lists the dayfile.
1 12.17.10 RSYSK11   VS22DBC   013428   DEVSYS   G   JOB1AABC
  08/28/85
  12.17.11  RESOURCE,TL=10.
  12.17.11  COMMENT.  JOB EXAMPLE TO DEMONSTRATE STANDARD 2.2 RUN
  12.17.11  COMMENT.  FILE ACCT205 ON THE NOS SYSTEM CONTAINS THE
  12.17.11  COMMENT.  CY205 ACCOUNTING INFORMATION.  AN EXAMPLE
  12.17.11  COMMENT.  OF THE CONTENT OF THIS FILE MIGHT BE
  12.17.11  COMMENT.
  12.17.11  COMMENT.  USER(U=205_USERNUM,PA=205_PASSWORD,AC=205_ACCOUNT)
  12.17.11  COMMENT.  RESOURCE(TL=10,JCAT=JDEFAULT)
  12.17.11  COMMENT.
  12.17.11  FTN200.
  12.17.11  FORTRAN 200 CYCLE K10      BUILT 08/17/85 18:00
  12.17.12  COMPILING LOOP
  12.17.13        NO ERRORS
  12.17.13     0.148 SECONDS COMPILATION TIME
  12.17.13   ALL DONE
  12.17.13  LOAD.
  12.17.14   LOAD R2.2 CYCLE K10
  12.17.15   ALL DONE
  12.17.15  GO.
```

Figure 13-1. Example of a NOS 2 Interactive Session
Submitting a CYBER 200 Job (Sheet 2 of 3)

```
  12.17.16  STOP
  12.17.16   ALL DONE
  12.17.16  SYSTEM TIME UNITS (STU)                                  3.565
  12.17.16  USER CPU TIME (SECS)                                      .809
  12.17.16  SYSTEM CPU TIME SECS)                                    1.115
  12.17.16  USER MEMORY USAGE (PAGE*SECS)                           91.135
  12.17.16  USER AVERAGE WORKING SET SIZE (PAGES)                   112
  12.17.16  NUMBER OF VIRTUAL SYSTEM REQUESTS                       382
  12.17.16  NUMBER OF SMALL PAGE FAULTS                             36
  12.17.16  NUMBER OF DISK I/O REQUESTS                            20
  12.17.16  NUMBER OF DISK SECTORS TRANSFERRED                      37
  12.17.16  $$COMPLETE$$
END OF FILE
?? l-the sum is-2 <------------------------------------ Locates the program output.
    00008      200 FORMAT(' THE SUM IS')
                                        0001/00008
--EOR--
--EOR--
 THE SUM IS
?? p2 <------------------------------------------------ Displays the program output.
 THE SUM IS
 15
?? end
AALD IS A LOCAL FILE.
/route,aald,dc=lp <------------------------------------ Routes the output file to a
 ROUTE COMPLETE.  JSN IS AALF.                          printer.
/bye <-------------------------------------------------- Requests IAF logout.
UN=13428AA  LOG OFF   12.32.42
JSN=AAKM    SRU-S=        1.951
CHARACTERS=    8.181KCHS
IAF     CONNECT TIME 00.14.11. <------------------------- IAF logout.
LOGGED OUT.
```

Figure 13-1. Example of a NOS 2 Interactive Session
Submitting a CYBER 200 Job (Sheet 3 of 3)

Because no input file is specified on the FTN200 statement, the program to be compiled is read from the INPUT file (the job file).

The LOAD statement generates an executable program on the default file GO, and the GO statement executes the program on that file. The executable program reads its input data from the INPUT record, which is now the third record (the record containing the sequence of integers 1,2,3,4,5).

Figure 13-2 shows an example using the GO parameter on the FORTRAN control statement. With the GO parameter no LOAD and executing of controlee statements is needed.

The NOS 2 command SUBMIT,JOBFILE,E submits the CYBER 200 job file for execution. The NOS 2 command ENQUIRE,JSN displays the status of the user's jobs. After the job output file is returned to the NOS 2 print queue, the NOS 2 command, QGET,ABGK,PR changes the output file to a local file. The user locates and displays the dayfile showing that the job executed normally. He then locates and displays the program output. Finally, he routes the entire output file to a printer and logs out.

CYBER 200 INTERACTIVE SESSION

To begin a CYBER 200 interactive session, you first login to the front-end system, then connect to the CYBER 200 system, and finally enter a LOGON statement to login to the CYBER 200 system. Ask site personnel for the entries required to connect to the CYBER 200 system from your front-end system. Figure 13-3 shows a CYBER 200 interactive session within a NOS interactive session.

To execute a CYBER 200 control statement within an interactive session, you enter the statement and then wait for the ALL DONE response indicating the statement processing has finished.

OPERATING SYSTEM INTERFACE

A FORTRAN program can interact with the operating system while it is executing. This is accomplished by using the System Interface Language (SIL). Various debugging facilities can also be accessed by a FORTRAN program while it is executing.

The following paragraphs describe SIL and the debugging facilities. See the Operating System reference manual for more detailed information.

SYSTEM INTERFACE LANGUAGE

System Interface Language (SIL) is a set of subroutines that can be called from a FORTRAN program to exchange information with the operating system. (The SIL routines can also be called from CYBER 200 assembly language programs and Implementation Language (IMPL) programs.) The two types of SIL subroutines are non-input/output SIL subroutines and input/output SIL subroutines.

Some of the functions that the non-input/output SIL subroutines perform are:

- Informing the operating system of the system requirements of the program
- Communicating with the system operator
- Determining the task processing if an error occurs

Some of the functions that the input/output SIL subroutines perform are:

- Accessing permanent, local, pool, and tape files
- Creating public files
- Manipulating the file information table (FIT)
- Opening and closing files to prepare for input/output
- Performing input/output
- Positioning files

See figure 13-4 for the general format of a SIL call. See the operating system reference manual for a complete list of all of the SIL subroutines.

DEBUGGING UTILITIES

Program debugging is assisted by compilation diagnostics, execution diagnostics, and program abort dumps. A program abort dump contains information from the drop file, including a subroutine traceback. (The dump content is described in the VSOS Reference Manual, volume 1.)

Three debugging utilities are provided to aid you in debugging a FORTRAN program:

- The DEBUG utility enables you to specify places in the FORTRAN program where you want to suspend execution temporarily and display or alter the contents of selected locations.
- The LOOK utility enables you to examine the contents of selected locations of any type of file, such as a data file or controllee file.
- The DUMP utility displays the contents of selected portions of a drop file.

Each of these debugging utilities is described in detail in the operating system reference manual.

```
(date)   (time)
(NOS 2 logon banner)
FAMILY: ,user1,passwd <-------------------- Entry to logon to NOS 2.
JSN: ABGV,NAMIAF <------------------------- IAF logon is successful.

 READY.
normal <----------------------------------- Ensures the terminal is
 READY.                                     in normal mode.
batch <------------------------------------ Switches NOS 2 to batch mode.
RFL,0.
/get,job2 <-------------------------------- Gets the file JOB1.
/xedit,job2 <------------------------------ Calls a text editor to
 XEDIT 3.1.00                               display the job file
?? p*                                       contents.
/JOB
/NOSEQ
JOB2,STTY3.
/READ,ACCT205
COMMENT.  JOB EXAMPLE TO DEMONSTRATE COMPILE AND GO
COMMENT.  FILE ACCT205 ON THE NOS SYSTEM CONTAINS THE
COMMENT.  CY205 ACCOUNTING INFORMATION.  AN EXAMPLE
COMMENT.  OF THE CONTENT OF THIS FILE MIGHT BE
COMMENT.
COMMENT.  USER(U=205_USERNUM,PA=205_PASSWORD,AC=205_ACCOUNT)
COMMENT.  RESOURCE(TL=10,JCAT=JDEFAULT)
COMMENT.
FTN200,GO. <------------------------------- Calls the compiler with
/EOR                                        with GO option.  Note
      PROGRAM LOOP                          that no LOAD and
      K=0                                   executing of controlee
      DO 10 I = 1,5                         statements are needed.
      READ 100,J
  100 FORMAT(I1) <--------------------------- FORTRAN source code.
   10 K = J + K
      PRINT 200
  200 FORMAT(' THE SUM IS')
      PRINT 300,K
  300 FORMAT(1X,I2)
      STOP
      END
/EOR
1 <---------------------------------------- Input data to the FORTRAN
2                                           program.
3
4
5
/EOF
END OF FILE
```

Figure 13-2. Example of a NOS 2 Interactive Session
Submitting a CYBER 200 Job With the GO Option (Sheet 1 of 3)

```
?? end
JOB2 IS A LOCAL FILE
/submit,job2,e <------------------------------------------- Submits job to be
 12.34.00  SUBMIT COMPLETE.  JSN IS AALI.                   executed on 205.
/enquire,jsn <------------------------------------------- Displays job status.

 JSN SC CS DS LID STATUS          JSN SC CS DS LID STATUS

 AALI.B.  .RB.TY3.INPUT QUEUE     AALG.T.ON.BC.   .EXECUTING
/enquire,jsn

 JSN SC CS DS LID STATUS          JSN SC CS DS LID STATUS

 AALG.T.ON.BC.   EXECUTING
/enquire,jsn

 AALG.T.ON.BC.   .EXECUTING       AALK.R.  .RB.M10.PRINT QUEUE
/qget,aalk,pr <------------------------------------------- Output file becomes local
 QGET COMPLETE.                                            file.
/xedit,aalk <------------------------------------------- Calls a text editor to
 XEDIT 3.1.00                                              display the output file.
?? l-rsys- <------------------------------------------- Locates the dayfile.
--EOR--
--EOR--
1 12.22.10 RSYSK11   VS22DBC   013428   DEVSYS   G   JOB2AABF
   08/28/85
?? p* <------------------------------------------- Lists the dayfile.
1 12.22.10 RSYSK11   VS22DBC   013428   DEVSYS   G   JOB2AABF
  08/28/85
  12.22.11  RESOURCE,TL=10.
  12.22.11  COMMENT.  JOB EXAMPLE TO DEMONSTRATE COMPILE AND GO
  12.22.11  COMMENT.  FILE ACCT205 ON THE NOS SYSTEM CONTAINS THE
  12.22.11  COMMENT.  CY205 ACCOUNTING INFORMATION.  AN EXAMPLE
  12.22.11  COMMENT.  OF THE CONTENT OF THIS FILE MIGHT BE
  12.22.11  COMMENT.
  12.22.11  COMMENT.  USER(U=205_USERNUM,PA=205_PASSWORD,AC=205_ACCOUNT)
  12.22.11  COMMENT.  RESOURCE(TL=10,JCAT=JDEFAULT)
  12.22.11  COMMENT.
  12.22.11  FTN200,GO.
  12.22.11  FORTRAN 200 CYCLE K10      BUILT 08/17/85 18:00
  12.22.12  COMPILING LOOP
  12.22.13        NO ERRORS
  12.22.16  STOP
  12.22.16   ALL DONE
  12.22.16  SYSTEM TIME UNITS (STU)                          3.565
  12.22.16  USER CPU TIME (SECS)                              .809
  12.22.16  SYSTEM CPU TIME SECS)                            1.115
  12.22.16  USER MEMORY USAGE (PAGE*SECS)                   91.135
  12.22.16  USER AVERAGE WORKING SET SIZE (PAGES)           112
  12.22.16  NUMBER OF VIRTUAL SYSTEM REQUESTS               382
  12.22.16  NUMBER OF SMALL PAGE FAULTS                      36
  12.22.16  NUMBER OF DISK I/O REQUESTS                      20
  12.22.16  NUMBER OF DISK SECTORS TRANSFERRED               37
  12.22.16  $$COMPLETE$$
END OF FILE
```

Figure 13-2. Example of a NOS 2 Interactive Session
Submitting a CYBER 200 Job With the GO Option (Sheet 2 of 3)

```
?? l-the sum is-2                                          Locates the program
    00008      200 FORMAT(' THE SUM IS')                   output.
                                    0001/00008
--EOR--
 THE SUM IS
?? p2                                                      Displays the program
 THE SUM IS                                                output.
 15
?? end
AALK IS A LOCAL FILE.
/route,aalk,dc=lp                                          Routes the output file
 ROUTE COMPLETE.  JSN IS AALL.                             to a printer.
/bye                                                       Requests IAF logout.
UN=13428AA  LOG OFF   12.42.42
JSN=AAKM    SRU-S=        2.302
CHARACTERS=    4,706KCHS
IAF     CONNECT TIME 00.05.23.                             IAF logout.
LOGGED OUT.
```

Figure 13-2. Example of a NOS 2 Interactive Session
Submitting a CYBER 200 Job With the GO Option (Sheet 3 of 3)

```
(date)    (time)
(NOS 2 logon banner)
FAMILY:  ,user1,xpwxpwx                                    NOS 2 logon.
JSN: ABGU,NAMIAF                                           IAF logon is successful.

 READY.
hello,itf                                                  Requests ITF connection.

UN=USER1  LOG OFF    11.57.30.
JSN=ABGU  SRU-S         1.000.
IAF     CONNECT TIME 00.00.42.                             Leaves IAF application and
ITF 1.0 - 596                                              enters ITF application.
Terminal T1398, connection 7
Enter LID (or ?): abc                                      Enters valid logical id.
[ITF, connecting to host ABC.]
 PLEASE ENTER CY200 LOGON                                  Successful link to VSOS.
logon,user1,passwd                                         Valid CYBER 200 logon line.
OS 2.1    RSYSL592   VSYSL592   G 09998.7564 ACTIVE NONE   Banner for CYBER 200
NIL                                                        operating system.
$bye                                                       CYBER 200 logout request.
BYE                                                        Logout response.
ITF     CONNECT TIME:  00.02.52.                           Leaves ITF and returns to
JSN: ABJH, NAMIAF                                          IAF.
 READY.
bye                                                        Request to logout from IAF.

UN=USER1  LOG OFF    13.26.51.
JSN=ABJH  SRU-S         1.000.

IAF     CONNECT TIME:  00.02.28.                           IAF logout.
LOGGED OUT.
```

Figure 13-3. Example of a CYBER 200 Interactive Logon and Logout From a NOS 2 Front-End System

```
CALL Q5xxxxxx(p1, ... ,pn)

xxxxxx    The name of a SIL routine
pi        A SIL parameter
```

Figure 13-4. SIL Call Format

SUBPROGRAM LINKAGE

The following paragraphs describe the subprogram linkage conventions used by the FORTRAN compiler. The subprogram linkage conventions include:

The prologue and epilogue generated by the FORTRAN compiler for subroutines and functions

The standard calling sequence used for subroutine calls and function references

The fast calling sequence used for some subroutine calls and function references

The file initialization performed before program execution in order to enable input/output operations

PROLOGUE AND EPILOGUE

The FORTRAN compiler generates a prologue and epilogue for subroutine calls and function references. The prologue and epilogue depend on whether the subroutine or function is a non-zero-swap routine or a zero-swap routine.

A non-zero-swap routine is a routine that requires the values currently in the registers to be saved upon entering the routine. For a non-zero-swap routine, the prologue code performs the following functions:

1. Saves the values in registers #1A through #1F
2. Saves the values of the calling routine's registers and loads the registers with the values they had during the most recent execution of the called subprogram (this involves a swap instruction that destroys the contents of registers #1A through #1F)
3. Restores the values in registers #1A (return address), #1E (data base address), and #1F (Data Flag Branch Manager table pointer)
4. Updates the values in registers #1B (dynamic stack address), #1C (current stack pointer), and #1D (previous stack pointer)
5. Clears the length field of register #1F for the Data Flag Branch Manager

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

1. Saves the values of the registers that were assigned by the called routine and loads into the registers the values they had before the subprogram was called
2. If the length field of register #1F is nonzero, restores the condition-enable bits of the data flag branch register to the values they had before the subprogram was called, preserving the free data flags
3. Returns control to the address specified in register #1A

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

Compilation option O or Z is specified in the FTN200 control statement

There are no calls or function references (other than to FORTRAN routines that can be generated inline)

There are no input/output statements

No vector programming features are used

The generated code can be reasonably executed using only registers #3 through #13, and possibly registers #17, #18, and #19

No special call statements are used

STANDARD CALLING SEQUENCE

In general, the FORTRAN compiler uses the subprogram linkage conventions described in volume 2 of the operating system reference manual. When a subroutine or function that you have written calls a subroutine or references an external function, such as one written in assembly language, the standard calling sequence in machine language is essentially as shown in figure 13-5.

```
#78xx001E     Load register #1E with the address
              xx of the callee data base.

#78yy0017     Load register #17 with the
              parameter list descriptor yy.

#361A00zz     Branch to entry point zz of the
              called procedure after setting a
              return location in register #1A.
```

Figure 13-5. Standard Calling Sequence

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

Register #17 contains a descriptor of the argument vector. The length portion of register #17 contains the number of fullwords in the argument vector. The address portion contains the address of the first word of the argument vector.

Each word in the argument vector corresponds to an actual argument in the external procedure reference. The length portion of each word is zero unless the argument is of type character; if the argument is of type character, the length portion contains the length of the character variable or character array element. The address portion of each word contains the address of the actual argument.

At the time a subprogram reference is executed, each variable listed as a dummy argument is associated with the same storage location as the actual argument corresponding to it (call by address). Each definition of a dummy argument can change the value in that storage location. Thus, when control returns to the referencing program unit, the values of the actual arguments can be different from what they were before the subprogram reference.

For function references, halfword function results are returned in the high-order half of register #18, fullword function results are returned in register #18, and doubleword function results are returned in registers #18 and #19. Halfword results are returned for half-precision functions; fullword results are returned for integer, real, and logical functions; doubleword results are returned for double-precision and complex functions.

Vector functions return results in the result vector, which is passed in the parameter list.

FAST CALLS

Many intrinsic functions have a fast call entry point as well as a standard call entry point. The compiler generates a fast call to these functions unless the function name appears in an EXTERNAL statement in the calling program. The standard call entry point is the function name. See section 10 for a list of the fast call entry point names. (Not all intrinsic functions have fast call entry points). Fast calls are not generated for subprograms that you write.

The difference between the standard and fast calling sequences is the method by which parameters are passed to the called subprogram. For the standard calling sequence, parameters are passed using a parameter list in memory; for the fast calling sequence, parameters are passed using registers #3 through #6 as required by the number and length of the parameters. Scalar results are returned for functions called with the standard calling sequence. Vector functions return results in the result vector; register #18 is automatically preset by the caller to the descriptor of the result vector.

See figure 13-6 for an example of a fast call to a scalar function having one argument.

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

Figure 13-6. Fast Calling Sequence Example

In the instructions in the figure, xx contains the callee data base address, yy contains the function parameter, and zz contains the function entry point address. Function parameters must be loaded in consecutive registers beginning with register #3 and in the order specified in the function descriptions; see section 10 for the function descriptions.

All of the other global and environment registers are initialized by the operating system.

This section describes the FORTRAN control statement, its parameters, and the kinds of output information that it can direct the FORTRAN 200 compiler to produce.

The FTN200 control statement calls the FORTRAN 200 compiler, specifies the files for input and output, and determines the kind of output produced. The following forms can be used to invoke the compiler:

```
FTN200.
FTN200,olist.
FTN200 olist.
FTN200(olist)
```

where olist is an optional list of parameters in any order separated by commas or blanks. FTN200. executes the compiler with all options being default.

Each parameter consists of a keyword and an optional value. For example:

```
LIST=lfn
```

where LIST is the keyword and lfn is an optional local file name.

ABBREVIATION

Some keywords can be abbreviated. For single word keywords, the abbreviation can be from one to all of the leftmost consecutive letters of the word. The following are acceptable abbreviations of the keyword BINARY:

```
B
BI
BIN
BINA
BINAR
BINARY
```

DEFAULTS

There are two kinds of parameter defaults. The first kind occurs when the parameter and its value are both omitted. The second kind occurs when the parameter is given but the value is omitted.

The keywords, their minimal abbreviations, and their defaults are summarized in table 14-1 and discussed in detail in the following paragraphs. Appendix F contains the appropriate information for release VSOS 2.1.6.

KEYWORDS

Keywords can be either binary or multiple-option. A binary keyword either does or does not select its option. For example, you can select 64 bit comparison either by selecting C64 or by selecting C64=1. You can get the alternative (48 bit comparison) either by selecting nothing or by selecting C64=0. The multiple-option keywords, on the other hand, have several options in addition to the default. For example, OPTIMIZE has five options in addition to the default.

TABLE 14-1. KEYWORD ABBREVIATIONS AND PARAMETER DEFAULTS

Keyword	Minimal Abbreviation	First Default (Omit Keyword and Value)	Second Default (Omit Value Only)
ABC	ABC	ABC=0	ABC=1
ANSI	ANSI	ANSI=0	ANSI=W
BINARY	B	B=BINARY/16	B=BINARY/16
C64	C64	C64=0	C64=1
DO	DO	DO=0	DO=1
ERRORS	E	E=OUTPUT/16	E=ERRS/16
ELEV	ELEV	ELEV=W	ELEV=F
F66	F66	F66=0	F66=1
GO	GO	GO=0	GO=1
INPUT	I	I=INPUT	I=COMPILE
LIST	L	L=OUTPUT/336	L=LIST/336
LO	LO	LO=S	LO=SX
OPTIMIZE	OPT	OPT=0	OPT=1
SC	SC	SC=0	SC=1
SDEB	SDEB	SDEB=0	SDEB=1
SYNTAX	SYN	SYN=0	SYN=1
TM	TM	TM=HOST	TM=205
UNSAFE	UNS	UNS=0	UNS=1

KEYWORDS AND THEIR OPTIONS

Following is a list of the FORTRAN 200 control statement parameters and their options. They are listed alphabetically by keyword.

ABC

Instructs the compiler to perform array bound checking on all executable statements and to generate code that will detect it during execution. This option should not be turned on in a production code, because it may affect the performance of execution.

ABC=1

Set array bound checking on.

ABC=0

Set array bound checking off.

ABC

Same as ABC=1.

omit

Same as ABC=0.

ANSI

Specifies whether and how severely to diagnose non-ANSI extensions. The valid options are:

ANSI=F

Treats non-ANSI uses as fatal errors. All ANSI messages become fatal errors.

ANSI=W

Treats non-ANSI uses as warnings.

ANSI=0

Generates no ANSI diagnostics.

ANSI

Same as ANSI=W.

omit

Same as ANSI=0.

BINARY

Specifies the name of the file to which the compiler writes the binary object code. See appendix F for a description of the BINARY parameter in release VSOS 2.1.6.

The disposition of the binary file depends on the kind of file. If the file is an attached permanent file, then it is used, and any explicit or implied length specification is ignored. If the file is an existing local file, then the compiler returns it and creates a new file. The compiler creates a new local file in all other cases. The compiler performs these functions by calling Q5GETFIL with the RETURN parameter specified as described in the VSOS reference manual, Volume 1.

With VSOS Release 2.2, you do not need to specify file length because the system allocates as many blocks as necessary for the file to hold the binary object code. If you do use the length option, the first space allocation for the file is the length you specify.

The valid options are:

BINARY=lfn/len

Writes object code on the file lfn, initially setting the length at len blocks. The option len can be an integer constant or a hexadecimal constant prefixed with a #. A block is 512 consecutive words. The len default is 16 blocks. The compiler passes len as the file length in its call to Q5GETFIL.

BINARY=lfn

Same as BINARY=lfn/16.

BINARY

Same as BINARY=BINARY/16.

omit

Same as BINARY=BINARY/16.

BINARY=0

Generates no object code.

The BINARY parameter might conflict with the GO parameter. If BINARY is set to generate object code, the only acceptable option for the GO parameter is 0. Any other options generate warnings.

C64

Instructs the compiler to generate code that will compare all 64 bits of an integer for .EQ. and .NE. operators in the logical IF statement. The valid options are:

C64=1

Generates code to compare 64 bits.

C64=0

Generates code to compare 48 bits for integer arithmetic.

C64

Same as C64=1.

omit

Same as C64=0.

DO

Specifies how the compiler is to interpret DO loops. The option of DO=1 generates more efficient code for DO loops in some cases. On the other hand, the program will not execute correctly if DO=1 and a DO loop has a zero iteration count. The valid options are:

DO=1

Minimum iteration count is one.

DO=0

Minimum iteration count is zero.

DO

Same as DO=1.

omit

Same as DO=1 if F66 is specified, otherwise same as DO=0.

ERRORS

Specifies a file name for recording the error information. See appendix F for a description of the ERRORS parameter in release VSOS 2.1.6.

The file length that you specify with ERRORS is not necessarily always what you get; there is one exception. If you have specified the same file name with LIST, the actual file length will become the larger of the two. If your LIST specifies a larger file than your ERRORS specification, the LIST file size prevails.

The disposition of the errors file depends on the kind of file. If the file is an attached permanent file, then it is used, and any explicit or implied length specification is ignored. If the file is an existing local file, then the compiler returns it and creates a new file. The compiler creates a new local file in all other cases. The compiler performs these functions by calling Q5GETFIL with the RETURN parameter specified.

The valid options are:

ERRORS=lfn/len

Writes error diagnostics on the file lfn, initially setting the length at len blocks if there is an error of at least ELEV severity. The option len can be an integer constant or a hexadecimal constant prefixed with a #. A block has 512 words. The len default is 16 blocks. The compiler passes len as the file length in its call to Q5GETFIL.

With VSOS Release 2.2, you do not need to specify the file length because the system allocates as many blocks as necessary for the file to hold the error diagnostics. If you do use the length option, the first initial allocation for the file is 16 blocks.

If the error diagnostics take up more than 16 blocks of space, the system increases the length of the lfn file as needed to hold all the error diagnostics.

ERRORS=lfn

Same as ERRORS=lfn/16.

ERRORS

Same as ERRORS=ERRS/16.

omit

Same as ERRORS=OUTPUT/16.

ELEV

Specifies the error severity threshold for writing the error diagnostics on the ERRORS file. Error severity levels are ordered by increasing severity. Specification of a level selects the level and all levels above. The valid options are:

ELEV=W

Sets the write threshold at the warning level. The warning level signifies a syntax error. The system writes error diagnostics and the compilation continues. Also writes all fatal and catastrophic errors.

ELEV=F

Sets the write threshold at the fatal level. A fatal error prevents compiliation of the faulty statement and writes error diagnostics for the fatal level. The compiler generates no binary file. Also writes all catastrophic errors.

ELEV=C

Sets the write threshold at the catastrophic level. A catastrophic error stops the compiler with no further processing.

ELEV

Same as ELEV=F.

omit

Same as ELEV=W.

F66

Instructs the compiler whether the source file, as specified in the INPUT parameter, contains the ANSI X3.9-1966 FORTRAN based dialect of FORTRAN 200. When you specify F66, then ANSI=0 and DO=1 are assumed. If you have specified other values, warnings are issued. The valid options are:

F66=1

Source file contains the 1966 FORTRAN dialect.

F66=0

Source file contains the 1978 FORTRAN 200 dialect.

F66

Same as F66=1.

omit

Same F66=0.

GO

If the program compiles with no fatal errors, the GO parameter indicates that the program is to be executed; this means the user does not need to specify the parameters LOAD and GO as in VSOS release 2.1.6. With the GO parameter specified, the FTN200 statement compiles and executes the program. The GO parameter works only in those installations which have the system shared library feature active. The valid options are as follows:

GO

Execute the user program upon completion of a nonfatal compile.

GO=0

Only compile the user program.

GO=1

Same as GO.

Omit

Same as GO=0.

With the GO parameter, your program executes using the system shared library and a dynamic LINKER utility that loads dynamic modules and gives control to the called module. The system shared library is a file that contains the LINKER utility, directories, shared utilities, and a shared SYSLIB.

For batch jobs, the GO task reads and writes files as specified in your FORTRAN 200 source program. On the other hand, interactive sessions require that you create two files before beginning the compilation of your program: INPUT and Q5INPUT. INPUT holds your source code, which is input to the FTN200 compiler, and Q5INPUT contains the input data for the compiled program. Unless you specify other input files, you must create the INPUT and Q5INPUT file before beginning the compilation of your program. Additionally, interactive compile and GO creates two output files: Q5OUTPUT for the compiler listing and OUTPUT for the output of your executing program.

The GO parameter might conflict with the BINARY and SYNTAX parameters. For the BINARY parameter, if you specify GO=1, then you must specify BINARY=0; otherwise, error messages are returned. For the SYNTAX parameter, if you specify GO=1, then you must either omit the SYNTAX parameter or set SYNTAX=0; otherwise, error messages are returned.

If the FORTRAN 200 compiler is statically loaded (loaded with LINK=M) you must not specify the GO parameter; otherwise, an error message is returned.

INPUT

Specifies the name of the input source code file for the compiler. The valid options are:

INPUT=lfn

Name of the file is lfn.

INPUT

Same as INPUT=COMPILE.

omit

Same as INPUT=INPUT.

The maximum record length for the input file is 96 characters.

LIST

Specifies the file name to which the compiler can write the the source listing and other requested information except diagnostics. See appendix F for a description of the LIST parameter in release VSOS 2.1.6.

The file length that you specify with LIST is not necessarily always what you get; there is one exception. If you have specified the same file name with ERRORS, the actual file length will become the larger of the two. If your ERRORS specifies a larger file than your LIST specification, the ERRORS file size prevails.

The disposition of the list file depends on the kind of file. If the file is an attached permanent file, then it is used, and any explicit or implied length specification is ignored. If the file is an existing local file, then the compiler returns it and creates a new file. The compiler creates a new local file in all other cases. The compiler passes len as the file length in its call to Q5GETFIL.

With VSOS Release 2.2, you do not need to specify the file length, because the system allocates as many blocks as necessary for the file to hold the listing. If you do use the length option, the first space allocation for the file is 336 blocks.

The valid options are:

LIST=lfn/len

Writes listing on the file lfn, initially setting the length at len blocks. The option len can be an integer constant or a hexadecimal constant prefixed with a #. A block is 512 consecutive words. The len default is 336 blocks. LIST performs these functions by calling Q5GETFIL with the RETURN parameter specified. If the listing takes up more than 336 blocks of space, the system increases the length of the file as needed to hold the entire listing.

LIST=lfn

Same as LIST=lfn/336.

LIST

Same as LIST=LIST/336.

omit

Same as LIST=OUTPUT/336.

LIST=0

Supresses all listing except that directed to the ERRORS file.

LO

Specifies what information is to appear on any listing file generated with the LIST parameter. You can specify multiple options by concatenating the option letters. The valid options are:

LO=[op][op]...

The option op can specify any of the following to be written to the listing file:

- A Assembly listing of object code
- M Map of register file and storage assignments
- S Source code
- X Cross-reference map
- I Index map

LO

Same as LO=SX.

omit

Same as LO=S.

NOTES

The LO parameter might conflict with the SYNTAX parameter.

If the I option is specified, the S option must also be specified (for example, LO=SI is valid).

OPTIMIZE

Specifies to the compiler whether to optimize scalar code and, if so, which optimizations to perform. Performs mainframe dependent optimizations for the target mainframe as specified in the TM parameter. OPTIMIZE produces more efficient code at the expense of increased compilation time. As with the LO parameter, multiple options can be specified by concatenating the options.

OPTIMIZE=[op][op]...

The option op can be any of the following:

- D Optimize DO loops.
- P Propagate compile-time computable results.
- R Remove redundant code.
- S Schedule instructions.
- V Vectorize certain types of DO loops and transform other types into STACKLIB routines. See section 9 for more information.

OPTIMIZE=1

Same as OPTIMIZE=DPRSV.

OPTIMIZE=0

No optimization is performed.

OPTIMIZE

Same as OPTIMIZE=1.

omit

Same as OPTIMIZE=0

NOTE

The OPTIMIZE parameter might conflict with the SYNTAX parameter.

SC

Specifies how the compiler interprets the reserved names for the FORTRAN 200 special calls. The valid options are:

SC=1

Interprets names as machine instruction references.

SC=0

Interprets names as user-supplied subroutine references.

SC

Same as SC=1.

omit

Same as SC=0.

SDEB

Specifies whether to suppress the DEBUG symbol tables. The DEBUG utility needs the tables to reference FORTRAN 200 variables and line numbers. The run time error processor needs them for subroutine tracebacks. The BINARY and controllee files are shorter without the tables. The valid options are:

SDEB=1

Supresses the DEBUG symbol tables.

SDEB=0

Generates the DEBUG symbol tables.

SDEB

Same as SDEB=1.

omit

Same as SDEB=0.

SYNTAX

Instructs the compiler to perform a quick syntax check on the source program. See appendix F for a description of the SYNTAX parameter in release VSOS 2.1.6. The valid options are:

SYNTAX=1

Performs a full syntactic scan but generates no BINARY file.

SYNTAX=0

Compiles completely.

SYNTAX

Same as SYNTAX=1.

omit

Same as SYNTAX=0.

NOTE

The SYNTAX parameter might conflict with the BINARY, LO, GO, OPTIMIZE, and UNSAFE parameters. When SYNTAX=1, the only acceptable options for these parameters are BINARY=0, OPTIMIZE=0, UNSAFE=0, LO=S, LO=X, LO=SI, LO=SX, and LO=SIX.

Any other options generate warnings.

TM

Specifies the target mainframe to execute the generated object code The selections are mutually exclusive. The default is the host mainframe: the mainframe that compiled the program. The valid options are:

TM=n

The option n can be any of the following target mainframe codes:

205 CYBER 205

HOST Host mainframe

TM

Same as TM = 205

omit

Same as TM = HOST

UNSAFE

Instructs and permits the compiler to perform certain optimizations that might be unsafe. For example, if the end value of a DO loop is variable and if the loop contains dummy array references, the compiler cannot determine the number of iterations of the loop. Vectorization, therefore, might be unsafe because the loop count could exceed 65535, the maximum length for a vector. See section 9 for more information on the UNSAFE parameter. The valid options are:

UNSAFE=1

Allows unsafe optimization.

UNSAFE=0

Disallows unsafe optimization.

UNSAFE

Same as UNSAFE=1.

omit

Same as UNSAFE=0.

NOTE

The UNSAFE parameter might conflict with the SYNTAX parameter.

CONTROL STATEMENT EXAMPLES

Figures 14-1 and 14-2 show examples of the use of FORTRAN 200 control statements. Appendix F contains the appropriate information in figures 14-1 and 14-2 for release VSOS 2.1.6.

Figure 14-1 shows the all-default case. The FTN200 statement alone assigns default values to each option. The figure lists the default values.

The ERRORS file of figure 14-1 has a default length of 16, but the ERRORS file is the same file as the LIST file, and the greater length takes precedence.

FTN200. is equivalent to:

```
FTN200,ANSI=0,
    BINARY=BINARY/16,
    C64=0,
    DO=0,
    ERRORS=OUTPUT/336,
    ELEV=W,
    F66=0,
    GO=0,
    INPUT=INPUT,
    LIST=OUTPUT/336,
    LO=S,
    OPTIMIZE=0,
    SC=0,
    SDEB=0,
    SYNTAX=0,
    TM=HOST,
    UNSAFE=0.
```

Figure 14-1. Control Statement Example With Default Values

Figure 14-2 shows an example of a FORTRAN 200 control statement with some options specified and others allowed to default. The list shows all values, including the default values.

Although the ERRORS file of figure 14-2 has been specified with a length of 16, the LIST file is the same file and its default length is 336; the length of 336 takes precedence.

```
FTN200,I=SOURCE,L=LOOK,OPT=1,LO=AS,SC,TM=205,
       B=LGO/#AA,C64,E=LOOK/16


is equivalent to (including defaults):

FTN200,ANSI=0,
    BINARY=LGO/#AA,
    C64=1,
    DO=0,
    ERRORS=LOOK/336,
    ELEV=W,
    F66=0,
    GO=0,
    INPUT=SOURCE,
    LIST=LOOK/336,
    LO=AS,
    OPTIMIZE=DPRSV,
    SC=1,
    SYNTAX=0,
    TM=205,
    UNSAFE=0.
```

Figure 14-2. Control Statement Example

COMPILER-GENERATED LISTINGS

The FORTRAN 200 compiler can place a variety of information in the source listing file. The LO compilation options A, M, S, I, and X control that information placement.

A header line at the top of each page of the source listing contains the compiler version, the type of listing, the time, the date, and the page number.

The source program listing (including comments) is the first item to be placed on the file. It has 58 lines per printed page, excluding headers. The output lines are numbered on the right and the source lines on the left. The cross-reference maps, to be discussed later, use the source line numbers.

Any appropriate diagnostics appear at the end of each program unit. If you selected no compilation options but there were syntax errors, any diagnostics would appear immediately after the source listing. If, instead, your syntax were acceptable and there were no diagnostics, the message NO ERRORS would appear after the source listing. Each error diagnostic indicates the source line number of the error, the error number, and the error severity level. See appendix B for a summary of the diagnostics.

The following listings appear after the source program:

Cross-reference maps

Assembly listing

Storage map and register map

Index map

Any generated diagnostics follow the storage and register maps.

CROSS-REFERENCE MAPS

If you select LO=X, then one to four cross-reference maps will appear in the source listing. The maps appear immediately following the source program. The four cross-reference maps are:

Statement label map

Variable map

Symbolic constant map

Procedure map

Statement Label Map

The statement label map provides information about each statement label used in the program. See figure 14-3 for a statement label map format and figure 14-4 for a statement label map example. The statement label map will not print if the program has no statement labels. Uses of the statement label map include:

Finding unreferenced FORMAT statements and other unreferenced but labeled statements

Verifying that flow control statements have proper statement labels

Locating labeled statements and their references

```
STATEMENT LABEL MAP
--LABEL---DEFINED---ADDRESS---REFERENCES

   lbl      def       addr      refs
   .        .         .         .
   .        .         .         .
   .        .         .         .


lbl     A statement label that appears in the
        label field of a FORTRAN statement

def     The source line number of the statement
        in which lbl appears in the label field

addr    Bit address of the label relative to
        code section; suppressed for format
        labels or if no object was generated.

refs    The source line numbers of all source
        lines that contain references to lbl
```

Figure 14-3. Statement Label Map Format

```
FORTRAN 200 CYCLE 14       BUILT 11/22/82 20:38   SOURCE LISTING       PASCAL    COMPILED 12/07/82 14:22

    00001           PROGRAM PASCAL(OUTPUT)
    00002           INTEGER L(11),ONE,A,B
    00003           PARAMETER (ONE=1)
    00004           IADD(A,B) = A + B
    00005           DATA L(11) /ONE/
    00006           PRINT 4,(I,I=1,11)
    00007   4       FORMAT('1COMBINATIONS OF M THINGS TAKEN N AT ',
                   X  'A TIME.'//20X,'-N-' /11I5)
    00008           DO 200 I=1,10
    00009           K = 11 - I
    00010           L(K) = 1
    00011           DO 100 J = K,10
    00012   100     L(J) = IADD(L(J),L(J+1))
    00013   200     PRINT 3, (L(J),J=K,11)
    00014   3       FORMAT(11I5)
    00015           STOP
    00016           END


FORTRAN 200 CYCLE 14       BUILT 11/22/82 20:38   CROSS REF LISTING    PASCAL    COMPILED 12/07/82 14:22

 STATEMENT LABEL MAP
 --LABEL---DEFINED---REFERENCES

   100         12          11
   200         13           8
   3           14          13
   4            7           6

 VARIABLE MAP
 --NAME-------BLOCK------TYPE------CLASS-------REFERENCES     A=ARGLIST, C=CTRL OF DO, I=DATA INIT, R=READ, S=STORE, W=WRITE

   A                     INTEGER   SIMPLE          2       4       4
   B                     INTEGER   SIMPLE          2       4       4
   I                     INTEGER   SIMPLE          6       6       8       9
   J                     INTEGER   SIMPLE         11      12      12      12      13      13
   K                     INTEGER   SIMPLE          9/S    10      11      13
   L                     INTEGER   ARRAY           2       5/I    10/S    12/S    12      12      13
   PASCAL                          PROGRAM         1

 SYMBOLIC CONSTANT MAP
 --NAME-------TYPE-------VALUE-----------------------------------REFERENCES     S=DEFINITION LINE

   ONE         INTEGER    1                                          2       3/S

 PROCEDURE MAP
 --NAME-------TYPE-------CLASS------------REFERENCES     D=STMT FN DEF, A=ARGLIST

   IADD        INTEGER    STAT FUNC             4/S    12
```

Figure 14-4. Source Listing Example (Sheet 1 of 5)

```
FORTRAN 200 CYCLE 14      BUILT 11/22/82 20:38   ASSEMBLY LISTING     PASCAL    COMPILED 12/07/82 14:22

LOCATION      MACHINE            LINE    SOURCE          ASSEMBLY REPRESENTATION
COUNTER       INSTRUCTION        NUMBER  LABEL

                                         PASCAL    IDENT
                                                   ENTRY       PASCAL

                                 00001   A00006

0000000       7D00151C                  PASCAL    SWAP        ,C_#1A,CUR_STACK
0000020       781C001D                            RTOR        CUR_STACK,PREV_STACK
0000040       781B001C                            RTOR        DYN_SPACE,CUR_STACK
0000060       3E360028                            ES          CG_36,40
0000080       30360637                            SHIFTI      CG_36,6,CG_37
00000A0       631B371B                            ADDX        DYN_SPACE,CG_37,DYN_SPACE
00000C0       7B361C1C                            PACK        CG_36,CUR_STACK,CUR_STACK
00000E0       3E381280                            ES          CG_38,4736
0000100       631E3839                            ADDX        CALLEDATA,CG_38,CG_39
0000120       2A390016                            ELEN        CG_39,22
0000140       7D391400                            SWAP        CG_39,C_#20
0000160       782F0003                            RTOR        L_C00001_DESCR,PR_3
0000180       3E040000                            ES          PR_4,0
00001A0       3E050000                            ES          PR_5,0
00001C0       BE3A000000000000                    EX          CG_3A,.EXTC.F_PROLOG
0000200       BE1E000000000000                    EX          CALLEDATA,.DB.F_PROLOG
0000240       361A003A                            BSAVE       RETURN,CG_3A
0000260       781B0022                            RTOR        DYN_SPACE,PI_DYNSP
0000280       3E3B0060                  00006     ES          CG_38,96
00002A0       7F203B16                            STO         [DATABASE,CG_3B],C_#1
00002C0       78320017                            RTOR        *ARG_VECT,C_PARM_DESCR
00002E0       78320003                            RTOR        *ARG_VECT,PR_3
0000300       2A030003                            ELEN        PR_3,3
0000320       3E3C0061                            ES          CG_3C,97
0000340       7F203C30                            STO         [DATABASE,CG_3C],L_F4_DESCR
0000360       78350017                            RTOR        *ARG_VECT,C_PARM_DESCR
                                                  •
                                                  •
                                                  •
0000920       361A0039                            BSAVE       RETURN,CG_39
0000940       3E3A0032                            ES          CG_3A,50
0000960       7E203A3B                            LOD         [DATABASE,CG_3A],CG_3B
0000980       B4063B2600242B13                    IBXLE,BRB   CG_3B,AL_26,D00002,AL_2B,PR_13
00009C0       3E3C0032                            ES          CG_3C,50
00009E0       7F203C13                            STO         [DATABASE,CG_3C],PR_13
0000A00       3E030000                  00015     ES          PR_3,0
0000A20       BE3D000000000000                    EX          CG_3D,.EXTC.F_EPILOG
0000A60       BE1E000000000000                    EX          CALLEDATA,.DB.F_EPILOG
0000AA0       361A003D                            BSAVE       RETURN,CG_3D
                                                  END
```

Figure 14-4. Source Listing Example (Sheet 2 of 5)

FORTRAN 200 CYCLE 14 BUILT 11/22/82 20:38 REGISTER MAP PASCAL COMPILED 12/07/82 14:22

FULLWORD REGISTER M

REG. NO	NAME	REG. NO	NAME	REG. NO	NAME	REG. NO	NAME	REG. NO	NAME
00	0 (MACHINE ZERO)	33	*ARG_VECT	66	FR_66	99	FR_99	CC	FR_CC
01	DATA_FLAG_RETURN	34	*ARG_VECT	67	FR_67	9A	FR_9A	CD	FR_CD
02	DATA_FLAG_ENTRY	35	*ARG_VECT	68	FR_68	9B	FR_9B	CE	FR_CE
03	PR_3	36	CG_36	69	FR_69	9C	FR_9C	CF	FR_CF
04	PR_4	37	CG_37	6A	FR_6A	9D	FR_9D	D0	FR_D0
05	PR_5	38	CG_38	6B	FR_6B	9E	FR_9E	D1	FR_D1
06	PR_6	39	CG_39	6C	FR_6C	9F	FR_9F	D2	FR_D2
07	PR_7	3A	CG_3A	6D	FR_6D	A0	FR_A0	D3	FR_D3
08	PR_8	3B	CG_3B	6E	FR_6E	A1	FR_A1	D4	FR_D4
09	PR_9	3C	CG_3C	6F	FR_6F	A2	FR_A2	D5	FR_D5
0A	PR_A	3D	CG_3D	70	FR_70	A3	FR_A3	D6	FR_D6
0B	PR_B	3E	CG_3E	71	FR_71	A4	FR_A4	D7	FR_D7
0C	PR_C	3F	CG_3F	72	FR_72	A5	FR_A5	D8	FR_D8
0D	PR_D	40	CG_40	73	FR_73	A6	FR_A6	D9	FR_D9
0E	PR_E	41	FR_41	74	FR_74	A7	FR_A7	DA	FR_DA
0F	PR_F	42	FR_42	75	FR_75	A8	FR_A8	DB	FR_DB
10	PR_10	43	FR_43	76	FR_76	A9	FR_A9	DC	FR_DC
11	PR_11	44	FR_44	77	FR_77	AA	FR_AA	DD	FR_DD
12	PR_12	45	FR_45	78	FR_78	AB	FR_AB	DE	FR_DE
13	PR_13	46	FR_46	79	FR_79	AC	FR_AC	DF	FR_DF
14	C_#20	47	FR_47	7A	FR_7A	AD	FR_AD	E0	FR_E0
				•					
				•					
				•					
1E	CALLEDATA	51	FR_51	84	FR_84	B7	FR_B7	EA	FR_EA
1F	DATA_FLAG_TABLE	52	FR_52	85	FR_85	B8	FR_B8	EB	FR_EB
20	DATABASE	53	FR_53	86	FR_86	B9	FR_B9	EC	FR_EC
21	PARM_DESCR	54	FR_54	87	FR_87	BA	FR_BA	ED	FR_ED
22	PI_DYNSP	55	FR_55	88	FR_88	BB	FR_BB	EE	FR_EE
23	C_#8	56	FR_56	89	FR_89	BC	FR_BC	EF	FR_EF
24	C_#A	57	FR_57	8A	FR_8A	BD	FR_BD	F0	FR_F0
25	K	58	FR_58	8B	FR_8B	BE	FR_BE	F1	FR_F1
26	AL_26	59	FR_59	8C	FR_8C	BF	FR_BF	F2	FR_F2
27	CG_27	5A	FR_5A	8D	FR_8D	C0	FR_C0	F3	FR_F3
28	D_L_0000	5B	FR_5B	8E	FR_8E	C1	FR_C1	F4	FR_F4
29	D_L_0008	5C	FR_5C	8F	FR_8F	C2	FR_C2	F5	FR_F5
2A	J	5D	FR_5D	90	FR_90	C3	FR_C3	F6	FR_F6
2B	AL_2B	5E	FR_5E	91	FR_91	C4	FR_C4	F7	FR_F7
2C	CG_2C	5F	FR_5F	92	FR_92	C5	FR_C5	F8	FR_F8
2D	L_97_DESCR	60	FR_60	93	FR_93	C6	FR_C6	F9	FR_F9
2E	L_C1_DESCR	61	FR_61	94	FR_94	C7	FR_C7	FA	FR_FA
2F	L_C00001_DESCR	62	FR_62	95	FR_95	C8	FR_C8	FB	FR_FB
30	L_F4_DESCR	63	FR_63	96	FR_96	C9	FR_C9	FC	FR_FC
31	L_F3_DESCR	64	FR_64	97	FR_97	CA	FR_CA	FD	FR_FD
32	*ARG_VECT	65	FR_65	98	FR_98	CB	FR_CB	FE	FR_FE
								FF	FR_FF

Figure 14-4. Source Listing Example (Sheet 3 of 5)

```
FORTRAN 200 CYCLE 14       BUILT 11/22/82 20:38   STORAGE MAP          PASCAL    COMPILED 12/07/82 14:22

                              PROGRAM NAME IS PASCAL     TOTAL LENGTH IS       56 HEX HALF WORDS


 DATA AREA COPY OF ALL REGISTERS USED BY THIS FORTRAN PROGRAM
    START ADDRESS =          1280                                  (START ADDRESS IS RELATIVE TO DATA AREA BASE ADDRESS)

  SCALARS AND CONSTANTS ASSIGNED TO REGISTERS                      (LOCATIONS ARE RELATIVE TO DATA AREA BASE ADDRESS)
    LOCATION   REG.NO          NAME                          CLASS              TYPE

       1300      22      PI_DYNSP                      SIMPLE VARIABLE       INTGR
       1340      23      C_#B                          CONSTANT              INTGR
       1380      24      C_#A                          CONSTANT              INTGR
       13C0      25      K                             SIMPLE VARIABLE       INTGR
       1480      28      D_L_0000                      SIMPLE VARIABLE       INTGR
       14C0      29      D_L_0008                      SIMPLE VARIABLE       INTGR
       1500      2A      J                             SIMPLE VARIABLE       INTGR

  DESCRIPTORS ASSIGNED TO REGISTERS                                (LOCATIONS ARE RELATIVE TO DATA AREA BASE ADDRESS)
    LOCATION   REG.NO          NAME

       15C0      2D      L_97_DESCR                    ARRAY
       1600      2E      L_C1_DESCR                    ARRAY
       1640      2F      L_C00001_DESCR                CHAR/BIT/FORMAT
       1680      30      L_F4_DESCR                    CHAR/BIT/FORMAT
       16C0      31      L_F3_DESCR                    CHAR/BIT/FORMAT
       1700      32      *ARG_VECT                     ARGUMENT VECTOR
       1740      33      *ARG_VECT                     ARGUMENT VECTOR
       1780      34      *ARG_VECT                     ARGUMENT VECTOR
       17C0      35      *ARG_VECT                     ARGUMENT VECTOR

      NOTE: TOTAL NUMBER OF  REGISTERS TO BE FETCHED INTO REG.FILE STARTING WITH REG.20 HEX IS 16 HEX


 GENERATED OBJECT CODE
    START ADDRESS =            0       LENGTH =       56 HEX HALF WORDS      (START ADDRESS IS RELATIVE TO CODE AREA BASE ADDRESS)


 CHARACTER CONSTANTS,LITERALS AND FORMAT SEGMENTS
    START ADDRESS =            0       LENGTH =       1A HEX HALF WORDS      (START ADDRESS IS RELATIVE TO DATA AREA BASE ADDRESS)


 ARGUMENT VECTORS
    START ADDRESS =          340       LENGTH =       48 HEX HALF WORDS      (START ADDRESS IS RELATIVE TO DATA AREA BASE ADDRESS)
```

Figure 14-4. Source Listing Example (Sheet 4 of 5)

```
CONSTANTS, DESCRIPTORS, NON-COMMON VARIABLES, AND NAMELISTS NOT ASSIGNED TO REGISTERS
   START ADDRESS =          C40      LENGTH =       0 HEX HALF WORDS      (START ADDRESS IS RELATIVE TO DATA AREA BASE ADDRESS)

   LOCATION     SYMBOLIC NAME OR HEX VALUE            CLASS          TYPE  (LOCATIONS ARE RELATIVE TO DATA AREA BASE ADDRESS)

       C80      I                                SIMPLE VARIABLE     INT
       CC0      A                                SIMPLE VARIABLE     INT
       D00      B                                SIMPLE VARIABLE     INT
       D40      L                                ARRAY VARIABLE      INT
      1000      4F55545055542020                 CONSTANT            TYP

   LOCATION     SYMBOLIC NAME OR HEX VALUE            CLASS          TYPE  (LOCATIONS ARE RELATIVE TO DATA AREA BASE ADDRESS)

      1040                   0                   CONSTANT            INT


TEMPORARY STORAGE
                                    LENGTH =       0 HEX HALF WORDS      (STORAGE IS SCATTERED THROUGHOUT DATA AREA)


COMMON BLOCKS

   NO COMMON BLOCK IS SPECIFIED


LIST OF ALL ENTRY POINTS

   LOCATION     SYMBOLIC NAME                                             (LOCATIONS ARE RELATIVE TO CODE AREA BASE ADDRESS)

         0        PASCAL


LIST OF ALL EXTERNALS

                SYMBOLIC NAME

                  F_EPILOG
                  F_CPFSO
                  F_TFO
                  F_LPFO
                  F_PROLOG
14.30.37.UCLP, 10, L2T1P2 ,       0.59KLNS.
```

Figure 14-4. Source Listing Example (Sheet 5 of 5)

Variable Map

The variable map provides information about each symbolic name used in a program except for procedure names and symbolic constant names. The variable map is always printed when you select the LO=X compilation option. See figure 14-5 for the variable map format and figure 14-4 for a variable map example.

Some uses of the variable map include:

- Identifying symbolic names that are not associated with the proper data type
- Locating where symbolic names are assigned values
- Identifying functions that should be arrays
- Locating misspelled symbolic names
- Verifying that symbolic names are in the proper common blocks
- Finding all statements in a program for a given symbolic name reference
- Finding symbolic names that are defined but never used

```
VARIABLE MAP
--NAME---BLOCK---ADDRESS---TYPE---CLASS---REFERENCES

   sym    blk      addr    typ     cls       refs
   .      .        .       .       .         .
   .      .        .       .       .         .
   .      .        .       .       .         .
```

sym — A symbolic name that appears in the program. Symbolic names are listed in alphabetical order.

blk — The name of the common block in which sym appears. If sym appears in the unnamed common block, two consecutive slashes are printed for blk. If sym does not appear in any common block, the blk field is left blank.

addr — Bit address of the variable relative to the blk or the database if blk is left blank, or if the register is assigned to the variable. Full word registers are specified as REG rr and halfword registers as HREG rr.

typ — The data type with which sym is associated; typ can be any of the following:

```
INTEGER
HALF PRECISION
REAL
DOUBLE
COMPLEX
LOGICAL
CHAR*len (len is the character length)
BIT
```

cls — The class of sym; cls can be any of the following:

```
SIMPLE
ARRAY
DESCRIPTOR
DESCRIPTOR ARRAY
UNKNOWN
```

refs — The source line numbers of all source lines that contain references to sym. The source line numbers are listed in numerical order, and multiple references are listed. A source line number appearing in refs can be followed by a suffix. A suffix describes how sym is used in the source line. The suffixes and their meanings are:

- /A The symbolic name sym is an actual argument in a subroutine call or function reference.
- /C The symbolic name sym is the control variable of a DO loop.
- /I The symbolic name sym is initialized in a DATA statement.
- /R The symbolic name sym appears in the input/output list of an input statement.
- /S The symbolic name sym appears on the left side of an assignment statement.
- /W The symbolic name sym appears in the input/output list of an output statement.

Figure 14-5. Variable Map Format

Symbolic Constant Map

The symbolic constant map provides information about each symbolic constant used in a program. See figure 14-6 for the symbolic constant map format and figure 14-4 for a symbolic constant map example. The symbolic constant map will not print if the given program uses no symbolic constants. Uses of the symbolic constant map include:

- Verifying that symbolic constant names are assigned proper values
- Verifying association of a symbolic constant with the proper data type
- Finding symbolic constant names that are defined but never used
- Finding the defining PARAMETER statement for each symbolic constant and all occurrences of a given symbolic constant in the program

```
SYMBOLIC CONSTANT MAP
--NAME---TYPE---VALUE---REFERENCES

   sym   typ     val       refs
   .     .       .         .
   .     .       .         .
   .     .       .         .
```

sym — A symbolic name that appears in the program. Symbolic names are listed in alphabetical order.

typ — The data type with which sym is associated; typ can be any of the following:

```
INTEGER
HALF PRECISION
REAL
DOUBLE
COMPLEX
LOGICAL
CHAR*len (len is the character length)
BIT
```

val — The value assigned to the symbolic name sym. The format of var depends on the data type of sym:

Integer
: The integer value is printed. A negative value is preceded by a minus sign.

Half-precision, real, and double-precision
: The value is printed as a hexadecimal string constant. The format is X'nnn'.

Complex
: The complex value is printed as two hexadecimal string constants. The first constant represents the real part, and the second constant represents the imaginary part. The format is X'nnn',X'nnn'.

Logical
: The logical value is printed as the logical constant .TRUE. or .FALSE..

Character
: The character value is printed as a character string enclosed in apostrophes. If the string is too long to fit in the columns provided, the trailing apostrophe is replaced by an ellipsis.

Bit
: The bit value is printed as a bit string enclosed in apostrophes. The format is B'nnn'. If the string is too long to fit in the columns provided, the trailing apostrophe is replaced by an ellipsis.

refs — The source line numbers of all source lines that contain references to sym. The source line numbers are listed in numerical order, and multiple references are listed. A source line number appearing in refs can be followed by the suffix /S, which indicates that the symbolic constant is defined in that source line.

Figure 14-6. Symbolic Constant Map Format

Procedure Map

The procedure map provides information about subroutines, functions, statement functions, and external symbolic names used in a program. The procedure map will not print if the program uses no procedures. See figure 14-7 for the procedure map format and figure 14-4 for a procedure map example. Some uses of the procedure map include:

Identifying statement functions that should be arrays

Verifying association of procedure names with the proper data types

Finding misspelled procedure names

Finding statement function definitions

Finding defined statement function names that are never used

Finding all procedure name references

```
PROCEDURE MAP
--NAME---TYPE---CLASS---REFERENCES

   sym   typ     cls       refs
   .     .       .         .
   .     .       .         .
   .     .       .         .
```

sym The symbolic name of a subroutine, function, statement function, or external symbol. Symbolic names are listed in alphabetical order.

typ The data type with which sym is associated; typ can be any of the following:

- INTEGER
- HALF PRECISION
- REAL
- DOUBLE
- COMPLEX
- LOGICAL
- CHAR*len (len is the character length)
- BIT
- GENERIC (for generic intrinsic functions)

If the symbolic name sym is a subroutine name or an external symbol, the typ field is left blank.

cls The class of sym; cls can be any of the following:

- SUBROUTINE
  Subroutine
- DUMMY SUBR
  Subroutine name is a dummy argument
- INTRINSIC
  Intrinsic function
- STAT FUNC
  Statement function
- BASIC EXTRN
  Basic external function
- DUMMY FUNC
  Function name is a dummy argument
- EXTERNAL
  The symbolic name sym appears in an EXTERNAL statement and is not of any other class

refs The source line numbers of all source lines that contain references to sym. The source line numbers are listed in numerical order, and multiple references are listed. If sym is a statement function name, a source line number appearing in refs can be followed by the suffix /D, which indicates that the statement function is defined in that source line.

Figure 14-7. Procedure Map Format

ASSEMBLY LISTING

If you select LO=A, an assembly representation of the FORTRAN program appears after any cross-reference maps. See figure 14-4 for an assembly listing example. See the CYBER 200 Assembler reference manual for more information on assembly language. The assembly listing includes:

The location counter (the offset from the code area base address)

The machine instruction in hexadecimal (either halfword or fullword instruction)

The source line number of the associated source program statement

The instruction mnemonic, instruction qualifiers, and operands

REGISTER MAP AND STORAGE MAP

If you select LO=M, a listing of the contents of the 256-register register file and a storage map appear after any assembly listing. See figure 14-4 for an example register map and storage map. The storage map gives the following information:

Starting address and size of data area copy of the register file

Name, location, class, and data type of all scalars, constants, and externals assigned to registers

Name, location, and class of descriptors assigned to registers

Length and starting address of the object code

Length and starting address of character constants, literals, and format segments

Length and starting address of argument vectors

Length and starting address of constants, externals, descriptors, variables (not in common), namelist groups, and character scalars not assigned to registers

Quantity of temporary storage

Common blocks

Entry points

Externals

The FORTRAN 200 register usage conforms to standard CYBER 200 Operating System register conventions, which are described in volume 2 of the CYBER 200 Operating System reference manual.

INDEX MAP

If you select LO=SI, a source listing followed by a sorted compiled-module listing (index map) is provided. The index map is an alphabetized listing with the following information:

Module name

Number of errors

Starting page number

The I option is valid with the LO compilation options A, M, and X when the S option is also specified. See figure 14-8 for an index map example.

```
                                COMPILED-MODULE LISTING
                                -----------------------

MODULE NAME                             NUMBER OF ERRORS                STARTING PAGE NUMBER
-----------                             ----------------                --------------------

  CLINIC                        190  WARNING     40  FATAL                      1
  FILTER                         31  WARNING                                17
  FORTXD                  NO ERRORS                                         21
  OCEAN                           5  WARNING                                23
  RELAX                          45  WARNING     13  FATAL                  38
  STATE                           2  WARNING                                45
  STEP                           40  WARNING      5  FATAL                  47
  TRACER                         76  WARNING     13  FATAL                  56

    8 MODULES COMPILED
```

Figure 14-8. Compiled-Module Listing (Index-Map) Example

EXECUTION-TIME FILE REASSIGNMENT

You can change the file preconnections used by your program at execution time. To do so, you can either override the preconnection specifier list on the PROGRAM statement or append additional specifications to the list.

NOTE

Execution-time file reassignment is only effective for preconnections specified on the PROGRAM statement. It cannot change connections specified on OPEN statements.

To reassign files at execution time, you enter an execution statement consisting of the controllee file name (default name GO), a separator character (blank, comma, or left parenthesis), and the new preconnection specifier list. To completely replace the list, prefix the new list with two asterisks (**).

Assuming the controllee file name is GO, the possible formats are as follows:

```
    GO(**message)
or
    GO **message.
or
    GO,**message.
```

Entirely overrides. Preconnection specifiers in the message are used to replace the entire PROGRAM statement preconnection specifier list.

```
    GO(message)
or
    GO message.
or
    GO,message.
```

Concatenates. The specifications are processed as though the message followed a specification in the PROGRAM statement (that is, when a new unit is given in the message, it creates a new unit in addition to any unit in the PROGRAM statement). If the same unit is given in the message and previously declared in the PROGRAM statement, then it will reconnect the unit with the new file specified in the message. As a result of this, the old file is no longer connected to its unit. If you still need the old file, the preconnection specifier should be specified again in the message. (See page 7-2 for preconnection specifier information.)

The message is a list of declarations in the same format as the PROGRAM statement. See section 7 for a description of the PROGRAM statement.

Any unit given in the message but not in the PROGRAM statement creates a new unit and is in addition to any unit in the PROGRAM declaration.

If the same unit is given in the message and previously defined in the PROGRAM statement, it will first be disconnected from the file assigned in the PROGRAM statement declaration and then will make a new preconnection with the file specified in the message.

You cannot use the concatenation form to get around syntax, file name, or parameter errors in the original PROGRAM statement. The original PROGRAM statement declaration string is processed before the execution-time reassignments.

When you execute a program interactively under DEBUG, there will be a prompt for a preconnection specifier list. You can respond with a period for no reassignment or with a preconnection specifier list in any of the foregoing formats.

CONTROL OF DROP FILE SIZE

If you get an execution-time error message DROP FILE OVERFLOW, increase the size of the drop file and rerun the program. Increase the drop file size with the CDF parameter of the LOAD system control statement or with the D parameter of the SWITCH system control statement. Increasing the drop file size usually solves the overflow problem, but sometimes a program error (especially an infinite loop) could be the cause.

ERROR MESSAGES

If you improperly use the options in the compiler's control statement or if the compiler fails, it will issue an error message. The control statement error messages are listed in Appendix B.

CHARACTER SETS

A

The CYBER FORTRAN 200 compiler recognizes 52 characters. The FORTRAN character set is a subset of the ASCII 64-character set, and the ASCII 64-character set is a subset of the VSOS character set. See table A-1.

Any of the characters in the FORTRAN character set can appear in a FORTRAN program. Any of the characters in the 64-character set can appear in comments and character strings.

Table A-1 also shows the internal hexadecimal representation and the Hollerith punch code for each character. Each hexadecimal digit in the internal hexadecimal representation corresponds to 4 bits. The Hollerith punch code indicates the rows that are punched in a computer card for each character.

Some characters do not appear on all keypunches and terminals. If a particular character is not represented on a keypunch or terminal, a character that appears on the keypunch or terminal that has the same internal hexadecimal representation can be substituted.

TABLE A-1. CHARACTER SETS

CYBER FORTRAN 200 Character Set	ASCII 64-Character Set	VSOS Character Set	Hex	Decimal
Δ	Δ	Δ space	20	032
	!	! exclamation point	21	033
	"	" quote	22	034
	#	# pound sign	23	035
	$	$ dollar sign	24	036
	%	% percent sign	25	037
&	&	& ampersand	26	038
'	'	' apostrophe	27	039
(	(	(left parenthesis	28	040
)	)	) right parenthesis	29	041
*	*	* asterisk	2A	042
+	+	+ plus	2B	043
,	,	, comma	2C	044
-	-	- minus	2D	045
.	.	. period	2E	046
/	/	/ slash	2F	047
0	0	0	30	048
1	1	1	31	049
2	2	2	32	050
3	3	3	33	051
4	4	4	34	052
5	5	5	35	053
6	6	6	36	054
7	7	7	37	055
8	8	8	38	056
9	9	9	39	057
:	:	: colon	3A	058
;	;	; semicolon	3B	059
<	<	< less than	3C	060
=	=	= equals sign	3D	061
>	>	> greater than	3E	062
	?	? question mark	3F	063
	@	@ commercial at	40	064
A	A	A	41	065
B	B	B	42	066
C	C	C	43	067
D	D	D	44	068
E	E	E	45	069
F	F	F	46	070
G	G	G	47	071

TABLE A-1. CHARACTER SETS (Contd)

CYBER FORTRAN 200 Character Set	ASCII 64-Character Set	VSOS Character Set	Hex	Decimal
H	H	H	48	072
I	I	I	49	073
J	J	J	4A	074
K	K	K	4B	075
L	L	L	4C	076
M	M	M	4D	077
N	N	N	4E	078
O	O	O	4F	079
P	P	P	50	080
Q	Q	Q	51	081
R	R	R	52	082
S	S	S	53	083
T	T	T	54	084
U	U	U	55	085
V	V	V	56	086
W	W	W	57	087
X	X	X	58	088
Y	Y	Y	59	089
Z	Z	Z	5A	090
[	[	[left bracket	5B	091
	\	\ reverse slash	5C	092
]	]	] right bracket	5D	093
	^	^ circumflex	5E	094
	_	_ underscore	5F	095
	`	` reverse apostrophe	60	096
		a	61	097
		b	62	098
		c	63	099
		d	64	100
		e	65	101
		f	66	102
		g	67	103
		h	68	104
		i	69	105
		j	6A	106
		k	6B	107
		l	6C	108
		m	6D	109
		n	6E	110
		o	6F	111
		p	70	112
		q	71	113
		r	72	114
		s	73	115
		t	74	116
		u	75	117
		v	76	118
		w	77	119
		x	78	120
		y	79	121
		z	7A	122
		{ left brace	7B	123
		| vertical bar	7C	124
		} right brace	7D	125
		~ tilde	7E	126

This appendix describes the five groups of diagnostic messages:

Compiler failure messages

Compilation error messages

Execution-time error messages

Vectorizer messages

Control statement error messages

COMPILER FAILURE AND COMPILATION ERRORS

When the compiler fails or detects errors, it issues a message. The compiler failure and compilation error messages are listed in table B-1.

COMPILER FAILURE

When the compiler fails, it generates error messages.

The compiler error type is:

A(abort)

The compiler failed and compilation was terminated. The return code is 8 (RC=8).

COMPILATION ERRORS

When the compiler detects errors in the source program, it issues one or more messages.

Lowercase letters such as param, name, and window appear in some of the messages. These lowercase letters indicate the position where a part of the source code is inserted with the error message. The term param means a parameter is inserted, name means a name is inserted, and window means a source line segment is inserted. This provides you with more specific information on the nature of the error than the message alone provides.

Some of the error numbers under 900 have no currently assigned messages. These are reserved by CDC for future use. The range from 900 to 999 is reserved for individual site use.

The compilation error types are:

N(non-ANSI)

The statement with one or more errors did not comply with ANSI Standard X3.9-1978. Whether the error(s) are ignored, are warning, or are fatal depends on the state of the ANSI option.

W(warning)

The statement containing one or more errors compiled to the end, but part of the statement might not have been processed. The return code is 4 (RC=4).

F(fatal)

The statement containing one or more errors did not compile and did not generate object code. The return code is 8 (RC=8).

RETURN CODES

With a batch job, you can choose whether to initiate error exit processing or to allow the job processing to continue. You have control through the Termination Value (TV) control statement and the error Return Code value from the compiler.

All return codes that are less than or equal to the termination value you specify in the TV control statement are ignored, and the job processing continues. All return codes that are greater than the termination value you specify in the TV control statement initiate error processing as specified by the EXIT control statement. The Termination Value control statement is discussed in the appropriate operating system reference manual.

EXECUTION-TIME ERRORS

When errors are detected during execution of a previously compiled program, the system issues one or more messages. The execution-time error messages are listed in table B-2.

Some of the error numbers under 900 have no currently assigned messages. These are reserved by CDC for future use. The range from 900 to 999 is reserved for individual site use.

The System Error Processor (SEP) can be used to change the attributes of certain execution-time errors.

The execution-time error types are:

W(warning)

The system sends a warning message to the user via some peripheral device and continues execution until the warning limit is reached. The return code for the warning limit is 4 (RC=4). Errors with a warning classification can be altered by SEP to be fatal.

F(fatal)

Execution is terminated abnormally. The return code is 8 (RC=8). Errors with a fatal classification can be altered by SEP to be warnings.

C(catastrophic)

Execution is terminated abnormally. The return code is 8 (RC=8). Catastrophic errors cannot be altered by SEP to be fatal or warnings. You cannot control the condition except for replacement of the standard message.

Many execution-time error messages have extra information appended. The form of an execution-time error message is:

```
ERROR xxx IN subr AT LINE nn     message
```

where xxx is the error number, subr is the name of the error-containing routine, nn is the line number of the error-containing statement, and message is the execution-time error message as it appears in table B-2.

This form indicates the error location in the program. Since the error is detected in an execution-time routine, the identified statement should be an I/O statement or a reference to a FORTRAN 200 supplied function such as SIN or COS.

The form of the error message for data flag branch errors is:

```
ERROR xxx: EXECUTION INTERRUPTED
IN subr AT LINE nnn     message
```

where xxx, subr, nn, and message have the same meanings as in the foregoing execution-time error message.

Data flag branch error messages have the register 1 address appended to them.

If the error occurred in your routine, the subroutine should correspond to the register 1 address. If, however, the register 1 address is in an execution-time routine, the subroutine and line number identify the location in your program that generated the call to FORTRAN execution-time.

VECTORIZER MESSAGES

A vectorizer message indicates the compiler's first impediment to vectorization. The messages are only informative and not associated with any return code. The messages are listed in table B-3. They are issued when you specify the FORTRAN 200 control statement option OPTIMIZE=V. The form of a vectorizer message is:

```
LINE xxxxx LINE yyyyy  message
```

where xxxxx is the source line number where the DO loop begins, yyyyy is the source line number where the impediment is detected, and message is the vectorizer message.

CONTROL STATEMENT ERROR MESSAGES

If you improperly use the options in the compiler's control statement or if the compiler fails, it issues an error message. The control statement error messages are listed in table B-4. There are three kinds of error messages:

WARNING

Message issued and compilation proceeds.

ERROR

Message issued and compilation terminates.

COMPILER FAILURE

Message issued and compilation terminates.

TABLE B-1. COMPILER ERROR MESSAGES

Error Number	Error Type	Message	Significance	Action
93	F	SYNTAX ERROR IN SAVE STATEMENT	(self explanatory)	Correct error; recompile.
94	W	REDUNDANT SAVE STATEMENT	There are two SAVE statements. They have been used in such a way as to save the same variable.	Correct error; recompile.
95	W	REDUNDANT APPEARANCE OF name IN SAVE STATEMENT	A variable may appear in a SAVE statement only once.	Correct error; recompile.
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	A	COMPILER FAILURE - ERROR MESSAGE num TOO LONG	The compiler-built error message exceeded 117 characters.	Follow site-defined procedure.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Type	Message	Significance	Action
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 - INVALID 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.
102	F	INVALID 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	F	EXPECTED COMMON BLOCK NAME IN VICINITY OF <<<<window>>>>	A common block name is expected.	Correct error; recompile.
105	F	EXPECTED "," IN VICINITY OF <<<<window>>>>	A comma is expected between items in a SAVE statement.	Correct error; recompile.
106	F	MISSING OPERATOR OR DELIMITER IN VICINITY OF <<<<window>>>>	An operator or delimiter is required.	Supply missing operator or delimiter; recompile.
107	F	INVALID OPERAND IN VICINITY OF <<<<window>>>>	An expression contains an invalid operand.	Correct error; recompile.
108	F	INVALID OR MISSING DELIMITER IN VICINITY OF <<<<window>>>>	A delimiter is required.	Supply missing delimiter or correct error in existing delimiter; recompile.
109	F	INVALID 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	INVALID USE OF HEXADECIMAL CONSTANT	A hexadecimal constant is used improperly.	Correct error; recompile.
112	F	RECURSIVE SUBPROGRAM REFERENCE	A subprogram calls itself.	Remove recursive subprogram references from the program; recompile.
113	F	INVALID ARGUMENT DELIMITER IN VICINITY OF <<<<window>>>>	Arguments must be delimited by commas.	Correct error; recompile.
114	W	SAVE STATEMENT MUST PRECEDE ALL EXECUTABLE STATEMENTS	A SAVE statement incorrectly follows executable statements.	Correct error; recompile.
115	F	EXPECTED "/" IN VICINITY OF <<<<window>>>>	A slash must precede and follow any common block name in a SAVE statement.	Correct error; recompile.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Type	Message	Significance	Action
116	F	EXPECTED COMMON BLOCK NAME IN VICINITY OF <<<<window>>>>	A common block name is expected in the SAVE statement.	Correct error; recompile.
117	W	STATEMENT FUNCTION name NOT ALLOWED IN SAVE STATEMENT	SAVE applies only to local variables or COMMON blocks.	Correct error; recompile.
118	F	INVALID OPERATOR IN VICINITY OF <<<<window>>>>	The operator cannot be used in the expression.	Correct error; recompile.
120	W	EXTERNAL name NOT ALLOWED IN SAVE STATEMENT	SAVE applies only to local variables or COMMON blocks.	Correct error; recompile.
121	W	INTRINSIC FUNCTION name NOT ALLOWED IN SAVE STATEMENT	SAVE applies only to local variables or COMMON blocks.	Correct error; recompile.
122	F	INVALID TYPE MIXING IN VICINITY OF <<<<window>>>>	The data types of two entities that appear in the statement are incompatible.	Correct error; recompile.
123	W	NAMELIST GROUP name NOT ALLOWED IN SAVE STATEMENT	SAVE applies only to local variables or COMMON blocks.	Correct error; recompile.
124	F	INVALID TYPE USAGE IN A RELATIONAL OR ARITHMETIC EXPRESSION IN VICINITY OF <<<<window>>>>	(self explanatory)	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	A 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	F	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	F	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	F	EXPONENT OVERFLOW IN CONSTANT ARITHMETIC	Constant arithmetic produces exponent overflow.	Verify that exponent overflow does not affect the logic of the program.
130	F	INVALID DELIMITER IN VECTOR REFERENCE IN VICINITY OF <<<<window>>>>	A vector reference has an invalid delimiter: e.g. a colon where a semicolon is required.	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	W	ENTRY POINT name NOT ALLOWED IN SAVE STATEMENT	SAVE applies only to local variables or COMMON blocks.	Correct error; recompile.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Type	Message	Significance	Action
133	W	PROGRAM NAME NOT ALLOWED IN SAVE STATEMENT	SAVE applies only to local variables or COMMON blocks.	Correct error; recompile.
134	W	BLOCK DATA NAME NOT ALLOWED IN SAVE STATEMENT	SAVE applies only to local variables or COMMON blocks.	Correct error; recompile.
135	F	INVALID LABEL IN VICINITY OF <<<<window>>>>	A label must be numeric and between 1 and 99999.	Supply numeric label; recompile.
136	F	DESCRIPTOR TYPE IS NOT INTEGER, HALF PRECISION, REAL, BIT, OR COMPLEX	A descriptor must be of one of these types.	Change the type of the descriptor; recompile.
137	F	INVALID DELIMITER FOR HEX OR BIT CONSTANT IN VICINITY OF <<<<window>>>>	Hexadecimal and bit constants must be delimited by apostrophes.	Change delimiters to apostrophes; recompile.
138	F	DOUBLY DEFINED LABEL	The same label appears on more than one statement in a program.	Change one of the occurrences of the label. Also, check all references to the label that is changed in order to maintain correct logic; recompile.
139	F	NO SOURCE PROGRAM	The input file specified in the FORTRAN control statement does not exist or is empty.	Correct error in the INPUT parameter of the FORTRAN control statement; recompile.
140	W	SYMBOLIC CONSTANT name NOT ALLOWED IN SAVE STATEMENT	SAVE applies only to local variables or COMMON blocks.	Correct error; recompile.
141	W	DUMMY ARGUMENT name NOT ALLOWED IN SAVE STATEMENT	SAVE applies only to local variables or COMMON blocks.	Correct error; recompile.
142	W	COMMON BLOCK ELEMENT name NOT ALLOWED IN SAVE STATEMENT	SAVE applies only to local variables or COMMON blocks.	Correct error; recompile.
143	W	SAVED VARIABLE name CANNOT APPEAR IN COMMON STATEMENT	SAVE applies only to local variables or COMMON blocks.	Correct error; recompile.
144	F	SUBSCRIPT MUST BE INTEGER CONSTANT	The subscript is not an integer constant.	Change the subscript to integer constant; recompile.
145	F	SPECIFICATION STATEMENTS MUST PRECEDE ALL EXECUTABLE STATEMENTS	A specification statement appears after an executable statement.	Move all specification statements in front of all executable statements; recompile.
146	F	INVALID VARIABLE name 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 IN VICINITY OF <<<<window>>>>	An error appears in the DATA statement.	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 IN VICINITY OF <<<<window>>>>	An error appears in a hexadecimal or bit constant.	Correct error; recompile.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Type	Message	Significance	Action
151	F	INVALID DATA ITEM IN VICINITY OF <<<<window>>>>	Incorrect data item in a DATA statement.	Correct error; recompile.
152	F	INVALID VECTOR REFERENCE TYPE IN DATA STATEMENT IN VICINITY OF <<<<window>>>>	Incorrect type for a vector reference in the variable list of a 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	INVALID USE OF VECTOR REFERENCE IN DATA STATEMENT	Vector reference used improperly in a DATA statement.	Correct error; recompile.
155	F	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 IN VICINITY OF <<<<window>>>>	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	INVALID ELEMENT IN SPECIFICATION STATEMENT IN VICINITY OF <<<<window>>>>	Invalid element in TYPE statement.	Correct error; recompile.
159	F	INVALID OPERATOR IN SPECIFICATION STATEMENT IN VICINITY OF <<<<window>>>>	Operator used incorrectly in TYPE statement or DIMENSION statement.	Correct error; recompile.
160	F	SYNTAX ERROR IN LENGTH SPECIFICATION OF CHARACTER VARIABLE(S) IN VICINITY OF <<<<window>>>>	The length specification that appears in a CHARACTER statement is incorrect.	Correct error; recompile.
161	F	SAVED VARIABLE name CANNOT APPEAR IN EXTERNAL STATEMENT	SAVE applies only to local variables or COMMON blocks.	Correct error; recompile.
162	W	VARIABLE name 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 SPECIFICATION FORMAT ERROR IN VICINITY OF <<<<window>>>>	The length specification that appears in a CHARACTER statement is incorrect.	Correct error; recompile.
164	F	CHARACTER LENGTH MUST BE GREATER THAN 0 AND LESS THAN 65536	The length specification for a character variable is zero, negative, or too large.	Correct error; recompile.
165	F	SAVED VARIABLE name CANNOT APPEAR IN INTRINSIC STATEMENT	SAVE applies only to local variables or COMMON blocks.	Correct error; recompile.
166	F	INVALID STATEMENT ON LOGICAL IF	The consequent statement on a logical IF is not allowed.	Correct error; recompile.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Type	Message	Significance	Action
167	W	NO LABELED COMMON IN BLOCK DATA SUBPROGRAM	No labeled common blocks are declared in the BLOCK DATA subprogram.	Verify that all statements appear in the BLOCK DATA subprogram as intended.
168	F	INVALID STATEMENT IN BLOCK DATA SUBPROGRAM	This statement cannot appear in a BLOCK DATA subprogram.	Correct error; recompile.
169	F	AN ASSUMED SIZE BOUND IS NOT PERMITTED IN AN ARRAY ASSIGNMENT STATEMENT	(self explanatory)	Correct error; recompile.
170	F	INVALID FIELD SPECIFICATION IN FORMAT STATEMENT IN THE VICINITY OF <<<<window>>>>	This FORMAT statement contains a field that is not currently defined.	Correct error; recompile.
171	N	ENCODE AND DECODE ARE NOT PART OF STANDARD FORTRAN	This is a non-ANSI feature.	Use internal I/O instead.
172	W	FUNCTION NAME IS NOT ASSIGNED A VALUE	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.
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	"()" IS REQUIRED IF FUNCTION HAS NO ARAGUMENTS	The subprogram is compiled as a main program.	Supply the argument list for the FUNCTION statement; recompile.
176	F	INVALID DUMMY ARGUMENT IN VICINITY OF <<<<window>>>>	An argument that appears in a FUNCTION or SUBROUTINE statement is invalid.	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	INVALID NAMELIST NAME	A namelist name is invalid.	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	(self explanatory)	Correct error; recompile.
181	F	INVALID OPERATOR IN VICINITY OF <<<<window>>>>	(self explanatory)	Correct error; recompile.
182	F	INVALID OR MISSING VARIABLE IN VICINITY OF <<<<window>>>>	Expected variable reference is missing or malformed.	Correct error; recompile.
183	F	SYNTAX ERROR IN LABEL STRING	Label string for computed go to contains syntax error	Correct error; recompile.
184	N	HOLLERITH CONSTANTS ARE NOT DEFINED IN ANSI FORTRAN	Character constants are the only means of representing character data in ANSI FORTRAN	Correct error; recompile.
185	F	INVALID LABEL REFERENCE IN VICINITY OF <<<<window>>>>	(self explanatory)	Correct error; recompile.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Type	Message	Significance	Action
186	F	MORE THAN 253 COMMON BLOCK NAMES	Too many common blocks are used in the program.	Reduce the number of common blocks used; recompile.
187	F	ATTEMPTED TO RE-ORDER COMMON	COMMON and EQUIVALENCE statements conflict.	Correct error; recompile.
188	F	VARIABLE name APPEARS IN COMMON MORE THAN ONCE	(self explanatory)	Eliminate all but one occurrence of the variable from the COMMON statement; recompile.
189	F	ENTRY MUST BE IN A SUBROUTINE OR FUNCTION	An ENTRY statement appears in a main program or a BLOCK DATA subprogram.	Remove the ENTRY statement; recompile.
190	F	DUPLICATION OF DUMMY ARGUMENT name	The same name appears more than once in the dummy argument list of a FUNCTION, SUBROUTINE, or ENTRY statement.	Eliminate all but one occurrence of the dummy argument from the argument list of the statement; recompile.
191	N	REAL, HALF PRECISION, OR DOUBLE PRECISION SUBSCRIPT USAGE IS NON-ANSI	(self explanatory)	Place an INT function around the non-integer expression or otherwise repair the statement; recompile.
192	F	INCORRECT FORMATION OF I/O STATEMENT	I/O statement control information list is incorrect.	Correct error; recompile.
193	F	AN EXTERNAL I/O UNIT MUST BE OF TYPE INTEGER IN param STATEMENT	Units can be specified as integers only.	Correct error; recompile.
194	F	EXPRESSION ON A COMPUTED GOTO MUST BE TYPE INTEGER	The expression's type was not integer.	Correct error; recompile.
195	F	INVALID OPTION IN I/O STATEMENT	The option specified cannot be used with the input/output statement.	Eliminate or change the option; recompile.
196	F	REFERENCED UNDEFINED FORMAT	The format specified in an input/output statement is not defined in the program.	Check for a missing FORMAT statement, or check for an error in the format number specified in the input/output statement.
197	F	RECORD LENGTH MUST BE INTEGER CONSTANT OR INTEGER VARIABLE	A non-integer record length is specified in an input/output statement.	Change the record length specification to an integer constant or an integer variable; recompile.
198	F	FORMAT SPECIFICATION MUST BE A FORMAT LABEL, ARRAY NAME, OR INTEGER VARIABLE ASSIGNED TO A FORMAT LABEL	The format specification in an input/output statement is not valid.	Supply a legal format specification; recompile.
199	F	INVALID ELEMENT IN I/O LIST IN VICINITY OF <<<<window>>>>	Invalid element: e.g., an expression in an input list.	Correct error; recompile.
200	F	SYNTAX ERROR IN I/O LIST IN THE VICINITY OF <<<<window>>>>	(self explanatory)	Correct error; recompile.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Type	Message	Significance	Action
202	F	INVALID FORMATION OF COMMON STATEMENT	Syntax error in commmon statement.	Correct error; recompile.
203	F	COMMON BLOCK NAME IS NOT SYMBOLIC	An invalid symbol is specified as a common block name.	Supply a valid identifier as the name of the common block; recompile.
204	F	DUPLICATE SYMBOLIC NAME name IN COMMON STATEMENT	The same symbol appears more than once in a COMMON statement.	Change the symbols so that all of the symbols in the COMMON statement are unique; recompile.
205	F	DATA CANNOT BE PRESET IN BLANK COMMON	BLOCK DATA subprograms can be used to initialize data in named common blocks only.	Remove initialized variable from blank common or use executable statements to initialize it.
206	F	DUMMY ARGUMENT CANNOT APPEAR IN COMMON	A dummy argument appears in a COMMON statement.	Change the name of the dummy argument or change the name in the COMMON statement; recompile.
207	F	INTRINSIC FUNCTION name WAS GIVEN A NON-CONFIRMING TYPE -- TYPING IGNORED	Intrinsic function types cannot be changed by type specification statements.	Correct error; recompile.
208	F	A VARIABLE IN A DIMENSION STATEMENT MUST BE DIMENSIONED	The dimension specification for a variable that appears in a DIMENSION statement is not specified.	Add the dimension specification to the variable name that appears in the statement; recompile.
209	F	MISSING COMMA IN VICINITY OF <<<<window>>>>	A comma is required.	Supply the comma; recompile.
210	F	DIMENSIONING FORMAT ERROR IN VICINITY OF <<<<window>>>>	(self explanatory)	Correct error; recompile.
211	F	ERROR IN DIMENSIONING name - ONLY LAST UPPER DIMENSION BOUND CAN BE "*"	For columnwise arrays, an asterisk can be specified for only the upper bound of the last dimension.	Verify that a columnwise array is intended and correct error; recompile.
212	F	DIMENSION BOUND VARIABLE name NOT IN COMMON, OR IN ARGUMENT LIST WITH ARRAY	The variable has no value assigned to it.	Correct error; recompile.
213	F	ERROR IN DIMENSIONING name - ROWWISE, SO ONLY FIRST UPPER DIMENSION BOUND CAN BE "*"	For rowwise arrays, an asterisk can be specified for only the upper bound of the first dimension.	Verify that a rowwise array is intended and correct error; recompile.
215	F	MORE THAN 7 DIMENSIONS SPECIFIED FOR ARRAY name	An array can have no more than 7 dimensions.	Reduce the number of dimensions; recompile.
216	F	EXPLICIT TYPE OF name CONFLICTS WITH PREVIOUS USAGE	The type declaration statement for the named variable specifies a different type than that implied by an earlier data declaration statement.	Move the explicit type statement so that it is the first reference to the variable.
217	F	INVALID OR MISSING REFERENCE IN DO STATEMENT	(self explanatory)	Correct error; recompile.
218	F	LABEL REFERENCE GREATER THAN 99999	A label can have no more than 5 digits.	Shorten the label to 5 digits. Correct all references to the label appropriately; recompile.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Type	Message	Significance	Action
219	F	INVALID PARAMETER IN DO STATEMENT	The initial terminal or incrementation parameter in the DO statement is invalid.	Correct error; recompile.
220	N	ANSI STANDARD PROHIBITS MIXING CHARACTER AND NON-CHARACTER VARIABLE IN COMMON	The ANSI FORTRAN standard requires that a common block contain only character or noncharacter variables.	To maintain conformance with the ANSI standard, split the common block into two blocks, one for character data and the other for noncharacter data.
221	F	DO LOOP NEVER TERMINATED	An END statement appears in the range of a DO loop.	Supply the last statement of the DO loop if it is missing, or move the END statement out of the DO loop; recompile.
222	F	A DO LOOP MAY NOT TERMINATE ON THIS STATEMENT	This statement cannot be the last statement in a DO loop.	Add a CONTINUE statement after this statement. Move the label of this statement to the label field of the CONTINUE statement; recompile.
224	F	INVALID COMPONENT BEING EQUIVALENCED	The argument of the EQUIVALENCE statement is invalid.	Correct error; recompile.
225	F	INVALID DELIMITER SEPARATING EQUIVALENCE GROUPS	Equivalence groups must be separated by commas.	Add commas between equivalence groups; recompile.
226	F	ARRAY ELEMENT MUST HAVE AT LEAST ONE SUBSCRIPT	An array name appears that does not have a subscript.	Supply the subscript; recompile.
227	F	ONLY SYMBOLIC NAMES CAN APPEAR IN EXTERNAL STATEMENTS	Something other than a symbolic name appears in an EXTERNAL statement.	Correct error; recompile.
228	F	EXTERNAL STATEMENT DID NOT PRECEDE REFERENCE OR VARIABLE name IS WRONG TYPE	(self explanatory)	Correct error; recompile.
229	F	INVALID USE OF NAME IN EXTERNAL STATEMENT	(self explanatory)	Correct error; recompile.
230	F	INVALID EXPRESSION IN IF STATEMENT	An arithmetic IF statement must have an arithmetic or logical expression.	Correct error; recompile.
231	F	COMMA IS ONLY OPERATOR ALLOWED BETWEEN LABELS	An operator other than a comma was found between labels.	Change the operator to a comma; recompile.
232	F	SUBSCRIPT EXPRESSION NOT INTEGER, REAL, HALF PRECISION, OR DOUBLE PRECISION	A subscript expression can be integer, real, half precision, or double-precision.	Change the type of expression in the subscript; recompile.
233	F	UNIT MUST BE SPECIFIED IN AN OPEN STATEMENT	The UNIT specifier was omitted from this OPEN statement; there is no default unit for the OPEN statement.	Add a UNIT specifier to the OPEN statement.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Type	Message	Significance	Action
234	F	ITEMS IN COMMON MUST BE ARRAYS OR SIMPLE VARIABLES	Something other than an array or simple variable appears in a common block.	Remove the erroneous element from the COMMON statement; recompile.
235	F	COMPUTED GO TO EXPRESSION MISSING IN VICINITY OF <<<<window>>>>	A computed GO TO statement expected an integer expression.	Correct error; recompile.
236	W	UNREFERENCED FORMAT	A FORMAT statement appears in a program, but is not referenced in an input/output statement.	Check the format references in all input/output statements to find out if the proper formats are specified.
238	W	UNREFERENCED NAMELIST	A NAMELIST statement appears in the program, but it is not referenced in an input/output statement.	Check the namelist references in all input/output statements to find out if the proper namelists are specified.
239	W	CONFLICTING INITIALIZATION(S) OF name	A variable, array element, or substring can be initialized only once within a program unit.	The most recently encountered initialization is used. To eliminate the warning message, remove the redundant initialization.
240	N	FORMAT SPECIFIER IS NON-ANSI	This format specifier does not conform to Standard FORTRAN.	Use a form found in the FORTRAN 77 Standard; recompile.
241	F	BUFFER MUST BE VARIABLE OR ARRAY OR SUBSCRIPTED VARIABLE	Bad buffer specification in buffer input/output statement.	Correct error; recompile.
242	F	EQUIVALENCE RELATION ERROR BETWEEN GROUPS	Equivalence declaration conflicts with other declarations.	Correct error; recompile.
243	F	NON-REDEFINABLE VARIABLE IN INPUT LIST	The input list specifies a variable whose value cannot be altered, such as a DO loop control variable.	Remove the non-redefinable variable from the input list.
244	F	ARRAY name REFERENCED WITH TOO MANY SUBSCRIPTS	The number of subscripts specified in an array reference is not the same as the number of dimensions declared in the array declarator.	Check the array declarator and correct the array reference appropriately; recompile.
245	F	CONSTANTS MAY BE TOO LARGE	Conversion routine detects possible problem.	Correct error; recompile.
246	F	EQUIVALENCE HAS ATTEMPTED TO RE-ORIGIN COMMON	The EQUIVALENCE statement is incompatible with a COMMON statement. A common block cannot be extended at its beginning.	Correct the EQUIVALENCE statement so that it does not extend the common block at its beginning; recompile.
247	W	MISSING SUBSCRIPT(S) FOR ARRAY name, LOWER BOUND(S) SUBSTITUTED	An array reference has missing subscript expressions; the lower bound of the missing subscript is used.	Verify that the lower bound is intended.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Type	Message	Significance	Action
249	N	HALF-PRECISION DATA TYPE IS NON-ANSI	The half-precision data type is not defined in the ANSI FORTRAN standard.	To maintain conformance with the ANSI standard, use an ANSI standard data type.
250	W	RETURN STATEMENT IGNORED IN BLOCK DATA SUBPROGRAM	A BLOCK DATA subprogram does not permit a RETURN statement. The RETURN statement is ignored.	No action necessary.
251	W	RETURN STATEMENT REPLACED BY STOP STATEMENT IN MAIN PROGRAM	A main program requires a STOP statement rather than a RETURN statement. The RETURN statement is assumed to be a STOP statement.	No action necessary.
252	W	INVALID PARAMETER IN RETURN STATEMENT	The parameter in the RETURN statement is ignored.	Verify that ignoring the parameter does not affect the logic of the program.
253	W	TYPE OF RETURN PARAMETER MUST BE INTEGER	The parameter in a RETURN statement must be integer. The noninteger parameter is ignored.	Verify that ignoring the parameter does not affect the logic of the program.
254	W	INVALID VALUE FOR RETURN STATEMENT	Value is greater than the number of alternate returns, or is not positive.	Correct error; recompile.
255	F	SYNTAX ERROR ON LEFT SIDE OF ASSIGNMENT STATEMENT IN VICINITY OF <<<<window>>>>	An error appears to the left of an equals sign.	Correct error; recompile.
256	F	NON-REDEFINABLE VARIABLE ON LEFT SIDE OF ASSIGNMENT STATEMENT	The value of the variable that appears to the left of the equals sign cannot be changed.	Correct error; recompile.
257	F	INVALID FORMAT SPECIFIER	The item specified in the field reserved for the FORMAT specifier does not refer to a FORMAT.	Correct error; recompile.
258	F	FORMAT STATEMENT IN BLOCK DATA SUBPROGRAM	A FORMAT statement cannot appear in a BLOCK DATA subprogram.	Remove the FORMAT statement from the BLOCK DATA subprogram; recompile.
259	F	END STATEMENT MUST NOT BE CONTINUED	The END statement only appears once in each program unit. It must not be continued.	Correct error; recompile.
260	F	ILLEGAL STATEMENT IN Q8LINKV SEQUENCE	Your statement that uses Q8LINKV is not a special call statement.	Rewrite your statement as a special call; recompile.
261	F	CANNOT ASSIGN TYPELESS RESULT TO HALF PRECISION, DOUBLE PRECISION, OR COMPLEX VARIABLE <...>	Typeless result is 64 bits; cannot assign to a non-64-bit variable.	Change the type of the variable; recompile.
262	F	ASSIGN MUST BE FOLLOWED EITHER BY A LABEL OR A DESCRIPTOR VARIABLE	(self explanatory)	Correct error; recompile.
263	F	ASSIGN VARIABLE MUST BE SIMPLE INTEGER VARIABLE	(self explanatory)	Correct error; recompile.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Type	Message	Significance	Action
264	F	MISSING SUBSCRIPTS IN VICINITY OF <<<<window>>>>	An array name appears without subscripts.	Supply the subscripts; recompile.
265	F	MISSING LABEL(S) IN GO TO - POSSIBLE MISUSE OF COMPUTED GO TO STATEMENT IN SOURCE	A computed GO TO statement must specify statement labels to which control can transfer depending on the condition.	Supply proper number of statement labels; recompile.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Type	Message	Significance	Action
268	F	LOGICAL CONSTANT CANNOT INITIALIZE OTHER TYPES	A logical constant can initialize a logical variable only.	Replace the logical constant with a constant of the appropriate type, or change the type specification of the variable being initialized to logical; recompile.
269	F	LIST OF VARIABLES IN THE DATA STATEMENT IS LONGER THAN THE LIST OF CONSTANTS.	(self explanatory)	Eliminate the excessive variables, or add more constants to the DATA statement; recompile.
270	F	FLOATING POINT NUMBER OUT OF ALLOWABLE RANGE	A real constant is too small or too large to be represented.	Correct error; recompile.
271	F	TYPE MUST BE INTEGER CONSTANT OR INTEGER VARIABLE	A noninteger number is used where an integer or an integer variable is required.	Change the number to an integer; recompile.
273	W	MISSING END STATEMENT	The compiler supplied an END statement.	No action necessary.
274	F	ARRAY DECLARATOR NOT A VARIABLE IN VICINITY OF <<<<window>>>>	Array declarator in dimension statement is not a variable.	Correct error; recompile.
275	F	VARIABLE name CANNOT BE DIMENSIONED	The variable's use in a previous declaration conflicts with DIMENSION. (e.g., name appears both in an EXTERNAL statement and a DIMENSION statement).	Correct error; recompile.
276	F	ATTEMPT TO RE-DIMENSION VARIABLE name	The variable is already dimensioned.	Eliminate one of the dimension specifications or change the variable name; recompile.
277	F	PROGRAM STARTS WITH A CONTINUATION CARD	The first statement of a program has a nonzero, nonblank character in column 6.	Supply the source statements that are missing from the beginning of the program; recompile.
278	N	USE OF AN ASSUMED SIZE ARRAY AS AN INTERNAL UNIT IS NON-ANSI	The ANSI FORTRAN standard does not support internal file I/O to or from an array of unknown length (*).	To maintain conformance with the ANSI standard, explicitly define the array length.
279	F	ADJUSTABLE ARRAY name IS NOT A DUMMY ARGUMENT	The array name does not appear in the argument list of the FUNCTION or SUBROUTINE statement.	Place the array name in the argument list of the FUNCTION or SUBROUTINE statement and correct all subprogram references appropriately; recompile.
280	W	DIMENSION BOUND EXPRESSION(S) FOR ARRAY name CONVERTED TO INTEGER	The dimension bound expressions were truncated to integer.	Verify that the intended values for dimension bound expressions are used.
281	F	LOGICAL VARIABLES CAN ONLY BE INITIALIZED WITH .TRUE. OR .FALSE.	(self explanatory)	Correct error; recompile.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Type	Message	Significance	Action
283	W	EQUIVALENCE VARIABLE ATTEMPTED TO BE ASSIGNED TO IMPROPER BOUNDARY	An EQUIVALENCE statement attempted to assign a logical, integer, real, double-precision, or complex variable to a nonword boundary, or a character variable to a nonbyte boundary.	Correct error; recompile.
285	F	DO LOOP IS BRANCHED INTO, BUT HAS NO EXTENDED RANGE	Control cannot branch into a DO loop without first branching out of it.	Correct error; recompile.
287	F	REFERENCE TO UNDEFINED LABEL	A label is referenced, but it does not appear in the label field of any statement in the program.	Change the label reference so that it references a label that exists in the program, or supply the missing label in the program; recompile.
288	F	ZERO**ZERO OR NEGATIVE POWER IS AN UNDEFINED ARITHMETIC EXPRESSION	Raising zero to the zero power or to a negative power produces an undefined result.	Correct error; recompile.
290	W	FORMAT NOT LABELED	A FORMAT statement requires a label in the label field. The unlabeled FORMAT statement is not used.	Verify that the FORMAT statement is not referenced in the program.
291	F	INVALID TYPE FOR VECTOR LENGTH IN VICINITY OF <<<<window>>>>	The length must be integer.	Correct error; recompile.
292	F	VECTOR LENGTH CANNOT BE A NEGATIVE CONSTANT	(self explanatory)	Correct error; recompile.
293	F	TYPE ERROR IN A VECTOR ARITHMETIC OR BIT ASSIGNMENT STATEMENT	(self explanatory)	Correct error; recompile.
294	F	INVALID TYPE IN A VECTOR EXPRESSION	(self explanatory)	Correct error; recompile.
295	F	VECTOR EXPRESSION ASSIGNED TO A NON-VECTOR VARIABLE	A vector must be on the left side of a vector assignment statement.	Replace the variable on the left of the vector assignment statement with a vector; recompile.
296	F	SUBSCRIPT REFERENCE FOR NON-DIMENSIONED ARRAY name	A subscript is specified for a variable that is not dimensioned.	Use a DIMENSION statement to dimension the variable, or remove the subscript from the variable reference; recompile.
297	F	DESCRIPTOR NOT INITIALIZED BY VECTOR REFERENCE	(self explanatory)	Initailize the descriptor with a vector reference.
298	W	COMMON BLOCK HAS BEEN PADDED IN ORDER TO ENSURE ALIGNMENT	Alignment of the common block is performed by the compiler to place a character variable on a byte boundary or other variables (except bit) on a word boundary.	No action necessary.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Type	Message	Significance	Action
299	F	FIRST AND LAST MUST BE VARIABLES OR ARRAY ELEMENTS	Invalid specification for first or last location in BUFFER IN or BUFFER OUT statement.	Correct error; recompile.
300	F	EXTRANEOUS INFORMATION AT END OF STATEMENT	The compiler ignored the extra information at the end of the statement.	Verify that the compiler interpreted the statement correctly.
301	F	STATEMENT CANNOT BE IDENTIFIED	Syntax error in statement.	Correct error; recompile.
302	N	VECTOR USAGE IS NON-ANSI	Explicit vector usage is not permitted under the ANSI standard.	Change vector assignment to a DO loop assignment; recompile.
303	F	DIGIT STRING EXCEEDS MAXIMUM OF FIVE	No more than 5 digits can appear in the digit string.	Reduce the string to 5 digits; recompile.
304	F	INVALID CHARACTER	A character is used that is not in the FORTRAN 200 character set.	Replace the character with the appropriate character from the FORTRAN 200 character set; recompile.
305	W	INVALID CONSTANT ON A PAUSE OR STOP	The constant is ignored.	Verify that the constant is not intended.
306	F	INVALID CONSTANT TYPE	(self explanatory)	Correct error; recompile.
308	F	HOLLERITH FIELD COUNT IS TOO LARGE	Too many characters are in a Hollerith field. No more than 255 characters can appear in a Hollerith field.	Reduce the Hollerith field to no more than 255 characters; recompile.
309	F	SYMBOLIC NAME HAS MORE THAN 8 CHARACTERS	A symbolic name can consist of no more than 8 characters.	Reduce the symbolic name to no more than 8 characters; recompile.
312	F	LOGICAL CONSTANT OR LOGICAL/ RELATIONAL OPERATOR IS INCORRECT	(self explanatory)	Correct error; recompile.
313	F	ERROR IN HOLLERITH COUNT	Hollerith count does not accurately identify the number of elements in Hollerith constant.	Correct error; recompile.
314	F	REAL NUMBER CANNOT BE FOLLOWED BY A LETTER	Real number is ill-formed.	Correct error; recompile.
315	F	COMPLEX NUMBER COMPONENTS MUST BE REAL OR INTEGER	A complex number can consist of real or integer components only.	Change the double-precision components of the complex number to real; recompile.
316	F	MISSING RIGHT PARENTHESIS	A right parenthesis is required.	Supply the right parenthesis; recompile.
317	F	AN ASSUMED SIZE ARRAY IS NOT ALLOWED IN AN I/O LIST	An I/O list cannot include an array reference to an array of unknown size.	Ensure that the size of each array in the I/O list has been specified.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Type	Message	Significance	Action
318	F	ZERO LENGTH CHARACTER STRING	The length of a character string is specified to be zero.	Change the zero to a positive integer; recompile.
319	F	INCORRECT ARGUMENT FIELD SYNTAX	(self explanatory)	Correct error; recompile.
320	W	IMPLICIT STATEMENT MUST BE FIRST SPECIFICATION STATEMENT	Other statements appear before an IMPLICIT statement. The IMPLICIT statement is ignored.	Verify that ignoring the IMPLICIT statement does not affect the logic of the program.
321	F	INVALID TYPE IN IMPLICIT STATEMENT	The valid types are INTEGER, HALF PRECISION, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, BIT, and CHARACTER.	Correct error; recompile.
322	F	INVALID USE OF "*"	(self explanatory)	Correct error; recompile.
323	F	IMPLICIT RANGE IS INCORRECT	The characters specified in the range of an IMPLICIT statement must be in alphabetical order. A character cannot be associated with more than one type.	Arrange the characters in alphabetical order and eliminate duplicate specifications for characters; recompile.
324	F	NON-FORTRAN CHARACTER FOUND AND IS NOT IN HOLLERITH CHARACTER STRING	A character is used that is not in the FORTRAN 200 character set. These characters can be used only in Hollerith strings.	Replace the character with the appropriate character from the FORTRAN 200 character set; recompile.
325	F	SYNTAX ERROR AFTER A SYMBOLIC NAME IN VICINITY OF <<<<window>>>>	Self explanatory lexical error.	Correct error; recompile.
326	F	INVALID CHARACTER AFTER A ZERO IN VICINITY OF <<<<window>>>>	Self explanatory lexical error.	Correct error; recompile.
327	F	SYNTAX ERROR AFTER AN INTEGER CONSTANT IN VICINITY OF <<<<window>>>>	Self explanatory lexical error.	Correct error; recompile.
328	F	SYNTAX ERROR FOLLOWING A PERIOD IN VICINITY OF <<<<window>>>>	Self explanatory lexical error.	Correct error; recompile.
329	F	INVALID CHARACTER IN LOGICAL CONSTANT OR LOGICAL/ RELATIONAL OPERATOR IN VICINITY OF <<<<window>>>>	Self explanatory lexical error.	Correct error; recompile.
330	F	SYNTAX ERROR AFTER A REAL NUMBER IN VICINITY OF <<<<window>>>>	Self explanatory lexical error.	Correct error; recompile.
331	F	INVALID CHARACTER APPEARS IN THE NUMBER PART OF THE EXPONENT FIELD IN VICINITY OF <<<<window>>>>	Self explanatory lexical error.	Correct error; recompile.
332	W	TOO MANY DIGITS IN THE EXPONENT FIELD IN VICINITY OF <<<<window>>>>	The exponent field is truncated.	Verify that the truncation does not affect the logic of the program.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Type	Message	Significance	Action
333	F	SYNTAX ERROR FOLLOWING A SYMBOLIC STRING THAT WAS FOLLOWED BY A PERIOD IN VICINITY OF <<<window>>>	Self explanatory lexical error.	Correct error; recompile.
334	F	SYNTAX ERROR FOLLOWING A LOGICAL CONSTANT IN VICINITY OF <<<<window>>>>	Self explanatory lexical error.	Correct error; recompile.
335	F	SYNTAX ERROR FOLLOWING A REAL CONSTANT IN VICINITY OF <<<<window>>>>	Self explanatory lexical error.	Correct error; recompile.
336	F	SYNTAX ERROR FOLLOWING AN "*" IN VICINITY OF <<<<window>>>>	Self explanatory lexical error.	Correct error; recompile.
337	F	SYNTAX ERROR FOLLOWING A CHARACTER STRING IN VICINITY OF <<<<window>>>>	Self explanatory lexical error.	Correct error; recompile.
338	F	SYNTAX ERROR FOLLOWING A COMPLEX CONSTANT IN VICINITY OF <<<<window>>>>	Self explanatory lexical error.	Correct error; recompile.
339	F	SYNTAX ERROR IN A LABEL REFERENCE FIELD IN VICINITY OF <<<<window>>>>	Self explanatory lexical error.	Correct error; recompile.
340	F	SUBSCRIPT REFERENCE OUT OF RANGE	The subscript is less than the lower bound, or greater than the upper bound of the array.	Verify that the reference is intended.
341	F	DO LOOPS OR IF BLOCKS NESTED IMPROPERLY	Nested DO loops and IF blocks must appear entirely within outer DO loops and IF blocks.	Restructure the DO loops and IF blocks so that the nested DO loops are entirely within the outer DO loops and IF blocks; recompile.
342	F	DO-LOOP CONTROL VARIABLE USED IMPROPERLY	The variable used as the loop index cannot be altered within the range of the DO loop.	Remove all statements that alter the value of the loop index from the DO loop; recompile.
343	F	INVALID SYNTAX IN SUBARRAY REFERENCE IN VICINITY OF <<<<window>>>>	(self explanatory)	Correct error; recompile.
344	F	IMPLIED DO STRUCTURES DO NOT MATCH	Non-conformable arrays appear in an array assignment statement.	Correct error; recompile.
345	F	& DESCRIPTOR NOT ALLOWED TO EXIST ON READ STATEMENT	Only an output list can prefix a descriptor with an ampersand to transfer the descriptor value.	Use a DATA statement or an ASSIGN statement to set the descriptor value.
348	N	SYMBOLIC NAME HAS MORE THAN 6 CHARACTERS	A FORTRAN 200 symbolic name can have up to 8 characters. However, to satisfy ANSI standard X3.9, 1978, a symbolic name can consist of no more than 6 characters.	If the symbolic name is to meet ANSI standards, reduce the symbolic name to no more than 6 characters; recompile.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Type	Message	Significance	Action
349	F	TYPE OF OUTPUT ARGUMENT AND TYPE OF FUNCTION NAME DO NOT MATCH	(self explanatory)	Correct error; recompile.
350	F	INVALID OUTPUT ARGUMENT IN A FUNCTION REFERENCE	(self explanatory)	Correct error; recompile.
351	F	VECTOR EXPRESSION REQUIRES MORE TEMPORARIES, CODE CANNOT BE GENERATED	Compiler limitation exceeded.	Simplify statement; recompile.
352	F	VECTOR REFERENCE DATA ITEM USED FOR NON-DESCRIPTOR VARIABLE ITEM	Illegal vector reference in constant list in DATA statement.	Correct error; recompile.
353	F	CHARACTER CONSTANT CANNOT INITIALIZE A BIT VARIABLE	DATA statement cannot have character string as a value for a bit variable.	Correct error; recompile.
354	F	INVALID INITIALIZATION OF A CHARACTER OR BIT VARIABLE	A character variable may only be initialized with a character constant. A bit variable may only be initialized with a bit or hexadecimal constant.	Correct error; recompile.
355	W	CHARACTER CONSTANT TOO LONG - TRUNCATED ON THE RHS	The character constant is truncated on the right side. A character constant can contain no more than 255 characters.	Verify that the truncation does not affect the logic of the program.
356	W	HEX OR BIT CONSTANT TOO LONG - TRUNCATED ON THE LHS	The hexadecimal or bit constant is truncated on the left side. A hexadecimal or bit constant can contain no more than 255 characters.	Verify that the truncation does not affect the logic of the program.
357	F	BIT VARIABLES ARE NOT ALLOWED IN BUFFER IN/OUT	(self explanatory)	Correct error; recompile. Equivalence the bit variable to nonbit variables, and perform the I/O on the nonbit variables
358	F	INVALID DESCRIPTOR INITIALIZATION IN VICINITY OF <<<<window>>>>	(self explanatory)	Correct error; recompile.
359	W	VECTOR TYPE CHANGED TO BE SAME AS DESCRIPTOR IT INITIALIZES	Type conversion for vector in DATA statement.	Verify that this change does not affect the logic of the program.
360	N	NON-ANSI PROGRAM STATEMENT-- NO PARAMETER ALLOWED AFTER PROGRAM NAME	Declaring files or units for input/output operations after the program name in a program statement is illegal for ANSI FORTRAN.	Delete parameters following program name; recompile.
362	F	INVALID RIGHT-HAND SIDE FOR DESCRIPTOR ASSIGN	(self explanatory)	Correct error; recompile.
364	F	+-*/ ARE THE ONLY PERMITTED OPERATORS FOR COMPLEX VECTORS	Incorrect operator for complex vector expression.	Correct error; recompile.
365	F	ZERO ** ZERO OR NEGATIVE IS AN UNDEFINED ARITHMETIC EXPRESSION	The exponentiation of two zero constants produces an indefinite result.	Correct error; recompile.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Type	Message	Significance	Action
366	F	INVALID SUBSCRIPT IN IMPLIED DO IN VICINITY OF <<<<window>>>>	(self explanatory)	Correct error; recompile.
369	F	VARIABLE name APPEARS IN DESCRIPTOR STATEMENT MORE THAN ONCE	(self explanatory)	Correct error; recompile.
370	F	MISSING LABEL IN ARITHMETIC IF	An arithmetic IF must have three labels to which control can transfer depending on the condition. Labels can be duplicated.	Supply the missing label in the arithmetic IF statement; recompile.
371	A	JAM TEMP TABLE OVERFLOW	Compilation aborted.	Recompile without instruction scheduling.
372	F	EACH IMPLIED DO COMPONENT IN A SUBARRAY REFERENCE MUST BE TYPE INTEGER	(self explanatory)	Correct error; recompile.
373	F	INVALID SYMBOL IN EQUIVALENCE	A symbol equivalenced did not represent an array or simple variable.	Correct error; recompile.
374	F	TOO LITTLE DATA IN HEX OR BIT CONSTANT	The length of the bit constant must equal the length of the portion of the bit array being initialized.	Increase the length of the bit constant to the appropriate size; recompile.
375	F	TOO MUCH DATA IN HEX OR BIT CONSTANT	The length of the bit constant must equal the length of the portion of the bit array being initialized.	Decrease the length of the bit constant to the appropriate size; recompile.
376	A	NO EVEN-ODD REGISTER PAIR AVAILABLE	The compiler-generated code required an even-odd register pair and none was available.	Follow site-defined procedure.
377	W	INSTRUCTION SCHEDULING ABANDONED - REGISTER JAM	Compiler was unable to optimize instruction scheduling.	No action necessary; object code is generated.
378	F	THE COMMON BLOCK NAME AND AN ENTRY NAME ARE THE SAME	The name of a common block and the name of an entry point are the same.	Change one of the names; recompile.
379	F	SCALAR ARGUMENTS NOT ALLOWED IN Q8SDOT	(self explanatory)	Correct error; recompile.
380	F	RELATIVE BRANCH OUT OF RANGE	Branch too far in special call.	Correct error; recompile.
381	F	A SPECIAL CALL RELATIVE BRANCH MAY ONLY BRANCH TO A STATEMENT LABEL	Branching a constant number of halfwords is not permitted.	Correct error; recompile.
382	F	NON-ZERO OPERAND IN SPECIAL CALL FIELD THAT MUST BE NULL OR ZERO	Arguments are missing or in the wrong order.	Correct error; recompile.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Type	Message	Significance	Action
383	W	HOLLERITH CONSTANT TOO LONG - TRUNCATED ON RHS	The Hollerith constant is truncated on the right side. A Hollerith constant can have no more than 255 characters.	Verify that the truncation does not affect the logic of the program.
385	F	SUBROUTINE CONTAINS NON-STANDARD RETURN BUT NO * IN ARGUMENT LIST	Asterisks must appear in the argument list of the SUBROUTINE statement. Each asterisk must correspond to a statement label that appears in the argument list of the subroutine CALL statement.	Place asterisks in the appropriate positions in the argument list of the SUBROUTINE statement. Place statement labels in the appropriate positions in the CALL statements; recompile.
386	A	COMPILER FAILURE - IRRESOLVABLE REGISTER JAM	Compilation aborted.	Recompile without optimization.
387	W	R CONSTANT TOO LONG - TRUNCATED ON RHS	The R constant is truncated on the right side. An R constant can have no more than 255 characters.	Verify that the truncation does not affect the logic of the program.
388	F	HOLLERITH CONSTANT NOT PERMITTED IN SPECIAL CALL	(self explanatory)	Remove the Hollerith constant from the special call; recompile.
391	F	DUMMY ARGUMENT MAY NOT APPEAR IN EQUIVALENCE	(self explanatory)	Correct error; recompile.
392	F	MISSING SYMBOLIC NAME	Symbolic name missing from PARAMETER statement.	Supply the symbolic name; recompile.
393	F	MISSING "="	Equals sign missing from PARAMETER statement.	Supply the equals sign; recompile.
394	W	SYMBOLIC CONSTANT name PREVIOUSLY DECLARED	A symbolic constant is declared more than once. The first declaration was used.	Verify that the first declaration is intended.
395	F	SYMBOLIC CONSTANT name PREVIOUSLY USED FOR SOMETHING ELSE	A symbolic constant must not be the same as another symbol in the program.	Change the symbolic constant so that it is unique in the program; recompile.
396	F	VALUE MUST BE CONSTANT OR CONSTANT EXPRESSION	The value specified for a symbolic constant is not a constant.	Change the value to a constant or a constant expression; recompile.
397	F	INCOMPATIBLE TYPES FOR SYMBOLIC NAME AND ITS VALUE	(self explanatory)	Correct error; recompile.
398	W	PARAMETER STATEMENTS MUST PRECEDE DATA STATEMENTS	(self explanatory)	Move the PARAMETER statement in front of the DATA statement; recompile.
399	W	PARAMETER STATEMENTS MUST PRECEDE STATEMENT FUNCTION DEFINITIONS	(self explanatory)	Move the PARAMETER statement in front of the statement function definitions; recompile.
400	W	PARAMETER STATEMENTS MUST PRECEDE EXECUTABLE STATEMENTS	(self explanatory)	Move the PARAMETER statement in front of all executable statements; recompile.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Type	Message	Significance	Action
401	F	MISUSE OF SYMBOLIC CONSTANT name	A symbolic constant can be used like any other constant, except it cannot appear in a complex constant, in a FORMAT statement, or in a PROGRAM statement. Also, it cannot appear as input data.	Correct error; recompile.
404	F	DUPLICATE OR CONFLICTING IMPLICIT TYPE	A letter must not be assigned more than one implicit type.	Correct error; recompile.
405	F	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	INVALID BLOCK IF NESTING	A nested block IF must be entirely contained in an outer block IF.	Correct error; recompile.
407	A	COMPILER FAILURE -- INVALID VARIABLE TYPE DETECTED IN I/O LIST	Typeless or hex variable is found in I/O list.	Follow site-defined procedure.
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.
411	W	MISSING THEN IN ELSE IF STATEMENT	The keyword THEN must follow the keyword ELSE IF.	Supply the missing THEN.
412	A	COMPILER FAILURE - INVALID REGISTER TYPE DETECTED DURING REGISTER MAP PROCESSING	(self explanatory)	Follow site-defined procedure. Remove "A" list option.
413	A	COMPILER FAILURE -- INVALID PHASE DETECTED DURING REGISTER MAP PROCESSING	(self explanatory)	Follow site-defined procedure. Remove "A" list option.
414	F	INAPPROPRIATE NAME FOLLOWED BY LEFT PARENTHESIS IN VICINITY OF <<<<window>>>>	(self explanatory)	Correct error; recompile.
415	F	MISUSE OF LEFT PARENTHESIS IN VICINITY OF <<<<window>>>>	(self explanatory)	Correct error; recompile.
416	F	MISUSE OF COMMA IN VICINITY OF <<<<window>>>>	(self explanatory)	Correct error; recompile.
417	F	MISUSE OF SEMICOLON IN VICINITY OF <<<<window>>>>	(self explanatory)	Correct error; recompile.
418	F	MISUSE OF COLON IN VICINITY OF <<<<window>>>>	(Self explanatory)	Correct error; recompile.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Type	Message	Significance	Action
419	F	MISUSE OF CONCATENATION OPERATOR IN VICINITY OF <<<<window>>>>	(Self explanatory)	Correct error; recompile.
420	F	MISUSE OF ".NOT." OPERATOR IN VICINITY OF <<<<window>>>>	(self explanatory)	Correct error; recompile.
421	F	MISUSE OF RELATIONAL OPERATOR IN VICINITY OF <<<<window>>>>	(self explanatory)	Correct error; recompile.
422	F	MISUSE OF LOGICAL OPERATOR IN VICINITY OF <<<<window>>>>	(self explanatory)	Correct error; recompile.
423	F	TWO RELATIONAL OPERATORS IN A ROW IN VICINITY OF <<<<window>>>>	(self explanatory)	Correct error; recompile.
424	F	RIGHT PARENTHESIS FOLLOWED BY LEFT PARENTHESIS WITH NO INTERVENING OPERATOR IN VICINITY OF <<<<window>>>>	(self explanatory)	Correct error; recompile.
425	F	INCORRECT USE OF CHARACTER SUBSTRING IN VICINITY OF <<<<window>>>>	(self explanatory)	Correct error; recompile.
426	F	MISSING OPERATOR AFTER RIGHT PARENTHESIS IN VICINITY OF <<<<window>>>>	(self explanatory)	Correct error; recompile.
427	F	INVALID SUBSTRING NOTATION OR SUBSCRIPT/SUBSTRING REVERSED IN VICINITY OF <<<<window>>>>	Syntax incorrect for substring reference.	Correct error; recompile.
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	INVALID 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.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Type	Message	Significance	Action
434	F	INVALID 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 invalid 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.
436	F	MISSING OPERAND IN VICINITY OF <<<<window>>>>	(self explanatory)	Correct error; recompile.
437	F	MISUSE OF LOGICAL EXPRESSION IN VICINITY OF <<<<window>>>>	(self explanatory)	Correct error; recompile.
438	W	SUBSCRIPT LESS THAN LOWER DIMENSION BOUND	(self explanatory)	Correct error; recompile.
439	W	SUBSCRIPT GREATER THAN UPPER DIMENSION BOUND	(self explanatory)	Correct error; recompile.
440	F	INTRINSIC STATEMENTS ARE NOT ALLOWED IN BLOCK DATA SUBPROGRAMS	(self explanatory)	Remove INTRINSIC statements from block data subprogram; recompile.
441	F	INTRINSIC STATEMENTS MUST PRECEDE ALL EXECUTABLE STATEMENTS	(self explanatory)	Correct error; recompile.
442	F	EXPECTED INTRINSIC FUNCTION NAME - FOUND name	Something other than an intrinsic function name appears in an INTRINSIC statement.	Correct error or remove name from INTRINSIC statement; recompile.
443	F	SUBSTRING NAME name IS NOT TYPE CHARACTER	The variable must be type character.	Correct error; recompile.
444	N	INTRINSIC FUNCTION name IS NON-ANSI	The function is not in the list of intrinsic functions in ANSI Standard X3.9, 1978.	Declare it EXTERNAL; recompile.
445	F	ILLEGAL DELIMITER IN A SUBSTRING REFERENCE IN VICINITY OF <<<<window>>>>	The correct form is "("expression":"expression")".	Correct error; recompile.
446	F	NON-SCALAR SUBSCRIPT IN VICINITY OF <<<<window>>>>	A vector or descriptor must not be used as an array subscript.	Change subscript to a scalar; recompile.
448	F	ADJUSTABLE LENGTH CHARACTER FUNCTIONS DO NOT EXIST	Invalid character type declaration.	Correct error; recompile.
449	W	LENGTH EXPRESSION FOR CHARACTER VARIABLE(S) CONVERTED TO TYPE INTEGER	Invalid character type declaration.	Verify that conversion results in proper length specification.
450	F	EXPRESSION IN SUBSTRING IS NOT TYPE INTEGER	(self explanatory)	Correct error; recompile.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Type	Message	Significance	Action
451	F	EXECUTABLE LABEL WAS PREVIOUSLY REFERENCED AND/OR DEFINED AS A FORMAT LABEL	The label of an executable statement is referenced or defined as the label of a nonexecutable FORMAT statement.	Correct error; recompile.
452	F	FORMAT LABEL WAS PREVIOUSLY REFERENCED AND/OR DEFINED AS AN EXECUTABLE LABEL	The label of a nonexecutable FORMAT statement is referenced or defined as the label of an executable statement.	Correct error; recompile.
453	W	FORMAT LABEL MIGHT HAVE BEEN ASSIGNED TO GOTO VARIABLE name	Control must not transfer to a nonexecutable statement, such as a FORMAT statement.	Verify that the statement label assignment statement assigns the label of an executable statement to the assigned GOTO variable.
454	W	EXECUTABLE LABEL MIGHT HAVE BEEN ASSIGNED TO FORMAT VARIABLE name	A format specification must not reference an executable statement.	Verify that the statement label assignment statement assigns the label of a FORMAT statement to the format variable.
455	F	FORMAT VARIABLE name WAS NEVER ASSIGNED TO A FORMAT LABEL	The format variable is referenced before a label is assigned to it by a statement label assignment statement.	Provide a statement label assignment statement for the format label; recompile.
456	F	ARGUMENT OF GENERIC FUNCTION name HAS WRONG TYPE	The argument of the generic function is of the wrong data type.	Change the type of the argument; recompile.
457	F	CONFLICTING USES OF INTRINSIC FUNCTION NAME name	(self explanatory)	Correct error; recompile.
459	F	TOO FEW ARGUMENTS FOR INTRINSIC FUNCTION name	Additional arguments are required.	Provide proper number of arguments; recompile.
460	F	SYNTAX ERROR IN ARGUMENT LIST	The argument list does not use correct syntax.	Refer to the argument list definition within the program or, if predefined, within the documentation.
462	F	WRONG TYPE FOR ARGUMENT OF INTRINSIC FUNCTION name	The argument of the intrinsic function is of the wrong data type.	Change the type of the argument; recompile.
463	F	WRONG NUMBER OF ARGUMENTS FOR INTRINSIC FUNCTION name	(self explanatory)	Provide proper number of arguments; recompile.
464	F	STATEMENT FUNCTION NAMES CANNOT BE USED AS ACTUAL ARGUMENTS	Statement functions can be referenced only from the program unit in which they are defined.	Remove statement function name from argument list; recompile.
465	A	CONFLICTING USES OF INTRINSIC FUNCTION name'S SLOW ENTRY POINT NAME	(self explanatory)	Follow site-defined procedure.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Type	Message	Significance	Action
466	F	RESULT SPECIFICATION MISSING FROM VECTOR INTRINSIC FUNCTION name	A vector output argument must be the last item in the argument list of a vector intrinsic function and must be preceded by a semicolon.	Provide a vector output argument; recompile.
467	F	SEMICOLON NOT PERMITTED IN REFERENCE TO INTRINSIC FUNCTION name	(self explanatory)	Correct error; recompile.
468	F	ASSUMED LENGTH CHARACTER VARIABLE CANNOT BE CONCATENATED	A character variable whose declared length is * cannot be included in a character expression.	Use a character variable whose length is known.
470	F	INTRINSIC FUNCTION name CANNOT BE USED AS AN ACTUAL ARGUMENT	(self explanatory)	Correct error; recompile.
471	N	INTRINSIC FUNCTION name CANNOT BE USED AS AN ACTUAL ARGUMENT IN STANDARD CONFORMING PROGRAMS	This function usage is not permitted under ANSI Standard X3.9, 1978.	If it is a user supplied routine, declare it EXTERNAL and recompile.
472	F	EXTERNAL PROCEDURE name USED AS AN ACTUAL ARGUMENT NOT DECLARED IN AN EXTERNAL STATEMENT	In order to pass an external function or subroutine as an actual argument, you must declare its name in an EXTERNAL statement.	Declare the function or subroutine name in an EXTERNAL statement; recompile.
473	F	INTRINSIC FUNCTION name USED AS AN ACTUAL ARGUMENT NOT DECLARED IN INTRINSIC STATEMENT	In order to pass an intrinsic function as an actual argument, you must declare its name in an INTRINSIC statement.	Declare the intrinsic function name in an INTRINSIC statement; recompile.
474	A	COMPILER FAILURE - name CANNOT BE A BASIC EXTERNAL	(self explanatory)	Follow site-defined procedure.
475	F	NAMELIST GROUP NAME name CANNOT BE USED AS AN ACTUAL ARGUMENT	A namelist group name cannot be passed as an argument; instead, it must be explicitly defined in a NAMELIST statement in each program unit that uses it.	Remove namelist group name from argument list; recompile.
476	F	ENTRY POINT NAME name CANNOT BE USED AS AN ACTUAL ARGUMENT	(self explanatory)	Correct error; recompile.
477	F	PROGRAM NAME name CANNOT BE USED AS AN ACTUAL ARGUMENT	(self explanatory)	Correct error; recompile.
478	A	COMPILER FAILURE - ATTEMPT TO USE BLOCK DATA NAME name AS AN ACTUAL ARGUMENT	(self explanatory)	Follow site-defined procedure.
479	W	INTRINSIC FUNCTION name WAS TYPED -- TYPING IGNORED	Intrinsic function types cannot be affected by type specification statements.	No action required.
480	A	COMPILER FAILURE - VALUE OF NAME CLASS OUT OF RANGE	(self explanatory)	Follow site-defined procedure.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Type	Message	Significance	Action
481	F	DUMMY ARGUMENT name CANNOT BE USED AS A STATEMENT FUNCTION NAME	Statement functions cannot have the same name as a dummy argument of the subprogram in which the statement function is defined.	Change statement function name or argument name; recompile.
483	A	COMPILER FAILURE --- GFN_LIST FOR GENERIC name IS EMPTY	(self explanatory)	Follow site-defined procedure.
484	F	name CANNOT BE USED AS A STATEMENT FUNCTION NAME	(self explanatory)	Correct error; recompile.
485	F	FORWARD REFERENCE TO STATEMENT FUNCTION name NOT ALLOWED	A statement function must be defined before it is referenced.	Move statement function definition in front of reference; recompile.
486	F	LENGTH OF A STATEMENT FUNCTION DUMMY CHARACTER ARGUMENT name CANNOT BE ASSUMED	The length of a dummy argument appearing in a FUNCTION statement must be specified explicitly.	Correct error; recompile.
487	N	ONLY VARIABLES MAY BE USED AS STATEMENT FUNCTION DUMMY ARGUMENTS	Something other than a variable is used as a dummy argument in a statement function definition.	Change dummy argument to a variable; recompile.
488	F	name MAY ONLY APPEAR ONCE IN STATEMENT FUNCTION DUMMY ARGUMENT LIST	The name appears more than once in the dummy argument list of a statement function definition.	Correct error; recompile.
489	F	ERROR IN STATEMENT FUNCTION DUMMY ARGUMENT LIST	The dummy argument list of a statement function definition is invalid.	Correct error; recompile.
490	F	SYNTAX ERROR IN STATEMENT FUNCTION DEFINITION	(self explanatory)	Correct error; recompile.
491	F	ERROR IN STATEMENT FUNCTION STATEMENT EXPRESSION	(self explanatory)	Correct error; recompile.
492	F	TOO MANY ARGUMENTS FOR STATEMENT FUNCTION name	The number of actual and dummy arguments must be the same.	Correct error; recompile.
493	F	TOO FEW ARGUMENTS FOR STATEMENT FUNCTION name	The number of actual and dummy arguments must be the same.	Correct error; recompile.
494	F	SYNTAX ERROR IN REFERENCE TO STATEMENT FUNCTION name	(self explanatory)	Correct error; recompile.
495	F	A TYPE CONVERSION IS NOT POSSIBLE FOR AN ACTUAL ARGUMENT OF STATEMENT FUNCTION name	Type of actual argument not as expected.	Correct error; recompile.
496	A	COMPILER FAILURE - name IS NOT A STATEMENT FUNCTION	(self explanatory)	Follow site-defined procedure.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Type	Message	Significance	Action
497	F	RECURSIVE STATEMENT FUNCTION DEFINITION	A statement function directly or indirectly references itself.	Correct error; recompile.
498	F	MISSING ARGUMENT IN STATEMENT FUNCTION REFERENCE	The number of actual and dummy arguments must be the same.	Correct error; recompile.
499	F	VECTOR FUNCTION NAME MUST APPEAR IN A DESCRIPTOR STATEMENT WITHIN THE BODY OF THE FUNCTION	One of the following: 1. Vector function name is not in any descriptor statement, or 2. The descriptor statement containing the vector function name is not within the body of the function.	Correct error; recompile.
500	F	SYNTAX ERROR IN AN OPEN STATEMENT IN VICINITY OF <<<<window>>>>	(self explanatory)	Correct error; recompile.
501	F	SYNTAX ERROR IN A CLOSE STATEMENT IN VICINITY OF <<<<window>>>>	(self explanatory)	Correct error; recompile.
503	F	SYNTAX ERROR IN AN INQUIRE STATEMENT IN VICINITY OF <<<<window>>>>	(self explanatory)	Correct error; recompile.
505	F	SYNTAX ERROR IN A FILE POSITIONING STATEMENT IN VICINITY OF <<<<window>>>>	(self explanatory)	Correct error; recompile.
506	F	param IS NOT ALLOWED	(self explanatory)	Correct error; recompile.
507	F	VECTOR LENGTH CANNOT EXCEED 65535	The maximum vector length is 65535 elements.	Correct error; recompile.
508	F	SCALAR FUNCTION NAME CANNOT APPEAR IN A DESCRIPTOR STATEMENT	(self explanatory)	Correct error; recompile.
509	F	A VECTOR CANNOT BE USED AS AN ACTUAL ARGUMENT IN A STATEMENT FUNCTION REFERENCE	An actual argument in a statement function reference must be a scalar expression.	Correct error; recompile.
510	F	MORE THAN ONE UNIT SPECIFIER IN param STATEMENT	(self explanatory)	Correct error; recompile.
511	F	MORE THAN ONE FORMAT SPECIFIER IN param STATEMENT	(self explanatory)	Correct error; recompile.
512	F	MORE THAN ONE "REC=" SPECIFIER IN param STATEMENT	(self explanatory)	Correct error; recompile.
513	F	MORE THAN ONE "END=" SPECIFIER IN param STATEMENT	(self explanatory)	Correct error; recompile.
514	F	MORE THAN ONE "ERR=" SPECIFIER IN param STATEMENT	(self explanatory)	Correct error; recompile.
515	F	MORE THAN ONE "IOSTAT=" SPECIFIER IN param STATEMENT	(self explanatory)	Correct error; recompile.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Type	Message	Significance	Action
516	F	MORE THAN ONE "FILE=" SPECIFIER IN param STATEMENT	(self explanatory)	Correct error; recompile.
517	F	MORE THAN ONE "ACCESS=" SPECIFIER IN param STATEMENT	(self explanatory)	Correct error; recompile.
518	F	MORE THAN ONE "SEQUENTIAL=" SPECIFIER IN param STATEMENT	(self explanatory)	Correct error; recompile.
519	F	MORE THAN ONE "DiRECT=" SPECIFIER IN param STATEMENT	(self explanatory)	Correct error; recompile.
520	F	MORE THAN ONE "FORM=" SPECIFIER IN param STATEMENT	(self explanatory)	Correct error; recompile.
521	F	MORE THAN ONE "FORMATTED=" SPECIFIER IN param STATEMENT	(self explanatory)	Correct error; recompile.
522	F	MORE THAN ONE "UNFORMATTED=" SPECIFIER IN param STATEMENT	(self explanatory)	Correct error; recompile.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Type	Message	Significance	Action
523	F	MORE THAN ONE "RECL=" SPECIFIER IN param STATEMENT	(self explanatory)	Correct error; recompile.
524	F	MORE THAN ONE "BUFS=" SPECIFIER IN param STATEMENT	(self explanatory)	Correct error; recompile.
525	F	MORE THAN ONE "NEXTREC=" SPECIFIER IN param STATEMENT	(self explanatory)	Correct error; recompile.
526	F	MORE THAN ONE "NUMBER=" SPECIFIER IN param STATEMENT	(self explanatory)	Correct error; recompile.
527	F	MORE THAN ONE "NAME=" SPECIFIER IN param STATEMENT	(self explanatory)	Correct error; recompile.
528	F	MORE THAN ONE "EXIST=" SPECIFIER IN param STATEMENT	(self explanatory)	Correct error; recompile.
529	F	MORE THAN ONE "OPENED=" SPECIFIER IN param STATEMENT	(self explanatory)	Correct error; recompile.
530	F	MORE THAN ONE "BLANK=" IN param STATEMENT	(self explanatory)	Correct error; recompile.
531	F	MORE THAN ONE "STATUS=" SPECIFIER IN param STATEMENT	(self explanatory)	Correct error; recompile.
532	F	MORE THAN ONE "NAMED=" SPECIFIER IN param STATEMENT	(self explanatory)	Correct error; recompile.
535	F	UNIT SPECIFIER IN param STATEMENT MUST BE AN INTEGER	(self explanatory)	Correct error; recompile.
536	F	"REC=" SPECIFIER IN param STATEMENT MUST BE AN INTEGER	(self explanatory)	Correct error; recompile.
537	F	"END=" SPECIFIER IN param STATEMENT MUST BE AN EXECUTABLE STATEMENT NUMBER	(self explanatory)	Correct error; recompile.
538	F	"ERR=" SPECIFIER IN param STATEMENT MUST BE AN EXECUTABLE STATEMENT NUMBER	(self explanatory)	Correct error; recompile.
539	F	"IOSTAT=" SPECIFIER IN param STATEMENT MUST BE AN INTEGER VARIABLE ARRAY ELEMENT	(self explanatory)	Correct error; recompile.
540	F	"FILE=" SPECIFIER IN param STATEMENT MUST BE A CHARACTER EXPRESSION	(self explanatory)	Correct error; recompile.
541	F	"STATUS=" SPECIFIER IN param STATEMENT MUST BE A CHARACTER EXPRESSION	(self explanatory)	Correct error; recompile.
542	F	"ACCESS=" SPECIFIER IN param STATEMENT MUST BE A CHARACTER EXPRESSION	(self explanatory)	Correct error; recompile.
543	F	"ACCESS=" SPECIFIER IN param STATEMENT MUST BE A CHARACTER VARIABLE ARRAY ELEMENT, OR SUBSTRING	(self explanatory)	Correct error; recompile.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Type	Message	Significance	Action
544	F	"SEQUENTIAL=" SPECIFIER IN param STATEMENT MUST BE A CHARACTER VARIABLE, ARRAY ELEMENT OR SUBSTRING	(self explanatory)	Correct error; recompile.
545	F	"DIRECT=" SPECIFIER IN param STATEMENT MUST BE A CHARACTER VARIABLE ARRAY ELEMENT, OR SUBSTRING	(self explanatory)	Correct error; recompile.
546	F	"FORM=" SPECIFIER IN param STATEMENT MUST BE A CHARACTER EXPRESSION	(self explanatory)	Correct error; recompile.
547	F	"FORM=" SPECIFIER IN param STATEMENT MUST BE A CHARACTER VARIABLE ARRAY ELEMENT, OR SUBSTRING	(self explanatory)	Correct error; recompile.
548	F	"FORMATTED=" SPECIFIER IN param STATEMENT MUST BE A CHARACTER VARIABLE, ARRAY ELEMENT, OR SUBSTRING	(self explanatory)	Correct error; recompile.
549	F	"UNFORMATTED=" SPECIFIER IN param STATEMENT MUST BE A CHARACTER VARIABLE, ARRAY ELEMENT, OR SUBSTRING	(self explanatory)	Correct error; recompile.
550	F	"RECL=" SPECIFIER IN param STATEMENT MUST BE AN INTEGER EXPRESSION	(self explanatory)	Correct error; recompile.
551	F	"RECL=" SPECIFIER IN param STATEMENT MUST BE AN INTEGER VARIABLE OR ARRAY ELEMENT	(self explanatory)	Correct error; recompile.
552	F	"BUFS=" SPECIFIER IN param STATEMENT MUST BE AN INTEGER EXPRESSION	(self explanatory)	Correct error; recompile.
553	F	"BUFS=" SPECIFIER IN param STATEMENT MUST BE AN INTEGER OR ARRAY ELEMENT VARIABLE	(self explanatory)	Correct error; recompile.
554	F	"NEXTREC=" SPECIFIER IN param STATEMENT MUST BE AN INTEGER EXPRESSION	(self explanatory)	Correct error; recompile.
555	F	"NUMBER=" SPECIFIER IN param STATEMENT MUST BE AN INTEGER VARIABLE OR ARRAY ELEMENT	(self explanatory)	Correct error; recompile.
556	F	"NAME=" SPECIFIER IN param STATEMENT MUST BE A CHARACTER VARIABLE ARRAY ELEMENT, OR SUBSTRING	(self explanatory)	Correct error; recompile.
557	F	"NAMED=" SPECIFIER IN param STATEMENT MUST BE A LOGICAL VARIABLE OR ARRAY ELEMENT	(self explanatory)	Correct error; recompile.
558	F	"EXIST=" SPECIFIER IN param STATEMENT MUST BE A LOGICAL VARIABLE OR ARRAY ELEMENT	(self explanatory)	Correct error; recompile.
559	F	"OPENED=" SPECIFIER IN param STATEMENT MUST BE A LOGICAL VARIABLE OR ARRAY ELEMENT	(self explanatory)	Correct error; recompile.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Type	Message	Significance	Action
560	F	"BLANK=" SPECIFIER IN param STATEMENT MUST BE A CHARACTER EXPRESSION	(self explanatory)	Correct error; recompile.
565	N	PERFORMED MODE CONVERSION OF UNIT SPECIFIER TO TYPE INTEGER IN param STATEMENT	(self explanatory)	Surround expression with an INT function call, or otherwise repair it; recompile.
566	N	PERFORMED MODE CONVERSION OF "RECL=" SPECIFIER TO TYPE INTEGER IN param STATEMENT	(self explanatory)	Surround expression with an INT function call, or otherwise repair it; recompile.
567	N	PERFORMED MODE CONVERSION OF "BUFS=" SPECIFIER TO TYPE INTEGER IN param STATEMENT	(self explanatory)	Surround expression with an INT function call, or otherwise repair it; recompile.
568	N	PERFORMED MODE CONVERSION OF "REC=" SPECIFIER TO TYPE INTEGER IN param STATEMENT	(self explanatory)	Surround expression with an INT function call, or otherwise repair it; recompile.
570	F	AN INQUIRE STATEMENT MUST HAVE EITHER A UNIT SPECIFIER OR A "FILE=" SPECIFIER, BUT NOT BOTH	(self explanatory)	Correct error; recompile.
571	N	ASSIGN A CHARACTER CONSTANT TO A NON-CHARACTER VARIABLE IS NON-ANSI	(self explanatory)	Correct error; recompile.
572	N	NUMBER OF SUBSCRIPTS DOES NOT MATCH NUMBER OF DIMENSIONS DECLARED	(self explanatory)	Correct error; recompile.
573	F	VECTOR LENGTH FOR COMPLEX OR DOUBLE PRECISION VARIABLES CANNOT EXCEED 32767	(self explanatory)	Correct error; recompile.
577	F	BIT VECTOR FUNCTION DECLARED IN THE FORM OF A SCALAR FUNCTION IS ILLEGAL	Vector function declaration requires a nonempty argument list followed by the character string (;*) as illustrated in figure 9-36.	Correct error; recompile.
578	N	CHARACTER AND NONCHARACTER VARIABLES CANNOT BE EQUIVALENCED	ANSI FORTRAN permits only character to character equivalencing or real, integer, logical, double precision, and complex equivalencing.	Split equivalence groups into character and noncharacter groups; recompile.
580	F	ADJUSTABLE LENGTH SPECIFICATION name IS NOT IN COMMON, OR IN ARGUMENT LIST WITH CHARACTER VARIABLE	If, in a subroutine, an integer variable is used instead of a constant to specify a character dummy argument length, the integer variable must either be in common or be in a dummy argument in the same list as the character dummy argument.	Correct error; recompile.

TABLE B-1. COMPILER ERROR MESSAGES (Contd)

Error Number	Error Type	Message	Significance	Action
581	F	ADJUSTABLE LENGTH CHARACTER VARIABLE name IS NOT A DUMMY ARGUMENT	(self explanatory)	Correct error; recompile
582	A	COMPILER FAILURE - BAD REGISTER MANAGER REQUEST	(self explanatory)	Follow site-defined procedure.
584	A	COMPILER FAILURE - FAST CALL CONFUSION	(self explanatory)	Follow site-defined procedure.
585	F	VECTOR FUNCTION name RESULT LENGTH MUST BE INTEGER EXPRESSION	(self explanatory)	Correct error; recompile.
586	A	COMPILER FAILURE - ARGUMENT MODE MISMATCH FOR VECTOR FUNCTION name	(self explanatory)	Follow site-defined procedure.
587	F	ARRAY NAME CANNOT BE USED AS ACTUAL ARGUMENT OF name	(self explanatory)	Correct error; recompile.
588	F	EXTERNAL PROCEDURE NAME CANNOT BE USED AS ACTUAL ARGUMENT OF name	(self explanatory)	Correct error; recompile.
589	F	INTRINSIC FUNCTION NAME CANNOT BE USED AS ACTUAL ARGUMENT OF name	(self explanatory)	Correct error; recompile.
591	F	EXTERNAL SYMBOL CANNOT BE USED AS ACTUAL ARGUMENT OF name	(self explanatory)	Correct error; recompile.
592	F	DESCRIPTOR ARRAY NAME CANNOT BE USED AS ACTUAL ARGUMENT OF name	The named function requires an argument of another type.	Correct error; recompile.

TABLE B-2. EXECUTION TIME ERROR MESSAGES

Error Number	Error Level	Message	Significance	Action
001	C	SIL DETECTED ERROR	Followed by the SIL error message text. SIL error messages are explained in the OS Reference Manual.	Proper action is determinedby the SIL message.
002	C	ANOTHER I/O STATEMENT IS ALREADY IN PROGRESS	A function that does I/O has been called in an I/O list.	Remove I/O statement from function, or move function call outside I/O statement.
003	F	MAXIMUM INTERNAL FILE LENGTH EXCEEDED.	Attempted to read or write beyond the end of an internal file.	Reduce the number of records being referenced by the internal I/O state ment, or use an "END=" specifier on input.
004	C	"END=" IS NOT PERMITTED IN AN OUTPUT CILIST.	(self explanatory)	Correct error; recompile.
005	F	HOLLERITH CONVERSION ON INPUT IS IMPOSSIBLE	ANSI Standard FORTRAN '77 does not provide for an H descriptor in a FORMAT used for input.	Correct error; recompile.
006	C	SEQUENTIAL I/O MAY NOT SPECIFY A RECORD NUMBER	The compiler generated a sequential I/O call, but indicated direct access by specifying REC=.	Follow site-defined procedure.
007	C	ORDER OF LIBRARY CALLS DID NOT CONFORM TO CORRECT COMPILER GENERATED SEQUENCE	I/O library calls were made out of sequence. This should not happen if all I/O is done with FORTRAN routines.	Follow site-defined procedure.
008	F	REMAINDER OF FORMAT CONTAINS NO DATA CONVERSION EDIT DESCRIPTORS	The FORMAT was exhausted before the end of the I/O list. The repeatable part of the FORMAT cannot convert data.	Make the FORMAT and I/O agree with each other on the number of items to be read/written; rerun.

TABLE B-2. EXECUTION TIME ERROR MESSAGES (Contd)

Error Number	Error Level	Message	Significance	Action
009	F	VALUE GIVEN FOR DO VARIABLE TYPE IS NOT ACCEPTABLE	I/O list implied DO variables must be integer, half precision, real or double precision. This could indicate a compiler failure.	Follow site-defined procedure.
010	C	VALUE GIVEN FOR INTEGER UNIT IDENTIFIER IS OUTSIDE THE RANGE 0 ... 999	An integer unit number must be a one to three digit non-negative number.	Correct error; rerun.
011	C	ATTEMPT TO OPEN MORE THAN 2048 UNITS	2048 is maximum number of distinct units that may be opened in one program execution.	Correct error; rerun.
012	C	ATTEMPT TO OPEN MORE THAN 70 FILES	70 is maximum number of files that may be open at the same time.	Correct error; rerun.
013	C	ATTEMPT TO USE A UNIT ALREADY CLOSED	You are attempting to use a closed unit without first reopening it.	Correct error; rerun.
014	C	FILE ORGANIZATION CONFLICTS WITH ACCESS SPECIFIER	Direct access I/O is allowed only on direct access files, and sequential access I/O only on sequential files.	Use a file whose organization supports the type of I/O you wish to do.
015	C	ATTEMPT TO DECLARE AN ALTERNATE UNIT FOR A NONEXISTENT FILE	The program's execution control statement has an alternate unit specifier for an undeclared file.	Declare the file in the control statement or on the PROGRAM statement (if the ** form is not used).
016	C	INVALID PROGRAM STATEMENT PARAMETER TABLE	Format of data passed from main program is wrong. This would be an extraordinary error in FORTRAN main programs.	Follow site-defined procedure.
017	C	UNMATCHED PARENTHESES	The execution control statement is incorrectly formatted.	Either surround the list of preconnection specifiers and RLP specifiers with parentheses or terminate the list with a period or carriage return.
018	F	UNRECOGNIZABLE PARAMETER ENCOUNTERED IN Q7DFSET	A call to Q7DFSET has incorrect arguments.	Correct error; rerun.
019	F	UNRECOGNIZABLE PARAMETER ENCOUNTERED IN Q7DFCL1	A call to Q7DFCL1 has incorrect arguments.	Correct error; rerun.
020	F	UNRECOGNIZABLE PARAMETER ENCOUNTERED IN Q7DFOFF	A call to Q7DFOFF has incorrect arguments.	Correct error; rerun.
021	C	ROUTINES CALLING Q7DFSET NESTED TOO DEEPLY	You have called Q7DFSET from 101 different subroutine levels.	Correct error; rerun.

TABLE B-2. EXECUTION TIME ERROR MESSAGES (Contd)

Error Number	Error Level	Message	Significance	Action
022	W	DATA FLAG BRANCH - ORX - REGISTER 1 ADDRESS	You used Q7DFSET to set data flag register bit 28. There is a divide fault, exponent overflow, or result machine zero. Register 1 address indicates a location encountered after the condition.	Prevent the problem, or use Q7DFSET to specify a routine to handle the problem or don't use Q7DFSET to set data flag register bit 28.
023	W	DATA FLAG BRANCH - ORD - REGISTER 1 ADDRESS	You used Q7DFSET to set data flag register bit 24. The selected condition is not met, or there is a decimal arithmetic fault. Register 1 address indicates a location encountered after the condition.	Prevent the problem, or use Q7DFSET to specify a routine to handle the problem or don't use Q7DFSET to set data flag register bit 24.
024	F	DATA FLAG BRANCH - IMAGINARY SQUARE ROOT - REGISTER 1 ADDRESS	Data flag register bits 13, 29, and 45 are set. You are attempting to solve for square root of a negative number. Register 1 address indicates a location encountered after the condition.	Prevent the problem, or use Q7DFSET to specify a routine to handle the problem or don't use Q7DFSET to set data flag register bit 24.
025	F	DATA FLAG BRANCH - INDEFINITE RESULT - REGISTER 1 ADDRESS	Data flag register bits 14, 30, and 46 are set. There is an indefinite result or operand. Register 1 address indicates a location encountered after the condition.	Prevent the problem, or use Q7DFSET to specify a routine to handle the problem or to reset data flag register bit 30.
026	F	DATA FLAG BRANCH - ZERO DIVISOR - REGISTER 1 ADDRESS	Data flag register 9, 25, and 41 are set. This is an attempted division by zero. Register 1 address indicates a location encountered after the condition.	Prevent the problem, or use Q7DFSET to specify a routine to handle the problem or to reset data flag register bit 25.
027	W	DATA FLAG BRANCH - EXO - REGISTER 1 ADDRESS	You used Q7DFSET to set data flag register bit 26. There is a floating point exponent overflow. Register 1 address indicates a location encountered after the condition.	Prevent the problem, or use Q7DFSET to specify a routine to handle the problem or don't use Q7DFSET to set data flag register bit 26.
028	W	DATA FLAG BRANCH - RMZ - REGISTER 1 ADDRESS	You used Q7DFSET to set data flag register bit 27. Some operation's result was zero. Register 1 address indicates a location encountered after the condition.	Prevent the problem, or use Q7DFSET to specify a routine to handle the problem or don't use Q7DFSET to set data flag register bit 27.
029	W	DATA FLAG BRANCH - SSC - REGISTER 1 ADDRESS	You used Q7DFSET to set data flag register bit 21. The selected condition was not met. Register 1 address indicates a location encountered after the condition.	Prevent the problem, or use Q7DFSET to specify a routine to handle the problem or don't use Q7DFSET to set data flag register bit 21.

TABLE B-2. EXECUTION TIME ERROR MESSAGES (Contd)

Error Number	Error Level	Message	Significance	Action
030	W	DATA FLAG BRANCH - DDF - REGISTER 1 ADDRESS	You used Q7DFSET to set data flag register bit 22. A decimal instruction received bad data. Register 1 address indicates a location encountered after the condition.	Prevent the problem, or use Q7DFSET to specify a routine to handle the problem or don't use Q7DFSET to set data flag register bit 22.
031	W	DATA FLAG BRANCH - TBZ - REGISTER 1 ADDRESS	You used Q7DFSET to set data flag register bit 23. There is decimal arithmetic overflow. Register 1 address indicates a location encountered after the condition.	Prevent the problem, or use Q7DFSET to specify a routine to handle the problem or don't use Q7DFSET to set data flag register bit 23.
032	C	CLASS I DATA FLAG BRANCH - NO INTERRUPT ROUTINE PROVIDED - REGISTER 1 ADDRESS	Software Interrupt, bit 19; Job Interval Timer, bit 20; or Breakpoint, bit 31 enable bit was set. There is no response routine to handle the cause of the interrupt. Register 1 address indicates a location encountered after the condition.	Prevent the problem, or use Q7DFSET to specify a routine to handle the problem or to reset the data flag register bit 19, 20, and 31.
033	F	CLASS III INTERRUPT IN CLASS III INTERRUPT HANDLING ROUTINE - REGISTER 1 ADDRESS	Interrupt occured while a Data Flag Branch response was being processed. The response and the new interrupt are both associated with the bit range of 21 ... 31. Register 1 address indicates a location encountered after the condition.	Correct error; rerun.
034	C	DATA FLAG BRANCH, NO PRODUCT BITS ON - REGISTER 1 ADDRESS	This error message suggests a hardware or software error. This is an extraordinary condition. Register 1 address indicates a location encountered after the condition.	Follow site-defined procedure.
035	F	FORTRAN SECOND USE OF Q7DFCL1 CONFLICTS WITH USER	SECOND uses Q7DFCL1 to link itself to data flag branches resulting from bit 36.	Remove call to SECOND or remove previous Q7DFCL1 call enabling 'JIT'.
036	F	USER USE OF Q7DFCL1 CONFLICTS WITH FORTRAN SECOND	SECOND uses Q7DFCL1 to link itself to data flag branches resulting from bit 36.	Remove call to Q7DFCL1 or remove previous SECOND call.
038	C	VALUE GIVEN FOR ACCESS SPECIFIER IS NOT RECOGNIZED	The value must be either 'SEQUENTIAL' or 'DIRECT', optionally followed by blanks.	Correct error; rerun.
039	C	VALUE GIVEN FOR FORMAT SPECIFIER IS NOT RECOGNIZED	The value must be either 'FORMATTED' or 'UNFORMATTED', optionally followed by blanks.	Correct error; rerun.
040	C	BLANK SPECIFIER MAY NOT BE GIVEN FOR AN UNFORMATTED FILE	The "BLANK=" specifier does not apply when a file is opened as unformatted.	Correct error; rerun.
041	C	VALUE GIVEN FOR BLANK SPECIFIER IS NOT RECOGNIZED	Must be either 'NULL' or 'ZERO', optionally followed by blanks.	Correct error; rerun.

TABLE B-2. EXECUTION TIME ERROR MESSAGES (Contd)

Error Number	Error Level	Message	Significance	Action
042	C	VALUE GIVEN FOR OPEN STATUS IS NOT RECOGNIZED	Must be 'OLD', 'NEW', 'SCRATCH', or 'UNKNOWN', optionally followed by blanks.	Correct error; rerun.
043	C	RECORD LENGTH MUST BE SPECIFIED FOR A DIRECT ACCESS FILE	"RECL=" must be specified for the OPEN of a direct access file.	Correct error; rerun.
044	C	RECORD LENGTH MAY NOT BE SPECIFIED FOR A SEQUENTIAL ACCESS FILE	"RECL=" must not be specified on the OPEN of a sequential access file.	Correct error; rerun.
045	C	VALUE GIVEN FOR RECORD LENGTH IS OUT OF RANGE 1 ... (2**24)-1	The value associated with "RECL=" must fall within the prescribed range.	Correct error; rerun.
046	C	VALUE GIVEN FOR BUFFER LENGTH IS OUT OF RANGE 1 ... 24	The allowable values for buffer length lie between 1 and 24 512-word blocks.	Correct error; rerun.
047	C	FILE NAME MAY NOT HAVE LEADING OR EMBEDDED BLANKS	A blank character preceeds a non-blank character in the specified name.	Correct error; rerun.
048	C	FILE NAME MAY NOT BE SPECIFIED FOR A SCRATCH FILE	'SCRATCH' status files do not have names.	Correct error; rerun.
049	C	FILE NAME MUST BE SPECIFIED FOR A NEW FILE	Files without 'SCRATCH' status must be named in the OPEN statement	Correct error; rerun.
050	C	FILE NAME MUST BE SPECIFIED FOR AN OLD FILE	Files without 'SCRATCH' status must be named in the OPEN statement	Correct error; rerun.
051	F	FILE ALREADY CONNECTED AS A SEQUENTIAL ACCESS FILE	Attempted to use OPEN to change the file's "ACCESS=" specifier from 'SEQUENTIAL' to 'DIRECT'.	Correct error; rerun.
052	F	FILE ALREADY CONNECTED AS A DIRECT ACCESS FILE	Attempted to use OPEN to change the file's "ACCESS=" specifier from 'DIRECT' to 'SEQUENTIAL'.	Correct error; rerun.
053	F	FILE ALREADY CONNECTED AS A FORMATTED FILE	Attempted to use OPEN to change the file's "FORM=" specifier from 'FORMATTED' to 'UNFORMATTED'.	Correct error; rerun.
054	F	FILE ALREADY CONNECTED AS AN UNFORMATTED FILE	Attempted to use OPEN to change the file's "FORM=" specifier from 'UNFORMATTED' to 'FORMATTED'.	Correct error; rerun.
055	F	FILE ALREADY CONNECTED AS A SCRATCH FILE	Attempted to use OPEN to change the file connected to the specified unit from a 'SCRATCH' file to a named file.	Correct error; rerun.
056	F	FILE ALREADY CONNECTED AS A NAMED FILE	Attempted to use OPEN to change the file connected to the specified unit from an 'OLD' or 'NEW' file to a 'SCRATCH' file.	Correct error; rerun.

TABLE B-2. EXECUTION TIME ERROR MESSAGES (Contd)

Error Number	Error Level	Message	Significance	Action
057	F	RECORD LENGTH MAY NOT BE CHANGED FOR A CONNECTED FILE	Attempted to use OPEN to change the "RECL=" value of an already open file.	Correct error; rerun.
058	F	BUFFER LENGTH MAY NOT BE CHANGED FOR A CONNECTED FILE	Attempted to use OPEN to change the "BUFS=" value of an already open file.	Correct error; rerun.
059	F	ABSOLUTE VALUE OF CONSTANT EXCEEDS 255	Numbers in edit descriptors must not exceed 255, or be less than -255.	The following line and the period two lines down identify the number that is out of range.
060	C	A SCRATCH FILE MAY NOT BE CLOSED WITH 'KEEP'	Unnamed files cannot be kept.	Correct error; rerun.
061	C	VALUE GIVEN FOR CLOSE STATUS IS NOT RECOGNIZED	Must be 'KEEP' or 'DELETE', optionally followed by blanks.	Correct error; rerun.
062	C	UNIT DOES NOT EXIST	There is a malformed unit identifier such as a negative unit number.	Correct error; rerun.
063	F	ATTEMPT TO BACKSPACE A PRECONNECTED UNIT	BACKSPACE may be done only on units that have been used by OPEN or some data transfer I/O statement.	Correct error; rerun.
064	F	UNIT ALREADY CLOSED	Attempted to BACKSPACE or REWIND a closed unit.	Correct error; rerun.
065	F	UNIT NOT CONNECTED TO A SEQUENTIAL ACCESS FILE	Attempted to BACKSPACE or REWIND a unit opened for 'DIRECT' access.	Correct error; rerun.
066	C	UNEXPECTED CHARACTER	Execution time preconnection specifier list has an out-of-place character. The following line and the period on the second line down show the character to be changed.	Correct error; rerun.
067	C	INVALID CHARACTER	Execution time preconnection specifier list has an inappropriate character. The following line and the period on the second line down show the character to be changed.	Correct error; rerun.
068	C	PREMATURE END OF LIST ITEM	A separator was found before the end of the executation time preconnection specifier list item. The following line and the period on the second line down show the separator.	Correct error; rerun.
069	C	NULL LIST ITEM	Execution time preconnection specifier list begins or ends in a comma or it has two adjacent commas. The following line and the period on the second line down show the terminator of the null element.	Correct error; rerun.

TABLE B-2. EXECUTION TIME ERROR MESSAGES (Contd)

Error Number	Error Level	Message	Significance	Action
070	C	EXPECTED COMMA MISSING	Two execution time preconnection specifier list elements seem run together. The following line and period on the second line down show the beginning of the second element.	Correct error; rerun.
071	C	FILE OR UNIT NAME TOO LONG	File and unit names are limited to 8 characters. The following line and the period on the second line down show the ninth character.	Correct error; rerun.
072	C	MULTIPLE RLP SPECIFICATIONS	Only one RLP specification is allowed in the effective execution time preconnection specifier list. The following line and the period on the second line down show the second RLP specification.	Correct error; rerun.
073	C	REDUNDANT OR CONTRADICTORY UNIT IDENTIFIER USE	The same unit identifier may not appear in two preconnection specifiers. The following line and the period on the second line down show the second identifier use.	Correct error; rerun.
074	F	NEW FILE ALREADY EXISTS	You specified a 'NEW' status for an already existing file in the OPEN statement.	Correct error; rerun.
075	F	OLD FILE DOES NOT EXIST	You specified an 'OLD' status for a nonexistent file in the OPEN statement.	Correct error; rerun.
076	C	VALUE REPETITION COUNT IS GREATER THAN (2**47)-1	List directed input record has unreasonably large repetition (*) count.	Correct input file; rerun.
077	C	VALUE REPETITION COUNT MUST BE GREATER THAN ZERO	List directed input record has a negative or zero repetition (*) count value.	Correct input file; rerun.
078	C	INPUT IS INCOMPATIBLE WITH LIST ITEM TYPE	There is a conflict between a list item and the corresponding input record character(s) in the input from a 'FORMATTED' file.	Correct error; rerun.
079	C	RECORD SIZE TOO SMALL	System record size for your file is too short for list directed output to put all of the characters of a value on the same line.	Use a file with a larger minimum record length.
080	F	NUMBER OF VALUES REQUIRED BY INPUT LIST GREATER THAN NUMBER OF VALUES IN RECORD	The unformatted READ needs enough information in the record to define all input list items.	Correct error; rerun.

TABLE B-2. EXECUTION TIME ERROR MESSAGES (Contd)

Error Number	Error Level	Message	Significance	Action
081	C	FILE NAMES MUST CONTAIN AT LEAST ONE LETTER	Cannot have an all numeric file name. The following line and the period on the second line down identify the file name.	Correct error; rerun.
082	C	RLP VALUE MUST NOT EXCEED (2**26)-1	RLP value is too large. Entire machine address space is 2**26 large pages. The following line and the period on the second line down identify the incorrect RLP specification.	Correct error; rerun.
083	C	INVALID DATA TYPE CODE	List directed output encountered a garbled output list item. This is an extraordinary error for a FORTRAN routine.	Follow site-defined procedure.
084	F	CONSTANT MUST NOT BE 0 FOR THIS EDIT DESCRIPTOR	A numeric part of an edit descriptor was zero when it should not have been. The following line and the period on the second line down identify the edit descriptor.	Correct error; rerun.
085	F	PORTION OF TRANSLATED FORMAT STRING BEFORE LAST "(" IS TOO LONG	The run-time format length exceeds the format translator capacity. The following line and the period on the second line down shows where the translator's capacity was exhausted.	Correct error, rerun.
086	F	APOSTROPHE STRING IS LONGER THAN 255 CHARACTERS	Maximum length for an apostrophe string edit descriptor is 255 characters. The following line and the period on the second line down identify the string.	Correct error, rerun.
087	F	NULL APOSTROPHE STRING	Zero length character strings are disallowed.	Correct error; rerun.
088	F	UNEXPECTED END OF FORMAT	Reached end of format without finding all balancing right parentheses. Following line shows the format in question.	Correct error; rerun.
089	F	NON-BLANKS NOT PERMITTED BEFORE INITIAL LEFT PARENTHESIS	The first non-blank character in the format was not "(". The following line and the period on the second line down identify the first non-blank character.	Correct error; rerun.
090	F	INVALID EDIT DESCRIPTOR	Some succession of characters fails to form a proper edit descriptor. The following line and the period on the second line down indicate the region of format being examined.	Correct error; rerun.

TABLE B-2. EXECUTION TIME ERROR MESSAGES (Contd)

Error Number	Error Level	Message	Significance	Action
091	F	COMMA REQUIRED BEFORE THIS CHARACTER	The indicated edit descriptor and the one before must have a separating comma. The following line and the period on the second line down indicate the region of format being examined.	Correct error; rerun.
092	F	A DIGIT WAS EXPECTED AFTER THE SIGN	A digit must follow a plus or minus sign that appears in an edit descriptor. The following line and the period on the second line down indicate the region of format being examined.	Correct error; rerun.
093	F	THIS IS NOT A REPEATABLE EDIT DESCRIPTOR	A repetition count was placed before a non-repeatable edit descriptor. The following line and the period on the second line down indicate the region of format being examined.	Correct error; rerun.
094	F	EDIT DESCRIPTOR ENDED PREMATURELY	Separator encountered before edit descriptor was fully formed. The following line and the period on the second line down indicate the region of format being examined.	Correct error; rerun.
095	F	THIS CHARACTER IS PERMITTED ONLY IN A HOLLERITH STRING	The indicated character cannot be used to form any kind of edit descriptor except Hollerith, or apostrophe. Here, it is used in another context. The following line and the period on the second line down indicate the region of format being examined.	Correct error; rerun.
096	F	SIGNED CONSTANTS ARE VALID ONLY FOR SCALE FACTORS	Except for Hollerith and apostrophe edit descriptors, P is the only edit descriptor that can contain a plus or minus. The following line and the period on the second line down indicate the region of format being examined.	Correct error; rerun.
097	F	TO INCLUDE AN APOSTROPHE IN AN APOSTROPHE STRING USE TWO CONSECUTIVE APOSTROPHES	The ending of the apostrophe string edit descriptor suggests that the string was meant to include an apostrophe instead. The following line and the period on the second line down indicate the region of format being examined.	Correct error; rerun.

TABLE B-2. EXECUTION TIME ERROR MESSAGES (Contd)

Error Number	Error Level	Message	Significance	Action
098	C	INSUFFICIENT SPACE RESERVED FOR TRANSLATED FORMAT - LIBRARY FAILURE	This is an extraordinary condition. The translator should not run out of space. The following line and the period on the second line down indicate the area of the format being processed when the translator ran out of space.	Correct error; rerun.
099	C	INVALID EDIT DESCRIPTOR IN TRANSLATED FORMAT	The translated format contains an error.	Look for something that could have overwritten the format.
100	F	A SEPARATOR IS REQUIRED AFTER NAMELIST GROUP NAME	A NAMELIST group and the first variable name in an input record ran together.	Correct input file; rerun.
101	F	MAXIMUM RECORD SIZE EXCEEDED	Attempted to write a record that is too large for the file.	Correct error; rerun.
102	F	FORMAT EDIT DESCRIPTOR INCOMPATIBLE WITH LIST ITEM TYPE	Trying to process a list item type that does not fit the type described by the edit descriptor.	Correct error; rerun.
103	F	INVALID SEPARATOR ENCOUNTERED	List directed or NAMELIST input encountered a character other than blank, comma, or slash after a character constant.	Correct input file; rerun.
104	F	INVALID CHARACTER STRING DELIMITER	List directed or NAMELIST input encountered a null character string - two adjacent apostrophes followed by a non-apostrophe.	Correct input file; rerun.
105	C	ERROR IN TRANSLATED RUN TIME FORMAT	You have made a direct call to the format translator to translate a run-time format that has an error and ignored the error code.	Do not attempt to use the translated form of a format that has an error.
106	F	MAGNITUDE OF VALUE GREATER THAN (2**47)-1	An input value exceeds the machine's capacity to represent integers.	Correct error; rerun.
107	C	INVALID OUTPUT LIST ITEM	An I/O library call passed the library invalid information. This would be an extraordinary error for a FORTRAN program.	Follow site-defined procedure.
108	C	RECORD NUMBER MUST BE SPECIFIED FOR DIRECT ACCESS I/O	The compiler generated a direct access I/O call, but indicated sequential access by not specifying REC=.	Follow site-defined procedure.
109	C	NAMELIST ITEM DOES NOT BELONG TO GROUP IN WHICH IT APPEARS	NAMELIST input record specifies a variable name that is not declared in the NAMELIST group being read.	Correct error; rerun.
110	F	UNEXPECTED END OF LINE IN NAMELIST INPUT	End of line encountered in area that is disallowed by the format for the NAMELIST data.	Correct input file; rerun.

TABLE B-2. EXECUTION TIME ERROR MESSAGES (Contd)

Error Number	Error Level	Message	Significance	Action
111	C	UNRECOGNIZABLE SUBSCRIPT DIGIT IN NAMELIST INPUT	A subscript value has a non-numeric character in a NAMELIST input record.	Correct input file; rerun.
112	C	NAMELIST ITEM SUBSCRIPT IS OUT OF RANGE	A subscript value is too small or too large in a NAMELIST input record	Correct input file; rerun.
113	F	NAMELIST ITEM SUBSCRIPT LACKS EXPECTED ","	The dimensionality implied by a NAMELIST input record is smaller than that of the array.	Correct error; rerun.
114	F	NAMELIST ITEM SUBSCRIPT EXPRESSION LACKS EXPECTED ")"	The dimensionality implied by a NAMELIST input record is greater than that of the array.	Correct error; rerun.
115	F	NAMELIST ITEM ASSIGNMENT LACKS EXPECTED "="	The NAMELIST input record item name and its value run together with no "=" between.	Correct input file; rerun.
116	F	REPETITION COUNT AND VALUE MUST BE ON THE SAME LINE	An asterisk may not be the last character of a NAMELIST input line.	Correct input file; rerun.
117	C	FILE IS NOT POSITIONED AT BEGINNING OF A NAMELIST BLOCK	Attempted a NAMELIST input when the first record had no NAMELIST group name.	Correct error; rerun.
118	C	A NON Q7BUF IN/OUT OPERATION TO THIS UNIT IS NOT ACCEPTABLE	Attempted to mix Q7 and non-Q7 I/O on one unit.	Correct error; rerun.
119	C	A NON BUFFER IN/OUT OPERATION TO THIS UNIT IS NOT ACCEPTABLE	Attempted to mix BUFFER and non-BUFFER I/O on one unit.	Correct error; rerun.
120	F	ARRAY MUST BE ON A BLOCK BOUNDARY	User's ARRAY was not on a block boundary.	Use a COMMON declaration and load with parameter GRSP or GRLP.
121	F	MORE THAN 2 OUTSTANDING REQUESTS FOR SAME FILE	There are more than 2 outstanding I/O operations to a file.	Call Q7WAIT.
122	W	BLOCK LENGTH MUST BE A POSITIVE NON-ZERO VALUE	Block length for Q7SEEK must not be negative.	Correct error; rerun.
123	F	MAP PARAMETER CONTAINS AN INVALID VALUE	Map parameter contains something other than 'SMALL' or 'LARGE'.	Correct error; rerun.
124	F	THE PREVIOUS BUFFER I/O OPERATION ENDED ABNORMALLY AND THE UNIT FUNCTION HAS NOT BEEN CALLED.	(self explanatory)	Call UNIT function.
125	F	SCALE FACTOR IS OUT OF RANGE	The scale factor (k) on the P edit descriptor cannot be applied to a D, E, or G edit descriptor because it would remove all significance from the field.	Change the scale factor (k) or the field width (d) to fit within the range $-d < k < d+2$ where $k < 256$.
128	F	INCOMPATIBLE EDIT DESCRIPTOR FIELDS	The field width specified is too small to contain the other widths specified in the edit descriptor.	Increase the field width or decrease the other widths in the edit descriptor.

TABLE B-3. VECTORIZER MESSAGES

Message	Significance	Action
THIS LOOP IS BRANCHED INTO	A DO loop must be entered from the top.	Extended DO loop ranges are not vectorized. If the branches are to an extended range, moving the extended range into the DO loop body may permit vectorization.
THIS LOOP IS BRANCHED OUT OF	A DO loop must be exited at the bottom.	Extended DO loop ranges are not vectorized. If the branches are to an extended range, moving the extended range into the DO loop body may permit vectorization.
THE DO NEST CONTAINS EIGHT OR MORE LOOPS	Only the innermost seven loops of a nest can be vectorized.	Because an array can have at most seven subscripts, no attempt is made to vectorize more than seven loops. If possible, only the seven innermost loop control variables should be used in subscripts.
THIS OUTER LOOP HAS MORE THAN 65535 ITERATIONS	The length of a vector in a DO nest cannot exceed 65535 (or 32767 if the loop contains complex arrays).	The innermost loop can be vectorized regardless of its iteration count. The DO nest might be manually collapsed into a single DO statement.
A DESTINATION VARIABLE MAY BE A RECURSIVE DEFINITION	A feedback condition either does or may exist.	If array bounds are adjustable or assumed and the loop parameters are not constants, or if the subscripts are complicated, the vectorizer may be unable to determine if feedback does not occur and unable to vectorize. If the loop is feedback free, use of constant array bounds or simpler subscripts might permit vectorization. If, on the other hand, the loop is recursive, it cannot be properly executed on the vector hardware. Unsafe vectorization permits possible feedback among equivalenced arrays if the different array names are used.
A DESTINATION VARIABLE IS NOT REAL, INTEGER, LOGICAL, HALF PRECISION, OR COMPLEX	Double precision, character, and bit data elements cannot be vectorized.	Unless the data can be recoded in an acceptable data type, it cannot be vectorized.
THE DO VARIABLE IS NOT AN INTEGER	Only integer loop control variables are vectorized.	If the DO statement is originally of the form DO label cv=ll,ul,incr, the statement might be replaced with DO label icv=1,(ul-ll+incr)/incr, where icv is new integer variable, and the statement cv=ll+(icv-1)*incr can be inserted at the head of the loop.
	An outer loop has a nonunit incrementation parameter.	The loops might be rearranged to avoid a nonunit increment.
THIS IS AN OUTER LOOP WITH VARIABLE INITIAL OR TERMINAL VALUES WITHOUT UNSAFE OPTIMIZATION	Loop initial or terminal parameter is not constant and the control variable only subscripts adjustable or assumed dimensions.	The vectorizer is unable to determine if an outer loop has an iteration count above or below 65535. The innermost loop can be vectorized regardless of its iteration count. The DO nest might be manually collapsed into a single DO statement. If constant array bounds or constant loop parameters are used, the loop might vectorize. If unsafe optimization is selected, the loop will vectorize and the iteration count will be assumed to be less than or equal to 65535.
THIS LOOPS CONTAINS A NONVECTORIZABLE LOOP	Any nonvectorizable inner loop prohibits vectorization of all outer loops.	If the inner loop can be made vectorizable, the outer loop might also vectorize.
A PROPERTY OF AN EMBEDDED LOOP PREVENTS THIS LOOP FROM VECTORIZING	An inner loop prevents vectorization of all outer loops.	This covers a variety of problems usually caused by complicated subscripts, adjustable or assumed array bounds, and complicated loop parameters. Possibly a control variable is used in different subscript positions.

TABLE B-3. VECTORIZER MESSAGES (Contd)

Message	Significance	Action
A SOURCE VARIABLE IS NOT REAL, INTEGER, LOGICAL, HALF PRECISION, OR COMPLEX	Double-precision, character, and bit data elements will not be vectorized.	Unless the data can be recoded in an acceptable data type, it cannot be vectorized.
A SOURCE VARIABLE MAY BE A RECURSIVE DEFINITION	A feedback condition either does or may exist.	If array bounds are adjustable or assumed and the loop parameters are not constants, or if the subscripts are complicated, the vectorizer may be unable to determine if feedback does not occur and unable to vectorize. If the loop is feedback free, use of constant array bounds or simpler subscripts might permit vectorization. If, on the other hand, the loop is recursive, it cannot be properly executed on the vector hardware. Unsafe vectorization permits possible feedback among equivalenced arrays if the different array names are used.
A SCALAR USED IN AN OUTER LOOP IS ALSO USED IN AN INNER LOOP	A scalar can be defined only in the innermost loop in which it appears.	If the scalar is used for different purposes in the inner and outer loop, new scalars can be introduced for each purpose.
A SCALAR IS REFERENCED BEFORE OR IN THE SAME STATEMENT AS ITS FIRST DEFINITION	A scalar's value cannot depend upon its value in a previous iteration in the general case.	If the scalar values form an arithmetic progression, an interval recursive assignment can be used. An interval assignment s=s+r can be moved to the top of the loop. All references to s between the top of the loop and the original position of the assignment are changed to references of s-r. If the scalar is used to compute a sum, product, or dot product, partial results are not available within the loop.
A NONVECTORIZABLE FUNCTION IS CALLED	The loop references an external subprogram or nonvectorizable intrinsic function.	Only vectorizable intrinsic functions can be used in the loop. If the subprogram values are not dependent on any definitions in the loop, the subprogram values can be precomputed and stored in an auxiliary array for use in the loop. The vectorizable intrinsics are ABS, ACOS, ALOG, ALOG10, ASIN, ATAN, COS, EXP, FLOAT, IABS, IFIX, SIN, SQRT, and TAN.
AN OPERATOR CANNOT BE VECTORIZED	Only arithmetic (+, -, *, /, and **) logical operators are vectorizable.	The DO loop might be rewritten as two or more loops so that expressions with nonvectorizable operators are in separate unvectorized loops with their results stored in an auxiliary array for use in the loop.
THIS LOOP CONTAINS A VECTOR ASSIGNMENT	The loop cannot contain vector assignment statements.	Loops containing explicit vectors are assumed to be manually optimized.
THIS LOOP CONTAINS A NONVECTORIZABLE KIND OF STATEMENT	Only scalar assignment statements and embedded DO loops can be vectorized.	All other kinds of statements should be moved out of the loop if possible.
THE CONTROL VARIABLE OF THIS LOOP IS NEVER USED IN A SUBSCRIPT OR AN EXPRESSION	All statements in the loop are explicitly and implicitly independent of the value of the loop control variable.	The corresponding DO statement serves no discernible purpose and should probably be deleted.

TABLE B-3. VECTORIZER MESSAGES (Contd)

Message	Significance	Action
THIS OUTER LOOP HAS A VARIABLE INCREMENT	The value of an incrementation parameter of an outer or embedded loop cannot be computed during compilation and might be other than 1.	If the increment is constant, that constant should be used. The PARAMETER statement can be used to define symbolic constants. The innermost loop can always be vectorized regardless of its increment. The DO nest might be manually collapsed into a single DO statement or the loops might be rearranged to avoid a variable increment.
AN INNER LOOP MIGHT NOT COMPLETELY SPAN THE CORRESPONDING DIMENSION OF A DESTINATION ARRAY	When a control variable appears in a subscript for a dimension of an array, the dimension bounds must equal the initial and terminal parameter of the DO statement of the control variable.	The innermost loop can always be vectorized regardless of spanning. The DO nest might be manually collapsed into a single DO statement. Complicated subscripts, assumed or adjustable arrays, or complicated loop parameters can make it difficult to verify spanning.
AN INNER LOOP MIGHT NOT COMPLETELY SPAN THE CORRESPONDING DIMENSION OF A SOURCE ARRAY	When a control variable appears in a subscript for a dimension of an array, the dimension bounds must equal the initial and terminal parameter of the DO statement of the control variable.	The innermost loop can always be vectorized regardless of spanning. The DO nest might be manually collapsed into a single DO statement. Complicated subscripts, assumed or adjustable arrays, or complicated loop parameters can make it difficult to verify spanning.

TABLE B-4. CONTROL STATEMENT ERROR MESSAGES

Error Type	Message†	Significance
**COMPILER FAILURE--	IMPROPER CALL TO Q7PROMPT.	Compilation terminates.
**ERROR--	INPUT FILE RECORD TYPE UNDEFINED.	The input file has no record structure. Compilation terminates.
**ERROR--	RIGHT HAND SIDE OF 'LO=c' is INVALID.	The only valid options are A, M, S, I, X, or their combinations. Compilation terminates.
**ERROR--	'S' OPTION MUST BE SPECIFIED WITH 'I'.	If the S option is not specified with the I option, compilation terminates.
**ERROR--	RIGHT HAND SIDE OF 'OPT=c' IS INVALID.	The only valid options are D, P, R, S, V, or their combinations. Compilation terminates.
**ERROR--	SIL DETECTED ERROR FOR BINARY FILE.	Message followed by an SIL error message. Compilation terminates.
**ERROR--	SIL DETECTED ERROR FOR ERROR FILE.	Message followed by an SIL error message. Compilation terminates.
**ERROR--	SIL DETECTED ERROR FOR INPUT FILE.	Message followed by an SIL error message. Compilation terminates.
**ERROR--	SIL DETECTED ERROR FOR LIST FILE.	Message followed by an SIL error message. Compilation terminates.
**WARNING--	WHEN 'F66' IS SPECIFIED 'ANSI' IS NOT APPLICABLE. 'ANSI=0' IS ASSUMED.	While compiling the 1966 FORTRAN dialect, it makes no sense to diagnose deviations to the 1978 FORTRAN dialect. Compilation proceeds.
**WARNING--	WHEN 'F66' IS SPECIFIED 'DO=0' IS NOT APPLICATION. 'DO-1' IS ASSUMED.	The 1966 FORTRAN dialect does not support zero iterations for DO loops. Compilation proceeds.
**WARMING--	WHEN 'SYNTAX' IS SPECIFIED 'BINARY' IS NOT APPLICABLE. 'BINARY=0' IS ASSUMED.	When the compiler does quick syntax checking, it produces no object file. Compilation proceeds.
**WARNING--	WHEN 'SYNTAX' IS SPECIFIED 'LO=c' IS NOT APPLICABLE. IT IS IGNORED.	The SYNTAX control statement conflicts with LO=S, LO=X, and LO=SX. Compilation proceeds.
**WARNING--	WHEN 'SYNTAX' IS SPECIFIED 'OPTIMIZE' IS NOT APPLICABLE. 'OPTIMIZE=0' IS ASSUMED.	OPTIMIZE conflicts with SYNTAX. Compilation proceeds.
**WARMING--	WHEN 'SYNTAX' IS SPECIFIED 'UNSAFE' IS NOT APPLICABLE. 'UNSAFE=0' IS ASSUMED.	UNSAFE conflicts with SYNTAX. Compilation proceeds.

†A lower case c in a message signifies the position of a compiler-supplied, appropriate single character embedded in the message.

GLOSSARY C

Terms used in the main text of this manual are described in this section. The definitions give the general meanings of the terms. Precise definitions are given in the main text. Also, most general terms regarding computers and terms defined in the American National Standards documents regarding the FORTRAN language have been excluded.

Array -
An ordered set of variables identified by a single symbolic name. Referencing a single element of an array requires the array name plus a subscript that specifies the element's position in the array.

Array Declarator -
Specifies the dimensions of an array.

ASCII Data -
Characters, each of which has a standard internal representation. One byte (8 bits) is required for each character.

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.

Compilation Time -
The period of time during which the FORTRAN compiler is reading the program and producing the relocatable module for the program. Compilation is initiated by the FTN200 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 execution-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, half-precision, 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.

Execution Time -
The period of time during which the compiled program is executing. Execution is initiated by a system control statement.

Explicit Typing -
Specification of the data type of a variable or array by means of one of the explicit type statements (the INTEGER, HALF PRECISION, 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 object program that causes control to transfer to the external function during program execution. External functions contrast with inline functions.

File -
A collection of information that can be defined by output statements, or referenced by input statements. A file can reside on a disk or tape.

First-Letter Rule -
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 half-precision, 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.

Inline Function -
A type of predefined function. Referencing an inline function causes the function's object code to be inserted directly into the relocatable object code of the program during compilation. Inline 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 execution-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 inline functions, external functions, or both inline and external functions.

Program -
A procedure described in the FORTRAN programming language, consisting of at least a main program along with any functions and subroutines written by you that are referenced directly or indirectly by that main program.

Punch -
The name of the file to which records written by the PUNCH statement are written.

Record -
The amount of information read or written by a single FORTRAN READ or WRITE statement. In formatted input/output, a new record is started each time a slash edit descriptor or a format repetition is processed.

Referencing -
Identifying data for the purpose of making its current value available during the execution of the statement containing the reference. Also, identifying a procedure for the purpose of making the actions specified available for execution. Referencing contrasts with naming and defining.

Rowwise -
The ordering of the elements in an array declared in a ROWWISE statement (the other ordering is columnwise). The succession of subscripts corresponding to the elements of a rowwise array is with the value of the rightmost subscript expression varying the fastest.

Scalar -
A single value; contrasted to vectors, which are typically groups of values.

Semicolon Notation -
A notation used to express a vector. The semicolon separates the two items specifying the vector, namely, its first element and its length.

Side Effect -
The alteration of an argument or an element in a common area as a result of a function reference.

Small Page -
A block of 512 words in memory starting on a small page boundary. A small page is the smallest unit that can be moved in or out of main memory by the operating system.

Special Call -
A feature that can be used to cause specific machine instructions to be generated in the object code at compilation time.

STACKLIB Routine -
A routine that optimizes certain loops that cannot be vectorized.

Subarray -
A cross section of an array. Identified either by the array name or by the array name qualified by a subscript containing (among other kinds of subscript expressions) one or more subscript expressions in colon notation.

Subscripted Array Name -
An array name followed by a parenthesized list of integer constants or simple integer expressions that specify a particular element in an array. A subscripted array name is either an array element reference or an array element definition.

Symbolic Constant -
A name that has a constant value. The value is specified by the PARAMETER statement.

System Interface Language -
A set of subroutines that user programs can call to perform system functions.

Unit -
A disk or tape on which a file can be created and kept by the operating system.

Vector -
A data representation that typically consists of more than one value; contrasted to scalar data, which represent single values. A subset of an array of scalar elements or of the dynamic space area, delimited by a length and a subscript which designates the position in the array of the vector's first element.

Vectorize -
Cause vector machine instructions to be generated as part of the object code either by using vector data and referencing vector functions, or by including vectorizable DO loops in a program compiled when the OPTIMIZE=V compilation option has been selected.

Virtual Memory -
A conceptual extension of main memory achieved by a hardware/software technique which permits memory references beyond the physical limitation of main memory. Virtual memory addresses are associated with real addresses in physical memory during program execution.

FORTRAN 200 STATEMENT SUMMARY D

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. Brackets around an item indicate that the item is optional. Abbreviations used in this appendix are the following:

adecl	=	array declarator
aarg	=	actual argument
aexp	=	scalar arithmetic expression of any type except complex
alt	=	integer constant or simple integer variable
arexp	=	scalar arithmetic expression
arithvar	=	variable or array element of type integer, half-precision, real, double-precision, or complex
arrayexp	=	array expression or any scalar expression
bexp	=	vector bit expression
blk	=	common block name
bvec	=	bit vector
cblist	=	list of variables, arrays, or array declarators separated by commas
charvar	=	character variable, array element, or substring reference
cilist	=	control information list
cexp	=	scalar character expression
contvar	=	control variable, which can be a variable of any type except complex
d	=	a constant that specifies an initial value
darg	=	dummy argument
desc	=	descriptor or descriptor array element
disp	=	one to five decimal digits or a character constant
dlist	=	list of initial values, which can be constants or constants preceded by repeat specifications
exp	=	expression
fid	=	format identifier
first	=	first location in buffer
fspec	=	format specification
group	=	list of two or more variables, array elements, arrays, or substrings, separated by commas
grpname	=	namelist group name
int	=	intrinsic function name
iolist	=	input/output list
isexp	=	integer scalar expression
ivar	=	integer variable

k	=	integer constant, variable, expression enclosed in parentheses, or asterisk enclosed in parentheses that specifies the length in characters of v.
K	=	constant, symbolic constant enclosed in parentheses, expression enclosed in parentheses, or asterisk enclosed in parentheses that specifies the length in characters of each v.
last	=	last location in buffer
lexp	=	scalar logical expression
letlist	=	list of letters and ranges of letters
logvar	=	logical variable or array element
map	=	dynamic space mapping parameter
mode	=	data transfer mode
name	=	symbolic constant name
niolist	=	namelist input/output list
no-chars	=	record length in characters
proc	=	procedure name
ps	=	preconnection specifier
savename	=	variable, array, or common block name
sl	=	statement label
statfunc	=	statement function name
stmt	=	executable statement other than DO, logical IF, block IF, ELSE IF, ELSE, END IF, block WHERE, OTHERWISE, END WHERE, or END
suba	=	subarray reference
type	=	INTEGER, HALF PRECISION, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, CHARACTER, or BIT
uid	=	unit identifier
v	=	symbolic constant, variable, array, array declarator, or function name
vaexp	=	vector or scalar arithmetic expression
value	=	constant, constant expression, or extended constant expression
vec	=	vector reference, descriptor, or descriptor array element
vlist	=	list of variables, arrays, array elements, substrings, or implied DO loops, separated by commas

SCALAR ASSIGNMENT STATEMENTS

FLOW CONTROL STATEMENTS

SPECIFICATION STATEMENTS

PROCEDURE DEFINITION

INPUT/OUTPUT STATEMENTS

ARRAY ASSIGNMENT

VECTOR STATEMENTS

COMPATIBILITY FEATURES

Certain features of FORTRAN 200 are provided only for compatibility with FORTRAN Extended. The compatibility features are described in this appendix.

NOTE

The features described in this appendix should not be used for new programs and are intended only for the conversion of existing programs.

HOLLERITH CONSTANT COMPATIBILITY

Hollerith elements are described in section 2, Statement Elements. For compatibility, Hollerith constants are supported in relational and arithmetic expressions.

A Hollerith constant used in an arithmetic or relational expression is limited to eight characters. An H constant is left-justified with blank fill in a fullword. An H constant that is too long is truncated on the right, and a warning diagnostic is issued. An R constant is right-justified with binary zero fill in a fullword. An R constant that is too long is truncated on the right and a warning diagnostic is issued.

The Hollerith constant is considered typeless. A typeless constant is not converted for use as an argument or for assignment. If Hollerith constants are the only operands in an arithmetic expression, the result is type integer.

BUFFER IN AND BUFFER OUT COMPATIBILITY

Input, output, and memory transfer statements are described in section 6. The BUFFER IN and BUFFER OUT statements are provided for compatibility with FORTRAN Extended. The UNIT and LENGTH functions are also provided for compatibility.

The BUFFER IN and BUFFER OUT statements are used to transmit binary data between binary files and main memory. The length of the buffer area in which the data is contained should be an even number of bytes for tape files, or a multiple of pages for disk files. Ordering the data in this manner provides for the most economical use of storage.

A file referenced in a BUFFER IN or BUFFER OUT statement must be preconnected or connected for sequential access. The specified unit must not be referenced in any other data transfer input/output statement while connected. However, the unit can be closed and opened again. The unit can be referenced in the file positioning statements BACKSPACE, ENDFILE, and REWIND. The unit or the file can also be referenced in an INQUIRE statement.

After a BUFFER IN or BUFFER OUT, the error status of the logical unit involved should be checked by using the UNIT function before another operation with the unit is initiated. The unit status should also be checked before the buffered data is used. After the unit check, the number of bytes read by a BUFFER IN can be optained with the LENGTH function.

BUFFER IN STATEMENT

The format of the BUFFER IN statement is shown in figure E-1.

```
BUFFER IN (u,mode) (first,last)

u       External unit identifier.

mode    Disregarded.

first   A variable or array element name that
        can be type character, integer, half-
        precision, real, double-precision,
        complex, or logical, and which defines
        the first location of the buffer into
        which data is to be transmitted.

last    A variable or array element name that
        can be type character, integer, half-
        precision, complex, or logical, and
        which defines the location in the
        buffer into which the last data item
        is to be transmitted.
```

Figure E-1. BUFFER IN Statement

Execution of the BUFFER IN statement causes transfer of data from the specified external unit to the buffer defined by first and last. Only one record is read for each BUFFER IN statement.

The location of last cannot precede first in memory. The value (last-first+1) must be less than or equal to 12288 words (twenty-four 512-word blocks).

BUFFER OUT STATEMENT

The format of the BUFFER OUT statement is shown in figure E-2.

Execution of a BUFFER OUT statement transfers data to the specified external unit from the buffer defined by first and last. One logical record is written for each BUFFER OUT statement. The parameters first and last must refer to the same array, and last cannot precede first in memory.

```
BUFFER OUT (u,mode) (first,last)

u        External unit identifier

mode     Disregarded.

first    A variable or array element name that
         can be type character, half-precision,
         real, integer, double-precision, com-
         plex, or logical, and which defines
         the first location of the buffer from
         which data is to be transmitted.

last     A variable or array element name that
         can be type character, half-precision,
         real, integer, double-precision, com-
         plex, or logical, and which defines the
         location in the buffer from which the
         last data item is to be transmitted.
```

Figure E-2. BUFFER OUT Statement

UNIT FUNCTION

The format of the UNIT function is shown in figure E-3.

```
UNIT (u)

u    External unit identifier.
```

Figure E-3. UNIT Function

The UNIT function checks the status of a data transmission operation. The function returns one of the following values:

-1.0	Unit ready
0.0	Unit ready; end-of-file encountered
1.0	Unit ready; parity error encountered

After a BUFFER IN or BUFFER OUT, the UNIT function should be called before any further operations are performed on the file.

The UNIT function can be referenced in an arithmetic IF statement that branches to appropriate statements, as directed by the value returned.

Note that the meaning of the sign of the value returned by the UNIT function is different from that of the values returned by the input/output status specifiers described in section 6.

LENGTH FUNCTION

The format of the LENGTH function is shown in figure E-4.

```
LENGTH (u)

u    External unit identifier.
```

Figure E-4. LENGTH Function

This function returns an integer value that represents the number of bytes actually read. If the buffer area is larger than the physical record, the excess buffer space is undefined. If the physical record is larger than the buffer, the remainder of the record is lost.

SPECIFICATION COMPATIBILITY

Input/output lists and data formatting are described in section 6. For compatibility with FORTRAN Extended, the * specification is supported, the * specification is identical to the ' specification, except that asterisks replace the apostrophes.

INTRINSIC FUNCTION COMPATIBILITY

Intrinsic functions are described in section 10. For compatibility, a number of additional functions are supplied. The functions are shown in table E-1.

TABLE E-1. FUNCTIONS SUPPLIED FOR COMPATIBILITY

Function Reference	Argument Type	Result Type
MASK(n)	Logical Integer Real	Typeless
SHIFT(a,n)	Logical Integer Real	Typeless
COMPL(a)	Logical Integer Real	Typeless
AND(a1,a2,...)	Logical Integer Real	Typeless
OR(a1,a2,...)	Logical Integer Real	Typeless
XOR(a1,a2,...)	Logical Integer Real	Typeless

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

```
X = Y + SHIFT(I,5)
```

does not involve conversion of the SHIFT result from integer to real. The result is typeless and is used without conversion.

AND

AND(a1,a2,...) computes the bit-by-bit logical product of a1 through an.

COMPL

COMPL(a) computes the bit-by-bit Boolean complement of a.

MASK

MASK(n) forms a mask of n bits set to 1 starting at the left of the word. The value of n must be in the range 0 through 64. The result is undefined for an argument outside this range.

OR

OR(a1,a2,...) computes the bit-by-bit logical OR of a1 through an.

SHIFT

SHIFT(a,n) produces a shift of n bit positions in a. If n is positive, the shift is left circular. If n is negative, the shift is right end-off with sign extension from bit zero. The n value must in in the range -64 through 64. The result is undefined if n is outside this range. The argument n must be of type integer.

XOR

XOR(a1,a2,...) computes the bit-by-bit exclusive OR of a1 through an.

DIFFERENCES BETWEEN VSOS RELEASE 2.1.6 AND 2.2 F

This appendix has been included for those CDC customers who are not running the current PSR level 644 release VSOS 2.2 and subsequent releases.

To provide you with a better understanding of this appendix, a feature summary for release VSOS 2.2 has been supplied. In addition, this appendix will describe the FTN200 features as they appeared at release VSOS 2.1.6.

VSOS 2.2 FEATURES

The FTN200 features that changed as a result of VSOS 2.2 are as follows:

1. The basic procedure for compiling and executing a FTN200 program.

   At VSOS release 2.2 an alternative method was established for compiling and executing a FTN200 program. In order to execute a program on the 2.1.6 system the user needed the following control statements: FTN200, LOAD, and GO. Now, on VSOS release 2.2 the user can compile and execute a program by simply specifying GO (or GO=1) on the FTN200 control statements. For example, FTN200, GO=1. When using the GO parameter, the system shared library feature must be active at your installation.

2. The parameters on the FTN200 control statement.

   The following parameter has been added:

   GO parameter (See section 14 for more information)

   The following parameters have changed:

   BINARY - If the GO parameter is specified, then BINARY must be omitted or set to zero.

   You do not need to specify file length because the system allocates as many blocks as necessary for the file to hold the binary object code. If you do use the length option, the first space allocation for the file is the length you specify.

   ERRORS - You do not need to specify the file length because the system allocates as many blocks as necessary for the file to hold the error diagnostics. If you do not use the length option, the first initial allocation for the file is 16 blocks.

   LIST - You do not need to specify the file length because the system allocates as many blocks as necessary for the file to hold the listing. If you do not use the length option, the first space allocation for the file is 336 blocks.

   SYNTAX - If the GO parameter is specified, the SYNTAX parameter must be omitted or set to zero.

3. Changes to the LOAD statement.

   The following parameter has been added:

   LINK parameter (See the VSOS manual for more information)

   Note the following change in using the LIB parameter:

   LIB=F200LIB is not required on VSOS release 2.2 because the contents of F200LIB have now been put on SYSLIB. Since the default for the LIB parameter is LIB=SYSLIB, the LIB parameter can now be omitted.

4. FTN200 source code changes.

   Changed feature:

   Q8NORED subroutine (See section 11 for more information)

VSOS 2.1.6 FEATURES

The FTN200 features that changed as a result of release VSOS 2.2 will appear in this section, but will be documented as they appeared at release VSOS 2.1.6.

PROGRAM COMPILATION, LOADING, AND EXECUTION

CYBER 200 program compilation, loading, and execution are controlled by the CYBER 200 virtual state operating system (VSOS).

The three steps, compilation, loading, and execution, are performed when the following three CYBER 200 control statements are executed in a CYBER 200 job or interactive session:

1. An FTN200 statement (described in section 14).

2. A LOAD statement (described in the VSOS Reference Manual, volume 1).

3. A control statement naming the file on which the LOAD statement wrote the executable program. (The default file name is GO.)

NOTE

To load a FORTRAN 200 program, the LOAD statement must use the LIB parameter to specify the site's FORTRAN 200 library. For example: LOAD,LIB=F200LIB.

The input file read by the FORTRAN 200 compiler can contain more than one subprogram. Although subprograms can be compiled without a main program, a program cannot be loaded or executed without a main program.

Program compilation, loading, and execution are possible only within a CYBER 200 job or interactive session. For a full description of CYBER 200 jobs and interactive sessions, refer to the VSOS Reference Manual, volume 1.

CYBER 200 JOB SUBMITTAL

To submit a CYBER 200 job, you login to a front-end system, create a CYBER 200 job file, and then submit the CYBER 200 job file to the CYBER 200 system for execution. The actual statement used to submit the job differs depending on the front-end operating system. Ask site personnel for the appropriate statement for your site.

Figure F-1 shows a NOS 2 interactive session in which a batch job is submitted to a CYBER 200 system. The following paragraphs describe the session shown in the figure.

After logging in to the NOS 2 system and specifying NORMAL and BATCH modes, the user gets and displays the contents of the CYBER 200 job file.

The first statement in the job file is the job statement (ADEY,STABC.). The parameter STABC specifies the CYBER 200 mainframe identifier ABC. (You must ask site personnel for the mainframe identifier effective at your site.)

```
(date)    (time)
(NOS 2 logon banner)
FAMILY:  ,user1,passwd <------------------------ Entry to logon to NOS 2.
JSN: ABGU,NAMIAF <------------------------------ IAF logon is successful.

 READY.
normal <---------------------------------------- Ensures the terminal is in
 READY.                                          normal mode.
batch <----------------------------------------- Switches NOS 2 to batch
RFL,0.                                           mode.
/get,jobfile <---------------------------------- Gets the file JOBFILE.
/xedit,jobfile <-------------------------------- Calls a text editor to
 XEDIT 3.1.00                                    display the job file
?? p*                                            contents.
ADEY,STABC.
USER,USER=PUBS124,ACCOUNT=ACCT933,PASSWORD=XYZ.
RESOURCE,TL=5.
FTN200.
LOAD,LIB=F200LIB.
GO.
--EOR--
      PROGRAM LOOP
      K = 0
      DO 10 I=1,5
      READ 100,J
  100 FORMAT(I1)
   10 K = J + K
      PRINT 200
  200 FORMAT(' THE SUM IS')
      PRINT 300,K
  300 FORMAT(1X,I2)
      STOP
      END
--EOR--
1
2
3
4
5
END OF FILE
```

Figure F-1. Example of a NOS 2 Interactive Session Submitting a CYBER 200 Job (Sheet 1 of 2)

```
?? end
JOBFILE IS A LOCAL FILE
/submit,jobfile,e <-------------------------------------- Submits the job for
 13.08.17  SUBMIT COMPLETE.  JOBNAME IS ABGX                execution.
/enquire,jsn                                                Displays job status.
 JSN SC CS DS LID STATUS          JSN SC CS DS LID STATUS
ABGX.B.   .RB.ABC.INPUT QUEUE    ABGU. T.ON.BC.   .EXECUTING
/enquire,jsn
 JSN SC CS DS LID STATUS          JSN SC CS DS LID STATUS
ABGU. T.ON.BC.   .EXECUTING
/enquire,jsn
 JSN SC CS DS LID STATUS          JSN SC CS DS LID STATUS
ABKK.B.   .RB.C17.PRINT QUEUE    ABGU. T.ON.BC.   .EXECUTING
/qget,abkk,pr <------------------------------------------- Output file becomes local
QGET,ABGK.                                                  file.
/xedit,abkk <--------------------------------------------- Calls a text editor to
 XEDIT 3.1.00                                               display the output file.
?? l-adey- <---------------------------------------------- Locates the dayfile.
--EOR--
--EOR--
1 13.06.31  RSYSL592       VSYSL592       012306  PUBLIC   G    ADEY
   10/06/83
?? p* <--------------------------------------------------- Lists the dayfile.
1 13.06.31  RSYSL592       VSYSL592       012306  PUBLIC   G    ADEY
   10/06/83
  13.06.32 RESOURCE,TL=10.
  13.06.32 FTN200.
  13.06.35 FORTRAN 200 CYCLE L592     BUILT 07/27/83 14:31
  13.06.37 COMPILING LOOP
  13.06.37       NO ERRORS
  13.06.46     0.049 SECONDS COMPILATION TIME
  13.06.50  ALL DONE
  13.06.52 LOAD,LIB=F200LIB.
  13.06.55  LOAD R2.1 CYCLE L592
  13.07.03  ALL DONE
  13.07.04 GO.
  13.07.09    FILE 42GO      EXTENDED, NEW LENGTH    276
  13.07.09    FILE 42GO      EXTENDED, NEW LENGTH    324
  13.07.10 STOP
  13.07.13  ALL DONE
  13.07.14 SYSTEM TIME UNITS (STU)                5.188
  13.07.14 $$COMPLETE$$
END OF FILE
?? l-the sum is-2 <--------------------------------------- Locates the program output.
    00008     200 FORMAT(' THE SUM IS')
                                    0001/00008
--EOR--
--EOR--
 THE SUM IS
?? p2 <--------------------------------------------------- Displays the program output.
 THE SUM IS
 15
?? end
ABKK IS A LOCAL FILE
/route,abkk,dc=lp <--------------------------------------- Routes the output file to a
 ROUTE COMPLETE.                                            printer.
/bye <---------------------------------------------------- Requests IAF logout.


UN=USER1   LOG OFF    13.23.10.
JSN=ABGU   SRU-S         1.693.

IAF     CONNECT TIME:  00.15.46. <-------------------------- IAF logout.
LOGGED OUT.
```

Figure F-1. Example of a NOS 2 Interactive Session
Submitting a CYBER 200 Job (Sheet 2 of 2)

The second statement in the job file is the USER statement. It specifies the CYBER 200 user name, account name, and the user password.

The third statement in the job file is the RESOURCE statement. The parameter TL=5 specifies a five-second time limit.

As described earlier, the next three control statements (FTN200, LOAD, and GO) compile, load and execute a FORTRAN 200 program. The FORTRAN 200 program source is listed between two --EOR-- indicators. --EOR-- is the indicator the XEDIT editor uses to display an end-of-record delimiter. One end-of-record delimiter separates the program from the control statements and another separates the program from the input data (here, a sequence of integers).

Because no input file is specified on the FTN200 statement, the program to be compiled is read from file INPUT (the job file). The LOAD statement generates an executable program on the default file GO. The GO statement executes the program on file GO; the program reads its input data from file INPUT which now contains the sequence of integers from the job file.

The NOS 2 command SUBMIT,JOBFILE,E submits the CYBER 200 job file for execution. The NOS 2 command ENQUIRE,JSN displays the status of the user's jobs. After the job output file is returned to the NOS 2 print queue, the NOS 2 command, QGET,ABGK,PR changes the output file to a local file. The user locates and displays the dayfile showing that the job executed normally. He then locates and displays the program output. Finally, he routes the entire output file to a printer and logs out.

FORTRAN CONTROL STATEMENT

This section describes the FORTRAN control statement, its parameters, and the kinds of output information that it can direct the FORTRAN 200 compiler to produce.

DEFAULTS

There are two kinds of parameter defaults. The first kind occurs when the parameter and its value are both omitted. The second kind occurs when the parameter is given but the value is omitted.

The keywords, their minimal abbreviations, and their defaults are summarized in table F-1 and discussed in detail in the following paragraphs.

TABLE F-1. KEYWORD ABBREVIATIONS AND PARAMETER DEFAULTS

Keyword	Minimal Abbreviation	First Default (Omit Keyword and Value)	Second Default (Omit Value Only)
ANSI	ANSI	ANSI=0	ANSI=W
BINARY	B	B=BINARY/16	B=BINARY/16
C64	C64	C64=0	C64=1
DO	DO	DO=0	DO=1
ERRORS	E	E=OUTPUT/16	E=ERRS/16
ELEV	ELEV	ELEV=W	ELEV=F
F66	F66	F66=0	F66=1
INPUT	I	I=INPUT	I=COMPILE
LIST	L	L=OUTPUT/336	L=LIST/336
LO	LO	LO=S	LO=SX
OPTIMIZE	OPT	OPT=0	OPT=1
SC	SC	SC=0	SC=1
SDEB	SDEB	SDEB=0	SDEB=1
SYNTAX	SYN	SYN=0	SYN=1
TM	TM	TM=HOST	TM=205
UNSAFE	UNS	UNS=0	UNS=1

BINARY

Specifies the name of the file to which the compiler writes the binary object code.

The disposition of the binary file depends on the kind of file. If the file is an attached permanent file, then it is used, and any explicit or implied length specification is ignored. If the file is an existing local file, then the compiler returns it and creates a new file. The compiler creates a new local file in all other cases. The compiler performs these functions by calling Q5GETFIL with the RETURN parameter specified as described in the VSOS reference manual, Volume 1.

The valid options are:

BINARY=lfn/len

Writes object code on the file lfn, initially setting the length at len blocks. The option len can be an integer constant or a hexadecimal constant prefixed with a #. A block is 512 consecutive words. The len default is 16 blocks. The compiler passes len as the file length in its call to Q5GETFIL.

BINARY=lfn

Same as BINARY=lfn/16.

BINARY

Same as BINARY=BINARY/16.

omit

Same as BINARY=BINARY/16.

BINARY=0

Generates no object code.

NOTE

The BINARY parameter might conflict with the SYNTAX parameter. For instance, when BINARY=0, the only acceptable option for the SYNTAX parameter is 1. Otherwise, warnings are generated.

ERRORS

Specifies a file name for recording the error information.

The file length that you specify with ERRORS is not necessarily always what you get; there is one exception. If you have specified the same file name with LIST, the actual file length will become the larger of the two. If your LIST specifies a larger file than your ERRORS specification, the LIST file size prevails.

The disposition of the errors file depends on the kind of file. If the file is an attached permanent file, then it is used, and any explicit or implied length specification is ignored. If the file is an existing local file, then the compiler returns it and creates a new file. The compiler creates a new local file in all other cases. The compiler performs these functions by calling Q5GETFIL with the RETURN parameter specified.

The valid options are:

ERRORS=lfn/len

Writes error diagnostics on the file lfn, initially setting the length at len blocks if there is an error of at least ELEV severity. The option len can be an integer constant or a hexadecimal constant prefixed with a #. A block has 512 words. The len default is 16 blocks. The compiler passes len as the file length in its call to Q5GETFIL.

ERRORS=lfn

Same as ERRORS=lfn/16.

ERRORS

Same as ERRORS=ERRS/16.

omit

Same as ERRORS=OUTPUT/16.

LIST

Specifies the file name to which the compiler can write the the source listing and other requested information except diagnostics.

The file length that you specify with LIST is not necessarily always what you get; there is one exception. If you have specified the same file name with ERRORS, the actual file length will become the larger of the two. If your ERRORS specifies a larger file than your LIST specification, the ERRORS file size prevails.

The disposition of the list file depends on the kind of file. If the file is an attached permanent file, then it is used, and any explicit or implied length specification is ignored. If the file is an existing local file, then the compiler returns it and creates a new file. The compiler creates a new local file in all other cases. The compiler passes len as the file length in its call to Q5GETFIL.

SYNTAX

Instructs the compiler to perform a quick syntax check on the source program. The valid options are:

SYNTAX=1

Performs a full syntactic scan but generates no BINARY file.

SYNTAX=0

Compiles completely.

SYNTAX

Same as SYNTAX=1.

omit

Same as SYNTAX=0.

NOTE

The SYNTAX parameter might conflict with the BINARY, LO, OPTIMIZE, and UNSAFE parameters. When SYNTAX=1, the only acceptable options for these parameters are BINARY=0, OPTIMIZE=0, UNSAFE=0, LO=S, LO=X, and LO=SX.

Any other options generate warnings.

CONTROL STATEMENT EXAMPLES

Figures F-2 and F-3 show examples of the use of FORTRAN 200 control statements.

Figure F-2 shows the all-default case. The FTN200. statement alone assigns default values to each option. The figure lists the default values.

The ERRORS file of figure F-2 has a default length of 16, but the ERRORS file is the same file as the LIST file, and the greater length takes precedence.

Figure F-3 shows an example of a FORTRAN 200 control statement with some options specified and others allowed to default. The list shows all values, including the default values.

Although the ERRORS file of figure F-4 has been specified with a length of 16, the LIST file is the same file and its default length is 336; the length of 336 takes precedence.

FTN200. is equivalent to:

```
FTN200,ANSI=0,
    BINARY=BINARY/16,
    C64=0,
    DO=0,
    ERRORS=OUTPUT/336,
    ELEV=W,
    F66=0,
    INPUT=INPUT,
    LIST=OUTPUT/336,
    LO=S,
    OPTIMIZE=0,
    SC=0,
    SDEB=0,
    SYNTAX=0,
    TM=HOST,
    UNSAFE=0.
```

Figure F-2. Control Statement Example With Default Values

```
FTN200,I=SOURCE,L=LOOK,OPT=1,LO=AS,SC,
    TM=205,B=LGO/#AA,C64,E=LOOK/16
```

is equivalent to (including defaults):

```
FTN200,ANSI=0,
    BINARY=LGO/#AA,
    C64=1,
    DO=0,
    ERRORS=LOOK/336,
    ELEV=W,
    INPUT=SOURCE,
    LIST=LOOK/336,
    LO=AS,
    OPTIMIZE=DPRSV,
    SC=1,
    SYNTAX=0,
    TM=205,
    UNSAFE=0.
```

Figure F-3. Control Statement Example

Q8NORED

The subroutine Q8NORED supresses the normal file size reduction that is performed at the completion of program execution. Normally, files created by a program are initially 128 512-word blocks long, and are reduced at the completion of program execution to a minimal size. If Q8NORED is called in the program, the files specified in the Q9NORED call retain their initial size, rather than being reduced. See figure F-4 for the format of a Q8NORED call.

```
CALL Q8NORED
   or
CALL Q8NORED(uid1, ... ,uidn)

uid  An external unit identifier
```

Figure F-4. Q8NORED

If no logical unit numbers are specified in the Q8NORED call, all files retain their initial size, rather than being reduced.

This appendix presents an introduction to the vector processing capabilities of FORTRAN 200. The first part of this appendix introduces elementary concepts of vectors. The second part describes what kinds of program constructs can be vectorized by the automatic vectorization feature of the CYBER 200 compiler, and how they are vectorized for the greatest increase in execution speed. It also explains what kinds of program constructs inhibit vectorization. The third part gives a detailed description of how the hardware instructions accomplish vectorization; this part is intended for those desiring more information about vectorization concepts. The fourth part describes the vectorization report messages.

The discussion assumes you are familiar with a version of FORTRAN 77, but unfamiliar with vectors and vector programming. To apply the vectorization concepts presented in this appendix, your program must be written in FORTRAN 200 and compiled and executed on the CYBER 200.

INTRODUCTION TO VECTORS

FORTRAN programs frequently contain many DO loops that perform operations on arrays. If the arrays are large, the loops can be quite time-consuming. The vector processing capability of the CYBER 200 provides a way of making array operations within DO loops considerably faster.

WHAT IS A VECTOR?

To a mathematician, a vector is a set of N numbers uniquely defining a distance and a direction in an N-dimensional space. A FORTRAN programmer generally uses the term vector of length N to mean any one-dimensional set of N numbers. Arrays, and portions of arrays, can be thought of as vectors. For example, the following statements define a 5-element vector and store some values into it:

```
DIMENSION A(5)
DATA A /6.0, 1.0, 2.0, 9.0, 7.0/
```

The vector can be pictured as shown in figure G-1.

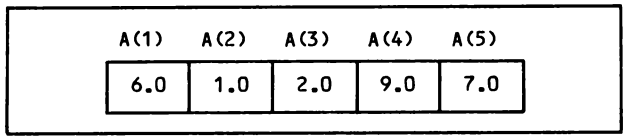


Figure G-1. Vector Example

A vector can be contrasted with a scalar. A scalar is simply a single value, such as a constant, variable, or array element. The familiar arithmetic operations in your existing FORTRAN programs are known as scalar operations. In the following example, the variables R and S are scalars. The addition of R and S is a scalar operation and the result is stored in the scalar variable T:

```
DATA R, S/1.0, 2.0/
 .
 .
 .
T = R + S
```

Scalar operations involve single-valued operands and calculate single-valued results.

Vector operations, however, operate on vectors and calculate results for vectors. Vectors usually consist of more than one element. The following example shows an array operation that can internally be performed as a single vector operation:

```
      DIMENSION A(5), B(5), C(5)
      DATA A/6.0, 1.0, 2.0, 9.0, 7.0/
      DATA B/0.0, 10.0, 8.0, 4.0, 1.0/
       .
       .
       .
      DO 10 I=1,5
         C(I) = A(I) + B(I)
10    CONTINUE
```

This program defines three arrays: A, B, and C. The contents of arrays A and B are added and the result is stored in the array C. The addition can be performed as vector addition because A and B can be represented as vectors in the machine. In a vector addition, elements of one vector are added to corresponding elements of another vector. The vector addition can be pictured as shown in figure G-2.

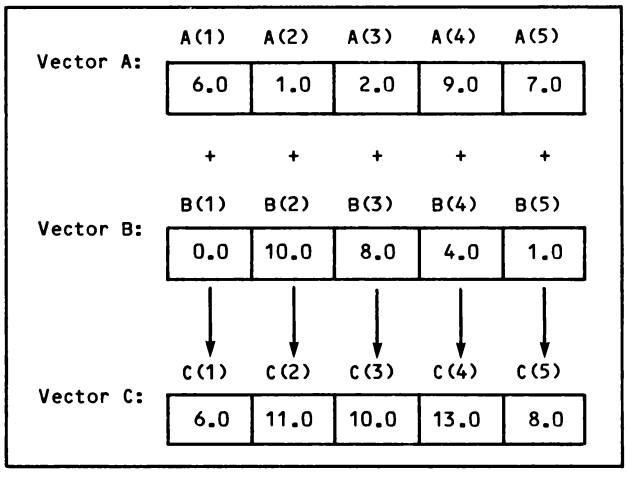


Figure G-2. Vector Addition

Thus, the result of this figure is a vector having the values 6.0, 11.0, 10.0, 13.0, and 8.0.

WHY ARE VECTOR OPERATIONS FASTER THAN SCALAR OPERATIONS?

Vector operations on the CYBER 200 are performed using a pipeline type of processor. To see why the pipeline processor executes faster, consider the following DO loop, which adds the number 5.0 to each element of a 10-element array A, and stores the results in the array B:

```
      DIMENSION A(10), B(10)
       .
       .
      DO 10 I=1,10
        B(I) = A(I) + 5.0
10    CONTINUE
```

Consider a single calculation within this loop, the calculation B(1) = A(1) + 5. The FORTRAN compiler translates this statement into a sequence of machine instructions. To simplify the example, we assume a hypothetical computer on which the assignment statement B(1) = A(1) + 5.0 is translated into the following machine instructions:

```
LOAD     A(1)
ADD      5.0
STORE    B(1)
BRANCH
```

Now focus on the ADD instruction. On a scalar processor, the ADD instruction for a particular operand must be complete before the ADD for the next operand can begin. For example, the ADD for A(1) must be complete before the ADD for A(2) can begin.

On a pipeline processor, however, the ADD is divided into a sequence of steps. In this example, we assume five steps, with each step requiring one cycle of execution time. Thus, the ADD instruction can be pictured as shown in figure G-3.

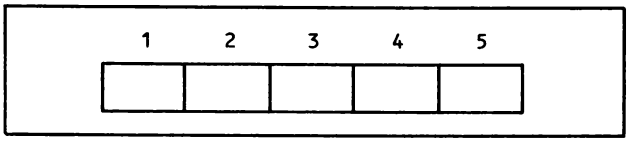


Figure G-3. Add Instruction

Each box represents one step of the ADD instruction.

In scalar processing, all steps must be complete for an operand before processing of the next operand can begin; a single ADD takes 5 cycles. With pipeline processing, after a step is complete, that step can process the next operand. In our example, at the beginning of the first cycle, the first step of the ADD instruction begins processing A(1). See figure G-4.

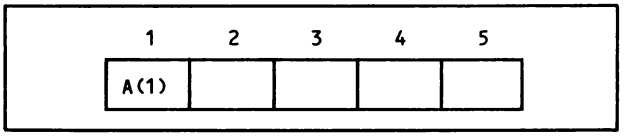


Figure G-4. Add Instruction - Cycle 1

When step 1 is complete for A(1), A(1) moves on to step 2 and step 1 begins for A(2). See figure G-5.

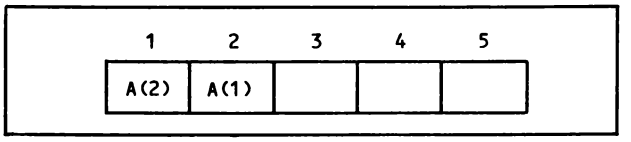


Figure G-5. Add Instruction - Cycle 2

At the end of cycle 2, A(1) moves on to step 3, A(2) moves to step 2, and step 1 begins processing the next operand. Diagrams for cycles 3, 4, and 5 are shown in figure G-6.

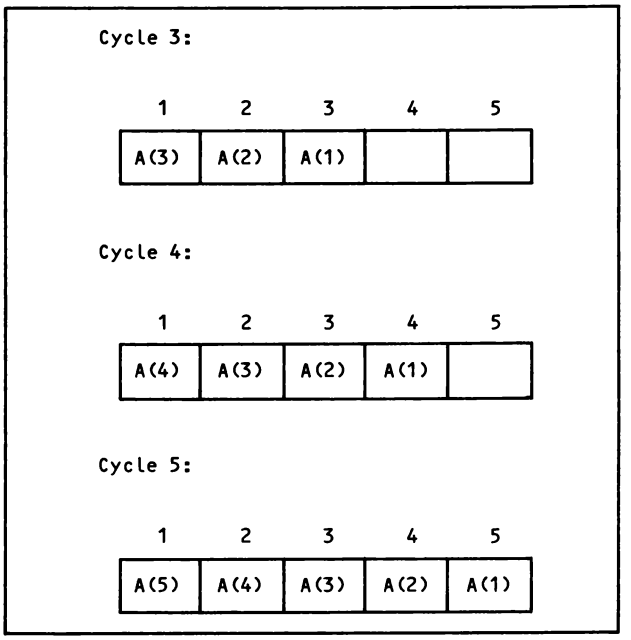


Figure G-6. Add Instruction - Cycles 3, 4, and 5

At cycle 5, five operands are being processed by the ADD instruction. At cycle 6, the operation A(1) + 5.0 is complete and the result is ready to be stored into B(1). See figure G-7.

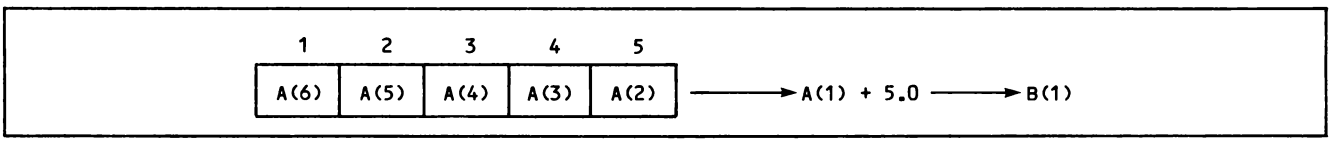


Figure G-7. Add Instruction - Cycle 6

Subsequently, an ADD is completed every cycle. The hardware that allows for processing multiple operands by a single instruction is known as a pipeline.

Not only are 205 pipes segmented, but memory access is by groups of words (8 words per operand) so that memory access is faster.

All the elements of A can be loaded in two or three memory cycles (depending on how A is positioned in memory), instead of ten separate cycles for each element.

VECTORIZATION

Vectorization is the process of transforming scalar operations into vector operations. The CYBER 200 compiler has the capability of automatically vectorizing a program using the vector-processing hardware of the CYBER 200. This section describes how the vector-processing capabilities available to the CYBER 200 compiler analyzes your DO loops which are candidates for vectorization. This section also describes constructs in DO loops that cannot be vectorized.

THE FORTRAN VECTORIZER

The CYBER 200 compiler has the capability of automatically vectorizing DO loops in your program. Automatic vectorization is performed by a component called the vectorizer.

To select automatic vectorization of your program, specify OPTIMIZE = V (or OPT=V) on the compilation command. DO loops are described in the next subsection. When OPT=V is selected, and the compiler finds no FATAL compilation errors, vectorization can occur. The vectorizer first scans the source code for DO loops. If there are no DO loops in your program, no vectorization will occur. Other types of loops are not vectorized.

Specifying OPT=V will usually result in faster execution time of the resulting object code but it does increase compilation time. See NOTE below.

A vectorization report automatically follows the source code if OPT=V is selected.

NOTE

Because of the startup overhead involved in vector processing, vector operations may not be faster than scalar operations for short vectors. However, vector operations are almost always faster for longer vectors; the longer the vectors, the greater the time savings. Also, the time savings achieved depends on the operations being performed.

DO LOOPS

DO loops must meet certain basic criteria before they are considered for vectorization. If the DO loop does satisfy these criteria, the vectorizer will analyze it further to determine if it can be vectorized. See figure G-8.

Once the DO loop meets these criteria, it is a candidate for vectorization. However, some types of statements within the loop can still prevent vectorization of the DO loop. For a description of statements that cannot be vectorized, see Items that Inhibit Vectorization.

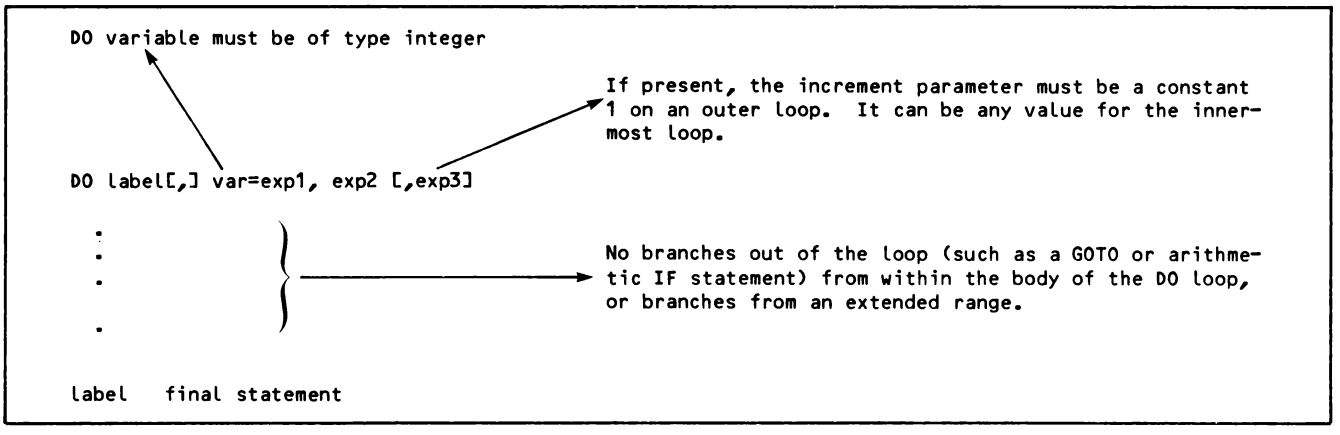


Figure G-8. DO Loop Criteria

NOTE

Multidimensional arrays in DO loops should be used with care. Whenever possible, the innermost loop should iterate over the first subscript, the next outermost loop should iterate over the second subscript, and so forth. A loop that iterates in this way will execute faster than one which iterates in some other order. This is because each reference to the array is made to the next closest array element (arrays are stored in columnwise order). For example, the following statements do not reference elements in the order they are stored:

```
      DIMENSION A(20,30,40),B(20,30,40)
        :
      DO 10 I=1,20
      DO 10 J=1,30
      DO 10 K=1,40
         A(I, J, K) = B(I, J, K)
10    CONTINUE
```

The following example ensures that elements are referenced in the order they are stored and therefore executes faster:

```
      DIMENSION A(20,30,40), B(20,30,40)
      :
      DO 10 K=1,40
      DO 10 J=1,30
      DO 10 I=1,20
        A(I, J, K) = B(I, J, K)
10    CONTINUE
```

The benefits of accessing arrays in storage order are achieved regardless of vectorization.

DATA TYPES

The data type of values used within DO loops affects the amount of vectorization that occurs. A statement containing a character or double precision operand cannot be vectorized. Real, integer, and half precision types are more likely to be vectorized. Logical and complex data can also be vectorized, but not as often as the above mentioned types.

Internal operations that accomplish vectorization can be type-specific; that is, they work only for a certain data type. The descriptions of the internal operations in the Vector Operations section indicate any type-specific restrictions. If a statement contains a data type that cannot be vectorized, the vectorization report will indicate that the DO loop was not vectorized. It is important to remember that implicit conversion does not change the data type of the array. For example, in the following assignment statement:

```
A = 1
```

the compiler assigns the real value 1.0 to A; even though you used an integer in the assignment statement, the resulting data type of A is real.

ITEMS THAT INHIBIT VECTORIZATION

Some loops have statements that cannot be vectorized. The types of statements that cannot be vectorized are as follows:

A statement that causes an external reference cannot be vectorized; for example, a CALL statement, a non-intrinsic function reference, and all input/output statements cause external references.

Statements that reference elements of type character or double precision cannot be vectorized.

A statement or group of statements that is executed an indeterminate number of times cannot be vectorized; for example, statements controlled by a logical or arithmetic IF cannot be vectorized.

Statements in a recurrence cycle cannot be vectorized. Recurrence cycles are described in the following subsection.

The vectorization report indicates statements that cannot be vectorized.

Recurrence Cycles

A recurrence cycle exists when one or more statements depend on one another. This can occur when more than one iteration of a loop causes the same location in memory to be accessed. Recurrence cycles prevent vectorization because all statements in a recurrence cycle must be executed for a particular iteration of a loop before the next iteration of the loop can take place. In other words, these statements cannot be executed in parallel and therefore cannot be vectorized.

For example:

```
      DO 10  I=1,N
          A(I) = B(I) + C(I)<------ Statement 1
          B(I+1) = A(I)<---------- Statement 2
10    CONTINUE
```

By unravelling the first three passes through the loop, we can more clearly see the recurrence cycle as shown in figure G-9.

The value of B(2) is used in statement 1 after it is assigned a value in statement 2 of the previous iteration (arrow labeled 1); the value of A(I) is used in statement 2 after it is assigned a value in statement 1 of the same iteration (arrow labeled 2). Due to this interdependence, there is a recurrence cycle.

However, if statement 2 is changed to B(I-1) = A(I), then no recurrence cycle exists. For example:

```
      DO 10  I=1,N
          A(I) = B(I) + C(I)<------ Statement 1
          B(I-1) = A(I)<---------- Statement 2
10    CONTINUE
```

Although A(I) is referenced in both statements of the DO loop; no recurrence cycle exists. By unraveling the first three passes through the loop, we can see that a recurrence cycle is not created as shown in figure G-10.

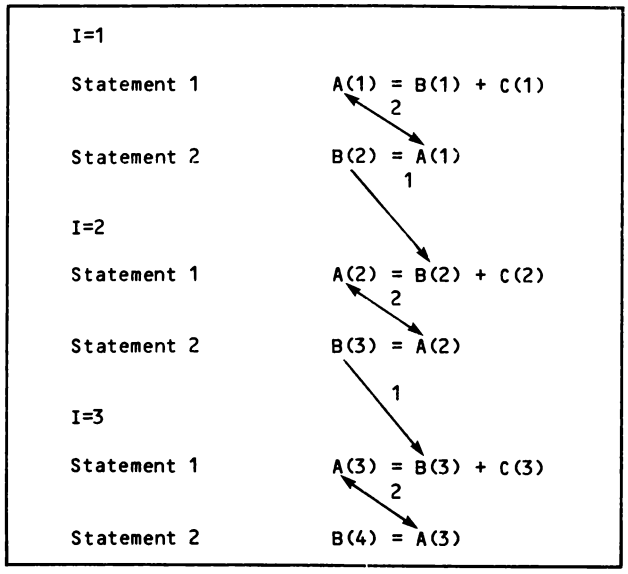


Figure G-9. Recurrence Cycle

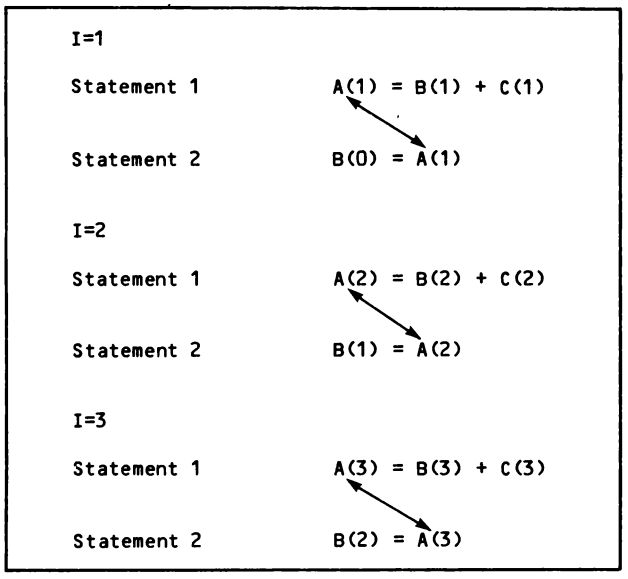


Figure G-10. Recurrence Cycle - Not Created

This is because in the I=2 iteration, B(2) appears to the right of the equal sign, but it has not been computed in a previous iteration of the same loop. Recurrence cycles prevent vectorization of the loops contained in the cycle. The vectorization report indicates one statement of a recurrence cycle.

Self recurrence is a recurrence cycle that occurs on one statement. For example:

```
      DO 10 J=1,5
         A(J) = A(J-1) * B(J)
10    CONTINUE
```

The value of A(J-1) is computed in a previous iteration of the loop. Self-recurrence occurs for A; therefore, the loop cannot be vectorized.

VECTOR OPERATIONS

The vectorization component of the compiler vectorizes a program by scanning the object code and replacing scalar operations by equivalent vector operations. Certain of these operations are performed by the CYBER 200 hardware. The hardware vector operations include scattering, gathering, reduction, interval vector generation, scalar expansion, broadcasting, intrinsic function promotion, and stripmining. These hardware operations are described in this subsection. This subsection is intended for those readers desiring more detailed information of the vector-processing methods used to vectorize. Many of the terms described in this subsection are used in the vectorization report described later in this appendix.

NOTE

The examples presented in this section are for illustrative purposes only. For actual examples of complete vectorized programs, see the vectorization report discussion in this appendix.

SCATTERING AND GATHERING

The scatter and gather operations are used to vectorize statements that reference noncontiguous elements of an array. The scatter instruction is used when contiguous elements of an array are stored into noncontiguous elements of another array. For example:

```
      DIMENSION A(3,3), B(3), C(9)
      DO 10 I=1,3
         DO 20 J=1,3
            A(I,J) = B(J)
20       CONTINUE
10    CONTINUE
```

In this example, every element of B is assigned to every third element of A in the inner DO loop. The assignment statement assigns elements of array B to the noncontiguous elements of array A. This is shown internally in figure G-11.

The gather instruction is used to gather noncontiguous elements of an array into an internal temporary vector. For example:

```
      DO 20 J=1,3
        B(J) = C(J**2) + B(J)
20    CONTINUE
```

This is shown internally in figure G-12.

THE INTERVAL VECTOR

Operations involving the assignment of a sequence of integers are vectorized using an interval vector. For example:

```
      DO 10 I=1,1000
         A(I) = I
10    CONTINUE
```

In this example, the integers 1 through 1000 are assigned to corresponding array elements. This is shown in figure G-13.

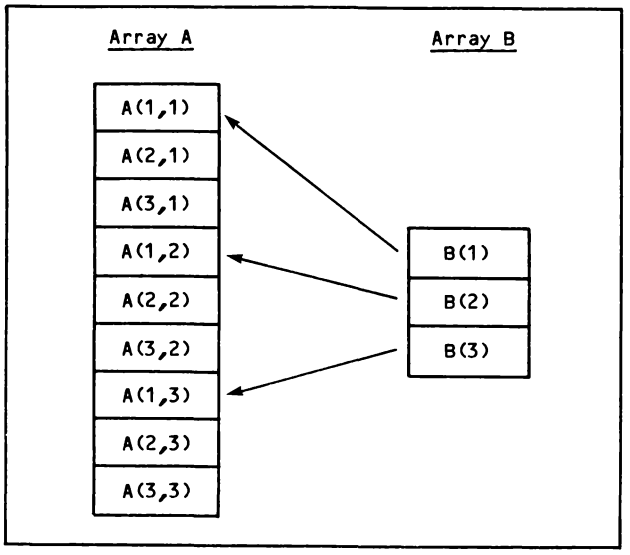


Figure G-11. Scatter Instruction - Internal

An interval vector is generated for the innermost loop's control variable, or for the scalar variable which assigned an arithmetic sequence. For example,

```
        J=0
        DO 10 I=1,500
        J=J+2
10      A(J)=I
```

Internally:

```
        A(2)=1
        A(4)=2
          .
          .
          .
        A(1000)=500
```

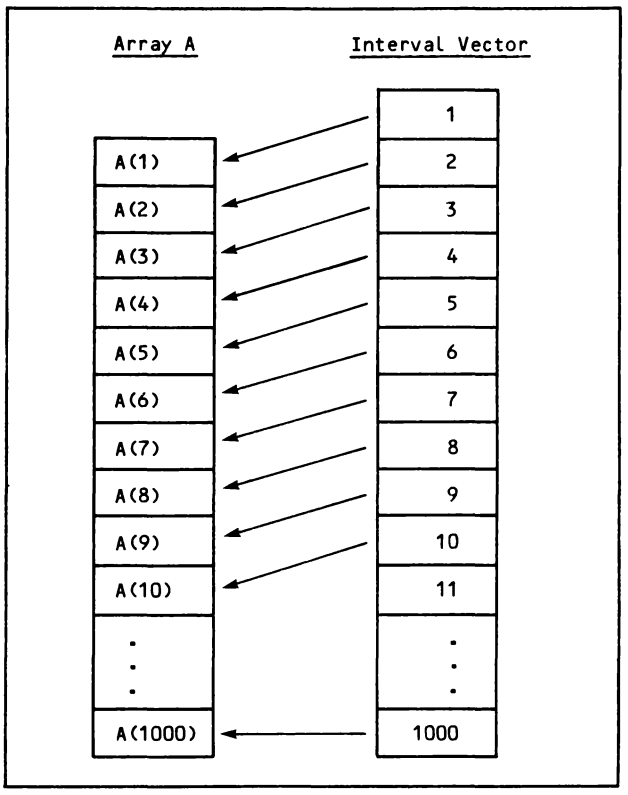


Figure G-13. Interval Vector

REDUCTION

Some loops are vectorized through operations known as sum, product, or dot product reduction. Reduction is commonly used when vector elements are summed into a scalar variable. Reduction uses the SUM, PRODUCT, or DOT hardware instructions to

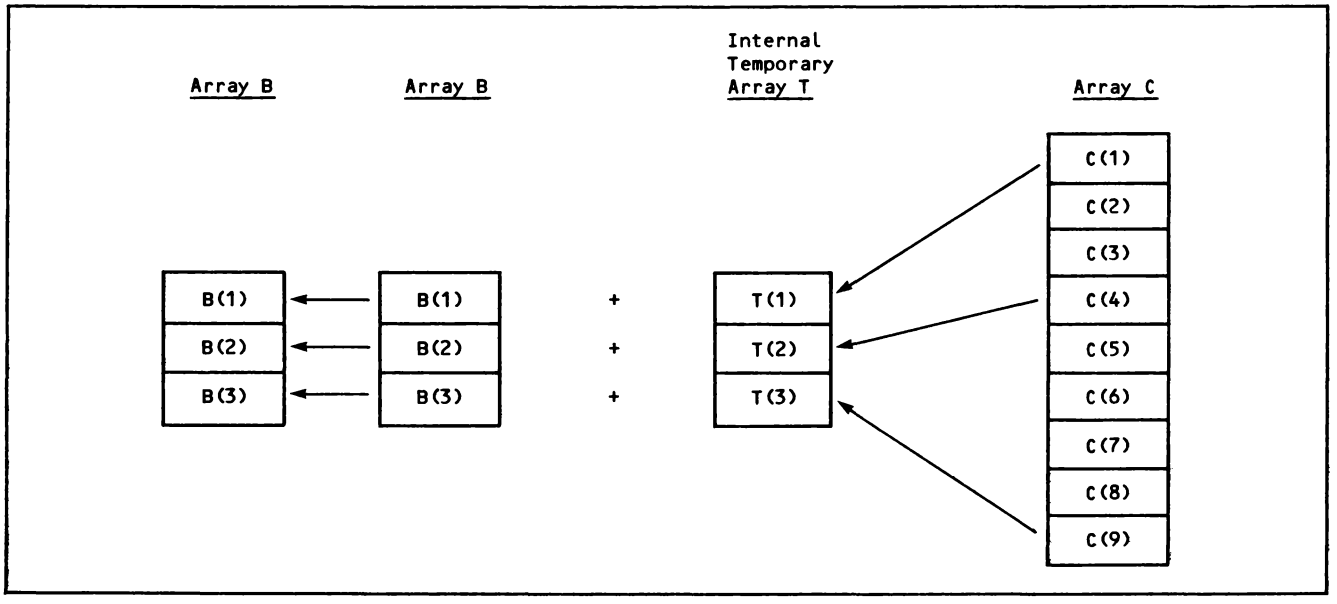


Figure G-12. Gather Instruction - Internal

reduce all elements of the vectors, and then assigns the result to a scalar variable. Reduction works only on type real, integer, and half precision data. For example:

```
      REAL  B(100), C(100)
      A = 0.0
      DO 10 I=1,100
          A = A + B(I) * C(I)
            .
            .
            .
10    CONTINUE
```

The product of B(I) * C(I) is summed into an internal temporary scalar arbitrarily named TEMP shown in figure G-14.

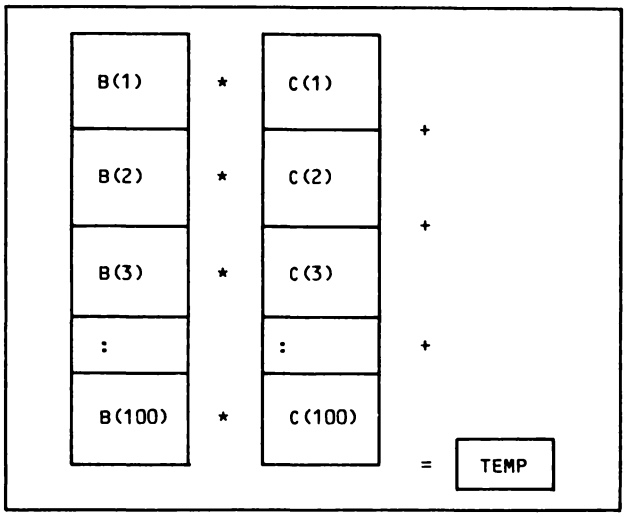


Figure G-14. Internal Temporary Scalar

The hardware instruction DOT computes the sum of all pairwise multiplications of B and C; the sum is added to and assigned to the scalar varible A and the process is then complete:

SUM (B(1) * C(1) + B(2) * C(2) + ... + B(100) * C(100)) + A ⟶ A

NOTE

Using the hardware instructions SUM, PRODUCT and DOT can sometimes generate different results from the scalar execution of the summation in the DO loop due to a different order and normalization of summing the elements (they are summed in parallel without normalization).

SCALAR EXPANSION

Scalar expansion (or promotion) is a technique that is used in the vectorization of expressions containing scalar operands. The vectorizer replaces a scalar with an internal temporary vector. This is done so that each trip through the loop can reference a different memory location, thereby allowing vector processing. For example:

```
      DO 5 I=1,10
          A=B(I) + R(I) <------- Statement 1
          C(I) = A + D(I) <----- Statement 2
5     CONTINUE
```

The vectorizer performs the assignment in statement 1 using a scalar expansion of A and the values in B and R. This is shown internally in figure G-15.

The value in A(10) is the value assigned to the scalar A upon loop termination; the values of A(1) through A(9) are not used in calculating the value of A at loop termination.

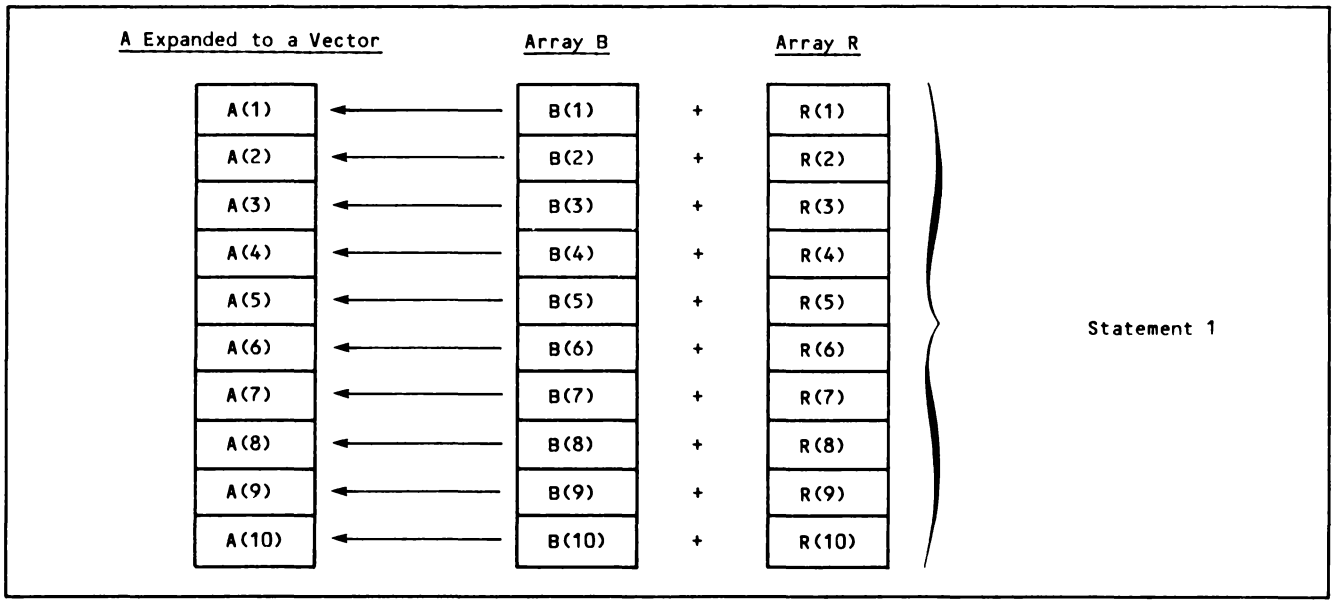


Figure G-15. Scalar Expansion

BROADCASTING

Broadcasting is the treatment of a scalar value as a temporary vector. The purpose of broadcasting is to perform vector operations when one of the operands has an invariant value.

For example:

```
      B=1.0
      DO 10 I=1,10
          A(I) = B <---------- Statement 1
10    CONTINUE
```

In statement 1, the value of B (in this case 1.0) is assigned to an element of array A on each iteration of the loop. The statement is vectorized using the broadcast operation as shown in figure G-16.

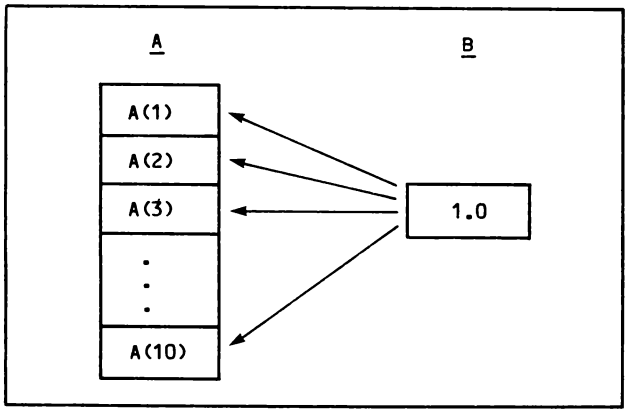


Figure G-16. Broacast Operation Example

INTRINSIC PROMOTION

The vectorizer replaces certain intrinsic function references and expressions involving exponentiation with an equivalent vector version. This operation is called intrinsic promotion.

For example:

```
      DO 10 I=1,10
      A(I) = SIN (B(I))
10    CONTINUE
```

The vector version of the SIN intrinsic can process a vector of values as shown in figure G-17.

The math routines called by exponentiation operations also have vector equivalents. Thus, for example, the statement

```
A(I) = X(I) ** 2
```

can be vectorized.

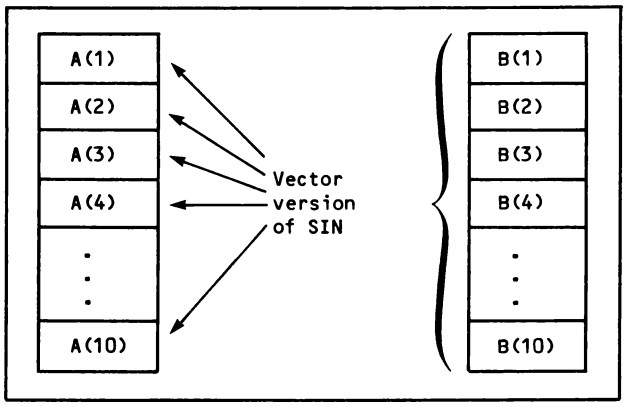


Figure G-17. Intrinsic Promotion Example

STRIPMINING

Stripmining is used by the vectorizer to vectorize loops where the iteration count either is unknown (as in an assumed-size array) or exceeds the hardware vector length of 65535. Dummy argument arrays that are dimensioned to size 1 are treated as assumed-size arrays and are candidates for stripmining in the vectorization process. For example (vector length exceeds 65535):

```
      DO 10 I=1,100000
          A(I)=B(I)
10    CONTINUE
```

Internally, two separate vector operations are performed. See figure G-18 for the first vector operation.

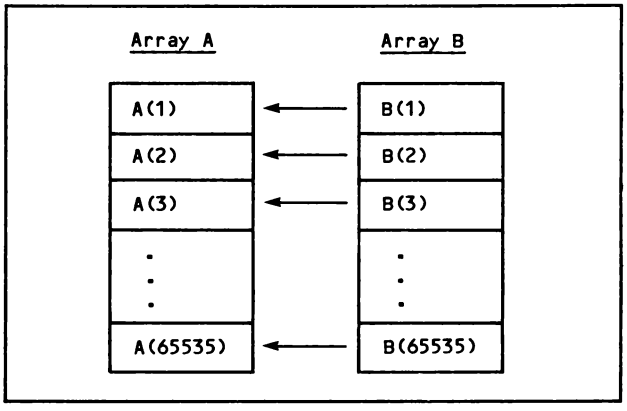


Figure G-18. First Vector Operation

See figure G-19 for the second vector operation.

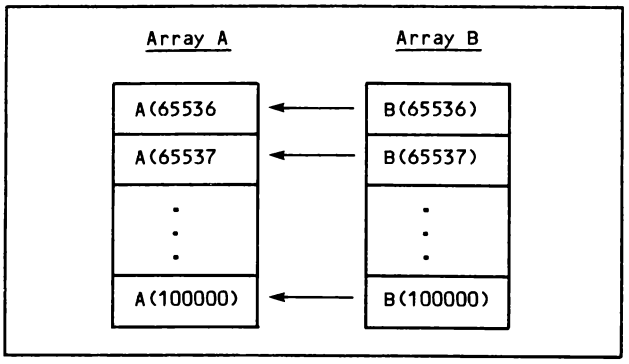


Figure G-19. Second Vector Operation

LOOP COLLAPSE

When a multidimensional array is vectorized, active contiguous dimensions are collapsed to a single dimension. Active dimensions are dimensions that are referenced in the DO loop. Loop collapse occurs only when the loop bounds span the array dimension. For example:

```
      DIMENSION A(5,15), B(5,15)
      DO 10 I=1, 10
          DO 20 J=1, 5
           (J,I) = B(J,I) + 5
20        CONTINUE
10    CONTINUE
```

VECTORIZER REPORT

Figure G-20 is an example of a FORTRAN 200 program and the vectorizer report it creates.

The line marked [A] indicates the number of loops in the source program, including the explicit DO loops and implicit loops of an array assignment. Implied loops of a READ or WRITE statement are not included.

The line marked [B] indicates all loops that are vectorized. If no loops are vectorized, the message NO LOOPS WERE VECTORIZED is printed and the report continues with the stacklib section.

Each vectorized loop is listed by indicating the first source line number (the number in the left margin) of the loop. For explicit DO loops this is the DO statement. For array assignments, it is the assignment. This is followed by a count of loops that are vectorized beginning at that line. For explicit DO loops, the count is always 1. For an array assignment, the count is the number of dimensions vectorized.

The line marked [C] indicates all loops that are stacklibbed. If no loops are vectorized, the message NO LOOPS WERE STACKLIBBED is printed and the report continues with the nonvectorizable section.

```
FORTRAN 200 CYCLE 235L3    BUILT 06/03/87 11:32   SOURCE LISTING       VX        COMPILED 06/16/87 18:50   SAFE OPT=V     PAGE     1

   00001            PROGRAM   VX                                                                                    0001/00001
   00002            REAL      A(10,10,10),B(10,10),C(10),D(10),S                                                    0001/00002
   00003            INTEGER   I,J,K                                                                                 0001/00003
   00004            A(*,*,1)=B                                                                                      0001/00004
   00005            A(1,*,*)=B                                                                                      0001/00005
   00006            DO 1 K=1,10                                                                                     0001/00006
   00007              DO 1 J=1,10                                                                                   0001/00007
   00008                DO 1 I=1,10                                                                                 0001/00008
   00009   1              S=S+A(I,J,K)                                                                              0001/00009
   00010            DO 2 I=1,10                                                                                     0001/00010
   00011   2          C(I)=C(I-1)+D(I)                                                                              0001/00011
   00012            DO 3 J=1,10                                                                                     0001/00012
   00013              DO 3 I=2,9                                                                                    0001/00013
   00014   3            B(I,J)=1                                                                                    0001/00014
   00015            DO 4 J=1,10                                                                                     0001/00015
   00016              DO 4 I=1,10                                                                                   0001/00016
   00017   4            CALL F(A(I,J,1),B(I,J))                                                                     0001/00017
   00018            END                                                                                             0001/00018


[A]  12 LOOPS WERE EXAMINED FOR POSSIBLE VECTORIZATION.

[B]  7 LOOPS WERE VECTORIZED.

       [FIRST LINE OF VECTORIZED LOOPS   (NUMBER OF LOOPS VECTORIZED)]
       LINE  (COUNT)        LINE  (COUNT)        LINE  (COUNT)        LINE  (COUNT)        LINE  (COUNT)        LINE  (COUNT)
       -------------        -------------        -------------        -------------        -------------        -------------
       00004 (    2)        00005 (    1)        00006 (    1)        00007 (    1)        00008 (    1)        00013 (    1)

[C]  1 LOOP  WAS  STACKLIBBED.

       [FIRST LINE OF STACKLIBBED LOOPS   (NUMBER OF LOOPS STACKLIBBED)]
       LINE  (COUNT)
       -------------
       00010 (    1)

[D]  4 LOOPS WERE LEFT SCALAR.

       [FIRST LINE OF THE LOOP / LINE THAT PREVENTS VECTORIZATION / THE REASON THAT PREVENTS VECTORIZATION]
       FIRST  AT LINE  REASON THAT THE LOOP WAS NOT VECTORIZED.
       -----  -------  ----------------------------------------
       00005   00005   A PROPERTY OF AN EMBEDDED LOOP PREVENTS THIS LOOP FROM VECTORIZING.
       00012   00014   AN INNER LOOP MIGHT NOT COMPLETELY SPAN THE CORRESPONDING DIMENSION OF A DESTINATION ARRAY (B).
       00015   00017   THIS LOOP CONTAINS A NONVECTORIZABLE LOOP.
       00016   00017   THIS LOOP CONTAINS A NONVECTORIZABLE KIND OF STATEMENT.
```

Figure G-20. Vectorizer Report Example

Each stacklibbed loop is listed by indicating the first source line number (the number in the left margin) of the loop. For explicit DO loops this is the DO statement. For array assignments, it is the assignment. This is followed by a count of loops that are stacklibbed beginning at that line. For explicit DO loops, the count is always 1. For an array assignment, the count is the number of dimensions stacklibbed.

The line marked (D) indicates all loops which can not be translated. These loops remain in their original scalar form. If no loops are left scalar, the message NO LOOPS WERE LEFT SCALAR is printed and the report ends.

Each nonvectorizable loop is listed by indicating the first source line number (the number in the left margin) of the loop. For explicit DO loops this is the DO statement. For array assignments, it is the assignment. This is followed by a line containing the nonvectorizable construct, and a message indicting why the loop was not vectorizable. In some cases, a symbol in parentheses is appended if it is a property of that symbol or a construct containing that symbol that prevented vectorization.

Because a loop is abandoned as soon as a single problem is found, a loop may have additional constructs which prevent vectorization.

INDEX

COMMENT SHEET

MANUAL TITLE: FORTRAN 200 Version 1 Reference Manual

PUBLICATION NO.: 60480200

REVISION: J

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 on the back (please include page number references).

_____ Please reply _____ No reply necessary

FOLD FOLD

CUT ALONG LINE

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 8241 MINNEAPOLIS, MN.

POSTAGE WILL BE PAID BY ADDRESSEE

CONTROL DATA

Technology and Publications Division
Mail Stop: SVL104
P.O. Box 3492
Sunnyvale, California 94088-3492

FOLD FOLD

NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.
FOLD ON DOTTED LINES AND TAPE

NAME:

COMPANY:

STREET ADDRESS:

CITY/STATE/ZIP:

TAPE TAPE

FORTRAN 200 Q7 Q8 QUICK INDEX

CORPORATE HEADQUARTERS, P.O. BOX O, MINNEAPOLIS, MINN. 55440
SALES OFFICES AND SERVICE CENTERS IN MAJOR CITIES THROUGHOUT THE WORLD

LITHO IN U.S.A.

CONTROL DATA