FORTRAN for NOS/VE
Language Definition

CONTROL DATA

FORTRAN Command Parameters

BINARY_OBJECT (B)

default	B=$LOCAL.LGO
B=file	Write object code to specified file.
B=$NULL	Do not write object code to a file.

COMPILATION_DIRECTIVES (CD)

default	CD=ON
CD=ON	Recognize C$ directives.
CD=OFF	Treat C$ directives as comments.

DEBUG (D)

default	D=NONE
D=NT	Do not produce line number and symbol tables for use by Debug.
D=NONE	Produce line number and symbol tables for use by Debug.

DEFAULT_COLLATION (DC)

default	DC=FIXED
DC=F	Use fixed (ASCII) collating sequence.
DC=U	Use user-specified (DISPLAY) collating sequence.

ERROR (E)

default	Write error messages to $ERRORS file.
E=file	Write error messages to specified file.

ERROR_LEVEL (EL)

default	EL=W
EL=T	List trivial errors only.
EL=W	List warning, fatal, and catastrophic errors.
EL=F	List fatal and catastrophic errors.
EL=C	List catastrophic errors only.

EXPRESSION_EVALUATION (EE)

default	EE=NONE
EE=NONE	Perform normal optimizations (no options selected).
EE=C	Evaluate expressions according to precedence rules.
EE=ME	Do not perform optimizations which eliminate instructions that might cause execution errors.
EE=MP	Do not perform optimizations that change floating-point operations to a mathematically equivalent form.
EE=R	Call intrinsic functions by reference rather than by value.

You can select multiple options for the EE parameter.

FORCED_SAVE (FS)

default	FS=OFF
FS=ON	Save variable and array values after RETURN or END.
FS=OFF	Do not save variable and array values after RETURN or END.

INPUT (I)

default	I=$INPUT
I=file	Input source code is on specified file.

LIST (L)

default	L=$LIST (for batch jobs); L=$NULL (for interactive jobs)
L=file	Write compiler output listing to specified file.

(Continued on inside back cover)

FORTRAN
for NOS/VE
Language Definition

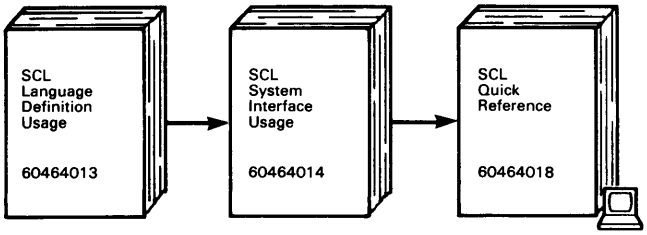
Usage

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

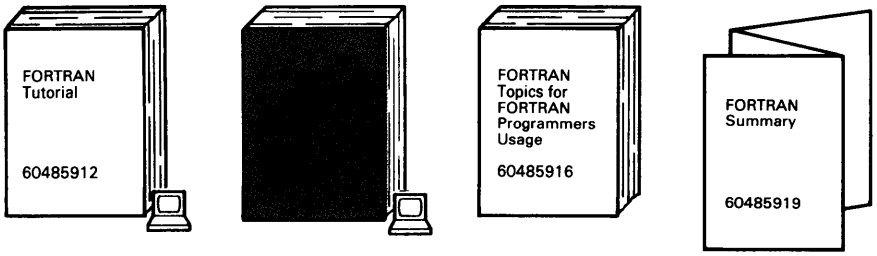
Publication Number 60485913

Related Manuals

Background (Access as Needed):



FORTRAN Manual Set:



Additional References:

→ indicates the recommended reading sequence

means available online

Manual History

Revision B documents the FORTRAN language for NOS/VE Version 1 at release level 1.1.1, PSR level 613. This revision reflects the following new language features: extended Hollerith constants, extended constant expressions, use of A, O, R, and Z descriptors with double precision list items, and use of SCL-style attribute names in file interface calls. This revision also documents the Debug utility. It was printed in June, 1984.

Previous Revision	System Version/ Product Level	Date
A	1.0.2	October, 1983

©1983, 1984 by Control Data Corporation. All rights reserved. Printed in the United States of America.

Contents

About This Manual 5

Introduction to NOS/VE FORTRAN 1-1
- Standard Features 1-1
- CDC Extensions 1-1
- The FORTRAN Compiler 1-1
- The NOS/VE Environment 1-2

Language Elements 2-1
- Writing FORTRAN Statements 2-1
- Symbolic Names 2-4
- Constants 2-5
- Variables 2-12
- Arrays 2-14
- Character Substrings 2-18
- Statement Order 2-20

Specification Statements 3-1
- Type Statements 3-2
- IMPLICIT Statement 3-7
- DIMENSION Statement 3-8
- PARAMETER Statement 3-9
- COMMON Statement 3-10
- EQUIVALENCE Statement 3-12
- SAVE Statement 3-14
- EXTERNAL Statement 3-15
- INTRINSIC Statement 3-16
- DATA Statement 3-18

Expressions and Assignment Statements 4-1
- Expressions 4-1
- Assignment Statements 4-15

Flow Control Statements 5-1
- GO TO Statements 5-1
- IF Statements 5-3
- DO Statement 5-8
- CONTINUE Statement 5-13
- PAUSE Statement 5-13
- STOP Statement 5-14
- END Statement 5-14
- RETURN Statement 5-14
- CALL Statement 5-14

Input/Output 6-1
- Introduction to Input/Output 6-2
- Formatted Input/Output 6-9
- Unformatted Input/Output 6-36
- List Directed Input/Output 6-37
- Namelist Input/Output (CDC Extension) 6-40
- Buffer Input/Output (CDC Extension) 6-44
- Mass Storage Input/Output (CDC Extension) 6-46
- Direct Access Files 6-53
- Internal Input/Output 6-55
- File Interface Input/Output (CDC Extension) 6-59
- Input/Output-Related Statements and Routines 6-110

Program Units 7-1
Main Programs 7-2
Subprograms 7-4
Statement Functions 7-7
Procedure Communication 7-7
Calling Other-Language Subprograms (CDC FORTRAN Only) 7-17

Intrinsic Functions 8-1
Generic and Specific Names 8-1
Function Descriptions 8-6

FORTRAN-Callable Subprograms 9-1
FORTRAN-Sort/Merge Subprograms 9-1
System Command Language Subprograms 9-10
Utility Subprograms 9-21

Compilation and Execution 10-1
FORTRAN Command 10-1
Compiler Output Listing 10-12
Execution Command 10-21

Debug Utility 11-1
Introduction to Debug 11-1
Accessing Debug 11-2
Debug Concepts 11-4
Debug Commands 11-9
Sample Debug Session 11-43

Examples 12-1
Program PASCAL 12-1
Program CORR 12-2
Program COMPSAL 12-5
Subroutine COUNTC 12-7
Program SCLCALL 12-8

Glossary A-1

ASCII Character Set and Collating Weight Tables B-1

Differences Between NOS/VE FORTRAN and NOS FORTRAN 5 ..... C-1

C$ Directives D-1

I/O Implementation E-1

Language Summary F-1

CDC Extensions to Standard FORTRAN G-1

Selecting Collation Tables for Indexed Sequential Files H-1

Index Index-1

About This Manual

This manual describes the CONTROL DATA® FORTRAN Version 1 language. FORTRAN Version 1 complies with the American National Standards Institute FORTRAN language described in document X3.9-1978 and known as FORTRAN 77. The FORTRAN Version 1 compiler is available under the NOS/VE Version 1 operating system.

This manual is intended to be used as a reference. It is not intended to teach the inexperienced programmer how to write FORTRAN programs.

Audience

You should be familiar with an existing FORTRAN language. In addition, you should know how to create and run jobs under the NOS/VE operating system.

Organization

The FORTRAN manual set consists of the following manuals:

FORTRAN Tutorial

This manual is intended for the programmer who has no previous FORTRAN experience. It presents a tutorial introduction to the FORTRAN language, beginning with the basic elements of the language and proceeding through more complex features.

Topics for FORTRAN Programmers

This manual is intended for experienced FORTRAN programmers who are new to NOS/VE. It presents introductory topics intended to help FORTRAN programmers use the NOS/VE operating system and NOS/VE FORTRAN effectively. Topics covered include System Command Language, debugging, input/output, optimization, virtual memory, and object libraries.

Summary

This manual presents a concise pocket-size summary of the FORTRAN language. It presents a complete list of FORTRAN statements in alphabetical order, and shows the parameters for each statement. It does not include detailed parameter descriptions.

Language Definition Manual

This manual presents detailed descriptions and definitions of all the statements and features of the NOS/VE FORTRAN language.

The FORTRAN Language Definition manual is organized into the following chapters:

Chapter 1 presents a brief introduction to NOS/VE FORTRAN and the NOS/VE programming environment.

Chapters 2 through 7 describe the FORTRAN language specifications. Chapter 2 describes the basic elements used in writing FORTRAN statements. Chapters 3 through 6 describe all of the FORTRAN statements. Chapter 7 presents the rules for organizing FORTRAN statements into executable programs. The descriptions include all language features provided by CDC's implementation of FORTRAN, as well as those specified by the ANSI standard.

Chapters 8 and 9 describe CDC-supplied function and subroutine calls that allow you to take advantage of certain operating system capabilities, and to access other CDC products directly from a FORTRAN program.

Chapter 10 describes the system commands used to compile and execute a FORTRAN program. Descriptions of all parameters and options are included. The output listings produced by the FORTRAN command are also described.

Chapter 11 describes the interactive Debug utility, a program that enables you to control and debug an executing program.

Chapter 12 presents some examples of executable programs that illustrate various features of FORTRAN.

Following chapter 12 is a set of appendixes that provides the following supplementary information:

Description of ASCII character set

Glossary of terms used in this manual

Description of compiler control directives

Description of internal structure of FORTRAN files

Summary of differences between NOS/VE FORTRAN and NOS FORTRAN 5

Summary of CDC extensions to ANSI FORTRAN

Conventions

Certain notations are used throughout the manual with consistent meaning. The notations are:

UPPERCASE	In language syntax, uppercase indicates a statement keyword or character that is to be written as shown. Although lowercase letters are interpreted the same as uppercase characters when used in FORTRAN keywords and symbols, uppercase is used for consistency. In occasional examples, keywords and symbols are shown in lowercase for illustrative purposes.
lowercase	In language syntax, lowercase indicates a name, number, symbol, or entity that you must supply.
Boldface	In language syntax, boldface type indicates a required keyword, parameter, or symbol.
Italics	In language syntax, optional keywords, parameters, and symbols are shown in italics.
...	In language syntax, a horizontal ellipsis indicates that the preceding optional item can be repeated as necessary.
. . .	In program examples, a vertical ellipsis indicates that other FORTRAN statements or parts of the program have not been shown because they are not relevant to the example.
Blue	In examples of terminal dialog, blue type indicates user input. Black type indicates system or processor output.
Δ	Blank character. This symbol is used wherever there might otherwise be doubt as to how many blanks are intended.
|	In examples of formatted input and output, vertical bars denote the input or output fields. When used to enclose a numeric quantity, vertical bars indicate the magnitude (absolute value) of the quantity.
	A blue flag in the margin indicates the major description of a CDC extension.

All numbers used in this manual are decimal unless otherwise indicated. Other number systems are indicated by a notation after the number. For example, 177 octal, FA34 hex.

CDC Extensions

CDC extensions to standard ANSI FORTRAN are indicated by an appropriate note immediately preceding the description of the nonstandard feature and by a flag in the margin. Only the primary description of the nonstandard feature is noted; other occurrences of the feature are not noted.

Ordering Manuals

Control Data manuals are available through Control Data sales offices or by sending an order to:

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

When ordering a manual, please specify the complete title, publication number, and revision level.

Submitting Comments

The last page of this manual is a comment sheet. Please use it to give us your opinion of the manual's usability, to suggest specific improvements, and to report technical or typographical errors. If the comment sheet has already been used, you can mail your comments to:

Control Data Corporation
Publications and Graphics Division
P. O. Box 3492
Sunnyvale, California 94088-3492

Please indicate whether you would like a written reply.

Introduction To NOS/VE FORTRAN 1

NOS/VE FORTRAN provides the features and capabilities set forth in the ANSI FORTRAN Standard as well as several CDC-unique features.

Standard Features

NOS/VE FORTRAN (hereafter referred to simply as FORTRAN) offers the full complement of standard FORTRAN capabilities. These capabilities include integer, single precision, double precision, and complex arithmetic; block control structures; character string processing; and a wide range of input/output capabilities. Most FORTRAN programs that strictly comply with the ANSI standard can be compiled using the FORTRAN compiler and executed under NOS/VE with no changes, regardless of the computer system for which the program was originally written.

CDC Extensions

In addition to the standard features, FORTRAN provides unique features which greatly enhance the power of the FORTRAN language. Most of these features consist of subprogram calls that allow you to take advantage of other CDC products. Although a program that uses these features cannot be transported directly to another computer system, these extensions allow greater flexibility and choice of options in the writing of FORTRAN programs. To facilitate the migration of programs from one system to another, a compiler option is available that will detect most non-ANSI usages within a program. The major extensions include:

Boolean data type

Allows you to manipulate octal, hexadecimal, and boolean string data items. Boolean constants and variables are described in chapters 2 and 3, and boolean expressions are described in chapter 4.

Namelist input/output

Allows you to perform formatted input/output without specifying a format or an input/output list. You simply specify a group name, and all items in the group are read or written according to a compiler-defined format. Namelist input/output is described in chapter 6.

Mass storage input/output

Allows you to create and access random files. Records on random files are accessed directly by record key. This provides a quicker method of access than conventional sequential input/output. Mass storage input/output is described in chapter 6.

C$ Directives

Control various aspects of compilation, such as whether or not certain source lines are to be compiled or ignored by the compiler. C$ directives are described in appendix D.

A complete list of all the CDC extensions to ANSI FORTRAN is presented in appendix G.

The FORTRAN Compiler

The FORTRAN compiler reads a file containing the FORTRAN source program, translates that program into an object program consisting of machine instructions, and (optionally) writes the object program to a file. The object program can then be loaded into memory and executed by system commands.

A FORTRAN source program consists of text lines formatted according to the rules of FORTRAN syntax. If the compiler detects a syntax error in the source program, it issues a descriptive message describing the nature of the error. The compiler detects errors at different levels of severity. If the errors are severe enough (fatal), the resulting object program cannot be executed; you must correct the errors and recompile. Generally, the diagnostic messages provide enough information to enable you to easily determine the cause of the errors.

The FORTRAN compiler produces object code at two levels of optimization. The lower level results in faster compilation, but produces an object program that executes more slowly. At the higher level of optimization, the compiler manipulates the generated object code to produce an object program that executes much faster. The level of optimization is selected by a parameter on the FORTRAN command.

In addition to the object program, the FORTRAN compiler produces two other output files: an output listing file and an error listing file. These files are optional and are selected by parameters on the FORTRAN command. The error listing file contains error messages that were issued during compilation. The output listing file contains a complete listing of the source program and, optionally, an object listing and a reference map. The reference map provides detailed information about symbolic names and other items used in the FORTRAN program, and is an extremely useful debugging tool.

The FORTRAN compiler provides a number of other options in addition to those described above. The available compiler options, the formats of the input and output files, and the commands for compiling and executing a FORTRAN program, are described in chapter 10.

The NOS/VE Environment

The NOS/VE operating system provides several software facilities you can use to make the process of creating and maintaining FORTRAN programs easier and more efficient. These facilities include:

Source Code Utility (SCU)

Allows you to create and maintain source programs. SCU is especially useful for creating and updating large collections of source programs called source libraries.

Object Code Management utilities (OCM)

Enables you to create and maintain libraries of compiled object programs (called object libraries). Object libraries are especially useful for programs that are to be shared by other programs.

File Management Utility (FMU)

Enables you to convert data files from one format to another. FMU can be used to convert files from previous CDC systems for use on NOS/VE.

Debug Utility

Enables you to debug a program during execution. You can stop the program at selected points or on the occurence of an error, and request formatted displays of variables and arrays. The Debug utility is described in chapter 11 of this manual.

FORTRAN provides subprogram calls that enable you to use the capabilities of NOS/VE to perform various programming tasks. These subprogram calls include:

Sort/Merge Calls

These calls enable you to sort the records of file into a specific order, and to merge the records of two or more files into a single file.

System Command Language Calls

These calls provide a method of communicating with SCL. The parameter interface calls enable you to reference parameters which you specify on the command that begins execution of a FORTRAN program. The variable interface calls enable you to retrieve or alter the values of existing SCL variables, as well as to define new SCL variables.

General Utility Calls

These calls enable you to perform a variety of tasks, such as generating program dumps, generating random number sequences, and obtaining time and date information from the operating system.

Figure 1-1 shows the relationship of a FORTRAN program to the NOS/VE operating system.

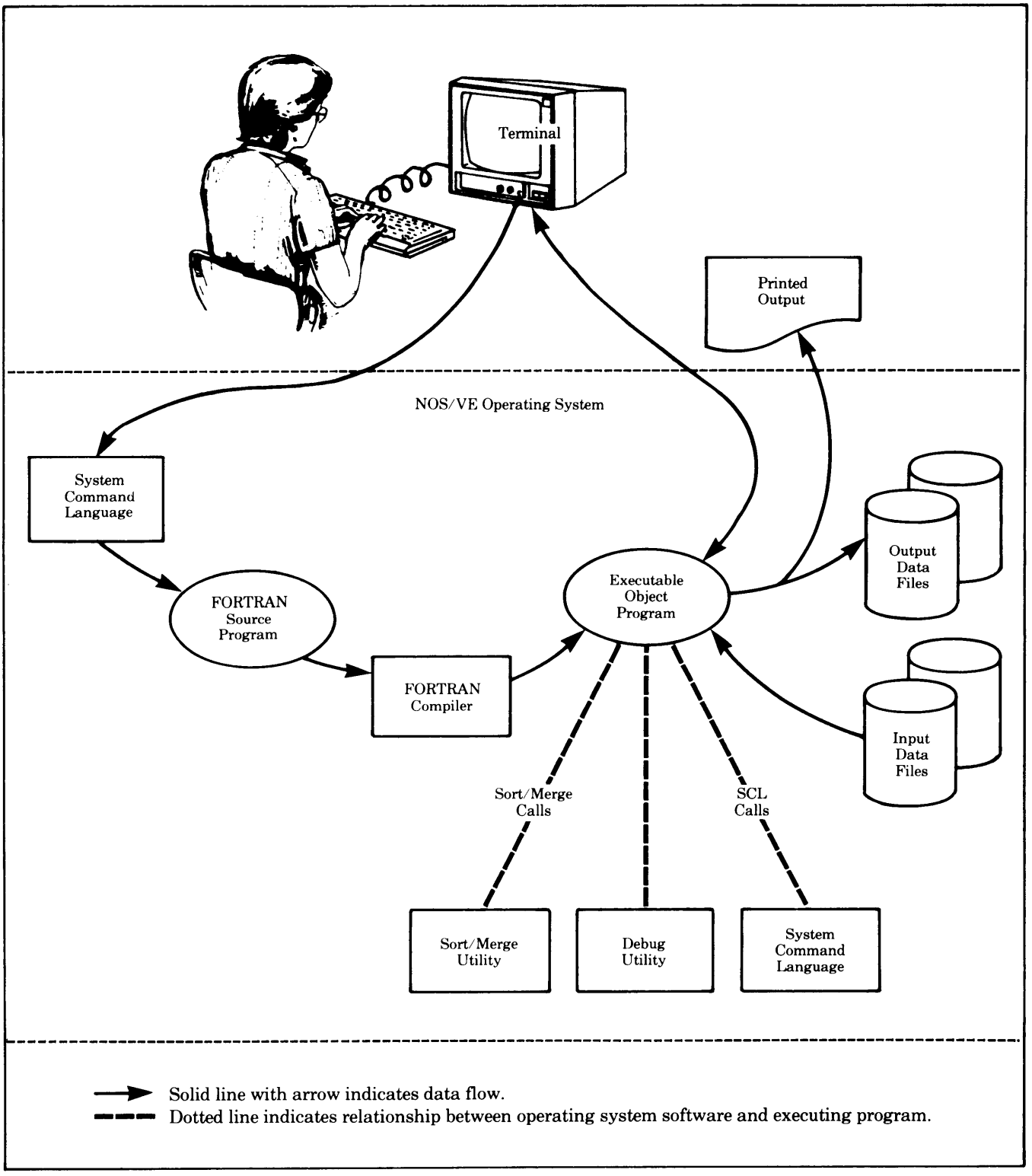


Figure 1-1. The FORTRAN Environment

Language Elements 2

FORTRAN statements are composed of elements that are combined according to the rules of FORTRAN syntax. These elements include constants, variables, substrings, arrays, and operators.

Writing FORTRAN Statements

FORTRAN statements are written using the FORTRAN character set shown in the following table:

Type	Characters
Alphabetic	A through Z (Lowercase letters are equivalent to uppercase letters when used in symbolic names and FORTRAN reserved words.)
Numeric	0 through 9
Special Characters	= equal + plus - minus * asterisk / slash (left parenthesis) right parenthesis , comma . decimal point $ currency symbol ' apostrophe : colon " quote blank

The ASCII representations of the characters are shown in appendix B.

Lowercase letters are equivalent to uppercase letters when used in symbolic names and FORTRAN keywords. For example, the following FORTRAN statements are equivalent:

```
READ (UNIT=1,FMT=99) AVAL, Z

read (unit=1,fmt=99) aval, z
```

However, in character strings, boolean string constants, and extended Hollerith constants, uppercase and lowercase characters are treated as distinct values. For example, the following character constants are not equivalent:

```
'ABCDE'

'abcde'
```

NOTE

For consistency and readability, the FORTRAN syntax descriptions in this manual use uppercase letters to indicate keywords and lowercase letters to indicate user-supplied values. However, in all FORTRAN statements, lowercase letters are valid and are treated the same as uppercase letters.

ASCII characters that are not included in the FORTRAN character set can be used in character, boolean string, and extended Hollerith constants; in apostrophe, H, and quote edit descriptors in format specifications; and in comment lines.

FORTRAN statements can be written in either nonsequenced (normal) mode or sequenced mode. Each program must be written entirely in one mode. Normal mode is principally used for batch jobs. Sequenced mode is a CDC extension and is intended for use with interactive applications. The SEQUENCED_LINES parameter of the FORTRAN command (described in chapter 10) selects sequenced mode.

A FORTRAN statement is written on one or more lines. The first (and possibly only) line of each statement is an initial line. Each additional line is a continuation line. Each statement has one initial line and may have zero through 19 continuation lines.

Lines can also be comment or compiler directive lines.

Nonsequenced Lines

A nonsequenced mode line consists of characters in positions 1 through 80. However, only positions 1 throough 72 are scanned by the compiler. You can use positions 73 through 80 for an identification field. The following example shows a program written in nonsequenced mode.

```
      PROGRAM PASCAL
C
C  THIS PROGRAM PRODUCES A PASCAL TRIANGLE
C
      INTEGER LROW(15)
      DO 10 I = 1,15
         LROW(I) = 1
10    CONTINUE
      PRINT '(17H1 PASCAL TRIANGLE, //1X,I5,/1X,2I5)',
     +LROW(15), LROW(14), LROW(15)
      DO 50 J = 14,2,-1
         DO 40 K = J,14
            LROW(K) = LROW(K) + LROW(K+1)
40       CONTINUE
         PRINT '(1X,15I5)', (LROW(M), M = J-1,15)
50    CONTINUE
      END
```

Initial Lines

Each statement begins with an initial line. The statement characters are written in positions 7 through 72 of the initial line. You can use blanks to improve readability. The initial line of a statement can contain a statement label in positions 1 through 5. Position 6 is blank.

Continuation Lines

If a statement is longer than 66 characters (positions 7 through 72 of the initial line), it can be continued on as many as 19 continuation lines. A character other than blank or zero in position 6 indicates a continuation line. Positions 1 through 5 must be blank.

The length of a nonsequenced statement cannot exceed 1320 characters (one initial line and 19 continuation lines, at 66 characters per line).

Statement Labels

A statement label (any one- through five-digit positive nonzero integer) can be written in positions 1 through 5 of the initial line of a statement. A statement label uniquely identifies a statement so that it can be referenced by other statements. Statements that will not be referenced do not need labels. Blanks and leading zeros are not significant. Labels need not occur in numerical order, but a given label must not be declared more than once in the same program unit. A label is known only in the program unit containing it and cannot be referenced from a different program unit. Any statement can be labeled, but only FORMAT and executable statement labels can be referenced by other statements.

Comment Lines

One of the characters C or asterisk (*) in position 1 indicates a comment line. Comments provide a method of placing program documentation in the source program.

Comments do not affect the program and can be placed anywhere within the program. Comments can appear between an initial line and a continuation line, or between two continuation lines. However, a comment line cannot follow the END statement of a program; a comment line following an END statement is treated as the first line of the next program unit.

Any line with blanks in positions 1 through 72 is also a comment line. Comment lines following an END statement are listed at the beginning of the next program unit.

Additional characters that are not in the FORTRAN character set can be included in comment lines. Comment lines can include any (printable graphic) characters listed in appendix B for the character set being used.

Compiler Directive Lines (CDC Extension)

The characters C and $ in positions 1 and 2 indicate a compiler directive line. A compiler directive must appear on a single line and any compiler directive terminates statement continuation.

Compiler directives are effective unless the COMPILATION_DIRECTIVES parameter of the FORTRAN command is used to suppress interpretation of compiler directives. If directive suppression is specified, compiler directives are interpreted as comment lines.

Each directive, including keyword and parameters, is written in positions 7 through 72. Compiler directives are described in appendix D.

Positions 73 Through 80

Positions 73 through 80 are not considered part of the statement or the line and can be used for identification information. Characters in the identification field are ignored by the compiler but are copied to the source program listing. If source program input comes from other than cards, positions 73 through 90 can be used for identification information.

Sequenced Lines

You can write a FORTRAN program with sequenced lines. Each line represents a source line and usually begins with a sequence number of one through five digts. The sequence numbers for source lines are usually in ascending order, although this is not a requirement. Source lines are interpreted as sequenced lines if you specify the SEQUENCED_LINES parameter on the FORTRAN command.

Example of a sequenced program:

```
00100      PROGRAM PASCAL
00110C
00120C  THIS PROGRAM PRODUCES A PASCAL TRIANGLE
00130C
00140      INTEGER LROW(15)
00150      DO 10 I = 1,15
00160         LROW(I) = 1
00170 10   CONTINUE
00180      PRINT '(17H1 PASCAL TRIANGLE, //1X,I5,/1X,2I5)',
00190      +LROW(15), LROW(14), LROW(15)
00200      DO 50 J = 14,2,-1
00210         DO 40 K = J,14
00220            LROW(K) = LROW(K) + LROW(K+1)
00230 40      CONTINUE
00250         PRINT '(1X,15I5)', (LROW(M), M = J-1,15)
00260 50   CONTINUE
00270      END
```

Like nonsequenced lines, sequenced lines can be initial lines, continuation lines, comment lines, and compiler directive lines.

A line consists of characters in positions 1 through 80. The sequence number of a sequenced line must appear to the left of all other nonblank characters in the line. The sequence number consists of one through five digits, usually at the beginning of a line. Blanks can precede the sequence number.

The statement can begin immediately after one or more blanks following the sequence number. You can insert blanks within the statement to improve readability.

Initial Lines

Every sequenced statement begins with an initial line. The initial line has at least one blank after the sequence number. The initial line can contain a statement label.

Statement Labels

Statement labels can have the same form as for nonsequenced lines. If you include a statement label, it must follow the sequence number and must be separated from the number by one or more blanks. Leading blanks and leading zeros within a label are disregarded. A label must not contain embedded blanks.

Continuation Lines

If a sequenced statement extends beyond position 80 of an initial line it can be continued on as many as 19 continuation lines. A continuation line in sequenced mode is indicated by the character + immediately following the sequence number. Blanks are optional between the + and the continuation of the statement.

Comment Lines

A comment line in sequenced mode is indicated by any character except blank or + immediately following the sequnce number. Any line without a sequence number is treated as a comment line. Note that in sequenced mode, comment lines can begin with characters other than C or asterisk (*).

Compiler Directive Lines

The characters C$ immediately following the sequence number indicate a compiler directive line. You can include one or more blanks between C$ and the beginning of the directive. The directive cannot be continued on subsequent lines.

Symbolic Names

A symbolic name is assigned by the user and consists of one through seven letters and digits (ANSI only allows six), beginning with a letter. (Lowercase letters are equivalent to uppercase letters.) Symbolic names are used for the following:

Main program name	Character substring name
Common block name	Array name
Subroutine name	Symbolic constant name
External function name	Intrinsic function name
Block data subprogram name	Dummy subprogram name
Variable name	Namelist group name

You can use names that are FORTRAN keywords as user-assigned symbolic names without conflict. For example:

```
PROGRAM TEST
PRINT = 1.0
PRINT*, PRINT
.
.
.
```

The name PRINT is legally used as both a variable name and a FORTRAN keyword.

However, certain naming conflicts are illegal and are diagnosed. For example, the sequence

```
PROGRAM ALPHA
ALPHA = 1.0
.
.
.
```

illegally uses the name ALPHA as a program unit name and a variable name.

The following example illegally uses the name RAY as both an array name and a subroutine name:

```
PROGRAM X
DIMENSION RAY(3)
.
.
.
CALL RAY
```

In general, you should avoid naming conflicts by assigning unique names to all program entities.

Constants

A constant is a quantity that remains fixed throughout program execution. The types of constants are integer, real, double precision, complex, boolean, logical, extended Hollerith, and character. You can reference a constant by its actual value or, with the exception of extended Hollerith constants, by a symbolic name associated with the constant. You can use the PARAMETER statement described in chapter 3 to assign a symbolic name to a constant. Integer, real, double precision, complex, and boolean constants are considered arithmetic constants.

Blanks in a constant, except a character, boolean string, or extended Hollerith constant, have no effect on the value of the constant.

Integer

An integer constant is a string of 1 through 19 decimal digits written without a decimal point. An integer constant has the form:

± **d...d**

d Decimal digit

An integer can be positive, negative, or zero. If the integer is positive, the plus sign can be omitted; if it is negative, the minus sign must be present. An integer constant must not contain a comma. The range of an integer constant is -(2**63)-1 through (2**63)-1 ((2**63)-1 is 9,223,372,036,854,775,807).

Examples of valid integer constants:

```
237
-74
+136772
-0024
```

Examples of invalid integer constants:

`46.`	`Decimal point not allowed`
`23A`	`Letter not allowed`
`7,200`	`Comma not allowed`

When an integer constant is used as a subscript, the maximum value is 2**29 and the minimum value is -(2**29) (2**29 is 536,870,912).

When values are converted (in an expression assignment statement) from real to integer or from integer to real, the valid range is also from -(2**63)+1 through (2**63)-1. Values outside this range cause a fatal runtime error when used as operands.

Real

A real constant consists of a string of decimal digits written with a decimal point or an exponent, or both. A real constant has one of these forms:

± coeff

± coeff E ± exp

± n E ± exp

coeff Coefficient in the form:

```
n.
n.n
.n
```

n Unsigned integer constant

exp Unsigned integer exponent (base 10)

A plus sign preceding the coefficient is optional if the constant is positive; a minus sign is required to denote a negative constant.

Permissible values for a real constant are -10**(-1234) through -10**(1232), 0, and 10**(-1234) through 10**(1232). Precision is approximately 14 decimal digits; the constant is stored internally in one computer word.

Examples of valid real constants:

```
7.5
-3.22
+4000.
.5
```

Examples of invalid real constants:

`33,500.`	Comma not allowed
`2.5A`	Letter not allowed

Optionally, you can write a real constant with a decimal exponent. An exponent is written as the letter E (or e) followed by an integer constant indicating the power of ten by which the number is to be multiplied. If the E is present, the integer constant following the letter E must not be omitted. The plus sign can be omitted if the exponent is positive, but the minus sign must be present if the exponent is negative.

Examples of valid real constants with exponents:

`42E1`	Value is 42. x 10**1 = 420
`.000285E+5`	Value is .000285 x 10**5 = 28.5
`6.205E6`	Value is 6.205 x 10**6 = 6 205 000
`700.e-2`	Value is 700. x 10**(-2) = 7 (Symbol e is equivalent to E.)

Example of invalid real constant with exponent:

`7.2E-3.4`	Exponent is not an integer

Double Precision

A double precision constant is written in the same way as a real constant with exponent, except that the exponent is prefixed by the letter D (or d) instead of E. A double precision constant has the form:

± coeff D ± exp

± n D ± exp

coeff Coefficient in the form:

n.
n.n
.n

n Unsigned integer constant

exp Unsigned integer exponent (base 10)

Double precision values are represented internally by two consecutive words, giving additional precision. A double precision constant is accurate to approximately 29 decimal digits. A plus sign preceding the coefficient is optional for a positive constant; the minus sign is required for a negative constant. The plus sign preceding the exponent can be omitted if the exponent is positive, but the minus sign must be present to denote a negative exponent.

Examples of valid double precision constants:

`5.834D2`	Value is 5.834 x 10**2 = 583.4
`14.D-5`	Value is 14. x 10**(-5) = .00014
`9.2d03`	Value is 9.2 x 10**3 = 9200 (The symbol d is equivalent to D.)
`3120D4`	Value is 3120. x 10**4 = 31 200 000

Example of invalid double precision constants:

`7.2D`	Exponent is missing
`D5`	Exponent alone is not allowed
`2,001.3D2`	Comma is not allowed
`3.14159265`	D and exponent are missing

Complex

Complex constants are written as a pair of real or integer constants or symbolic constants separated by a comma and enclosed in parentheses, as follows:

(real,imag)

real Real or integer constant or symbolic constant that represents the real part

imag Real or integer constant or symbolic constant that represents the imaginary part

NOTE

The term real as applied to the first component of a complex value should not be confused with the FORTRAN real data type.

The first constant represents the real part of the complex number and the second constant represents the imaginary part. The parentheses are a required part of the constant. Either constant can be preceded by a plus or minus sign. Complex values are represented internally by two consecutive computer words containing real (floating-point) values.

Type real constants that form the complex constant must be within the valid range for real constants.

Examples of valid complex constants (i = square root of -1):

`(1, 7.54)` Value is 1. + 7.54i

`(-2.1E1, 3.24)` Value is -21. + 3.24i

`(4, 5)` Value is 4.0 + 5.0i

`(0., -1.)` Value is 0.0 - 1.0i

Examples of invalid complex constants:

`(12.7D-4 16.1)` Comma is missing and double precision is not allowed

`4.7E+2, 1.942` Parentheses are missing

Logical

A logical constant has one of the following values:

.TRUE. Represents the logical value true

.FALSE. Represents the logical value false

The periods are a required part of the constant.

Examples of valid logical constants:

`.TRUE.  .true.  .True.`

`.FALSE.  .false.  .False.`

Examples of invalid logical constants:

`.TRUE` No terminating period

`.F.` Abbreviation not recognized

The logical values true and false are represented internally as hex values 000000000000000 and 010000000000000 respectively.

Boolean (CDC Extension)

A boolean constant is a boolean string constant, octal constant, or hexadecimal constant. A boolean constant is represented in one computer word.

Boolean String

A boolean string constant has one of the following forms:

nHs

L"s"

R"s"

"s"

n	An unsigned nonzero integer constant in the range 1 through 8
s	A string of 1 through 8 characters

A boolean string constant can contain no more than eight characters. A constant of the form nHs, L"s", R"s", or "s" that is used as an actual argument and exceeds eight characters is an extended Hollerith constant. (The extra characters are retained.) For all other uses of a boolean string constant, extra characters are truncated on the right, and the compiler issues a trivial diagnostic. (nHs, L"s", R"s", and "s" forms in format specifications are not limited to eight characters.)

The nHs and forms "s" indicate left-justified with blank fill. The value n in nHs specifies the number of characters in the string s. Blank fill means that any unassigned character positions in the computer word are set to blank (ASCII code 20 hex).

Example:

`2HAB`	Value is 414220...20 hex

The L"s" form indicates left-justified with binary zero fill. Binary zero fill means that any unassigned character positions are set to binary zero (ASCII code 00 hex).

Example:

`L"AB"`	Value is 414200...00 hex

The R"s" form indicates right-justified with binary zero fill.

Example:

`R"AB"`	Value is 00...004142 hex

The "s" form is equivalent to the nH form except that the characters need not be counted. Blanks are significant and characters that are not in the FORTRAN character set can be used.

In the L"s", R"s", and "s" forms, a quote within the string is represented by two consecutive quote characters; the consecutive quotes count as one character. In all forms, blanks are significant and characters that are not in the FORTRAN character set can be used.

Examples:

`"AB"`	Value is 414220...20 hex
`"C""D"`	Value is 43234420...20 hex

Extended Hollerith

Extended Hollerith constants are used only as actual arguments to external procedures. An extended Hollerith constant has one of the following forms:

nHs

L"s"

R"s"

"s"

n	An unsigned nonzero integer constant greater than 8.
s	A string of n characters for the nHs form, or a string of greater than 8 characters for any of the other forms.

An extended Hollerith constant is stored in two or more consecutive computer words. The length in words of an extended Hollerith constant is given by the expression

INT((l + 8 - 1)/8)

where l is the number of characters in the constant. (INT means that the fractional part of the result of the division is truncated.)

An extended Hollerith constant is stored beginning in the leftmost character position of the first word. If there are any unassigned character positions in the last word occupied by the constant, those positions are filled with either blanks or zeros, depending on the form of the constant. The characters in the last word occupied by the constant are either left-justified or right-justified within the word, depending on the form of the constant.

For the nHs form, n specifies the number of characters in the string s. Blanks are significant and characters not in the FORTRAN character set can be used.

For the nHs and "s" forms, characters in the last word occupied by the constant are left-justified with blank fill. Blank fill means that any unassigned character positions in the last word occupied by the constant are set to blank characters (ASCII code 20 hex).

Example:

`10HABCDEFGHIJ` Value is 4142434445464748|494A20...0020

For the L"s", R"s", and "s" forms, a " character within the string is represented by two consecutive " characters; the consecutive quotes count as one character.

For the L"s" form, characters in the last word occupied by the constant are left-justified with binary zero fill. Binary zero fill means that any unassigned character positions in the last word occupied by the constant are set to binary zero (ASCII code 00).

Example:

`L"ABCDEFGHIJ"` Value is 4142434445464748|494A00...00

For the R"s" form, characters in the last word occupied by the constant are right justified with binary zero fill.

Example:

`R"ABCDEFGHIJ"` Value is 4142434445464748|00...00494A

The "s" form is equivalent to the nH form except that you need not count characters. Blanks are significant and characters not in the FORTRAN character set can be used.

Examples:

`"ABCDEFGHIJ"` Value is 4142434445464748|494A20...20

`"QRSTU""VWXYZ"` Value is 515253545523565758595A20...20 (represents the string QRSTU"VWXYZ)

Octal

An octal constant has the form:

O"o"

o	A string of 1 through 22 octal digits. (If all 22 digits are used, the leftmost digit must be 0 or 1.)

An octal digit is one of the digits: 0, 1, 2, 3, 4, 5, 6, or 7. The string of octal digits is interpreted as an octal number. As many as 22 octal digits can be represented in a 64-bit computer word. The octal number is right-justified with binary zero fill.

Example:

`0"7"`	Value is 00...003F hex

Hexadecimal

A hexadecimal constant has the form:

Z"z"

z	A string of 1 through 16 hexadecimal digits

A hexadecimal digit is one of the characters: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, or F. (Lowercase letters are permissible.) The string of hexadecimal (hex) digits is interpreted as a base 16 number. As many as 16 hexadecimal digits can be represented in a 64-bit computer word. The hexadecimal number is right-justified with binary zero fill.

Example:

`Z"1A"`	Value is 00...001A hex

Character

A character constant has the form:

's'

s	A string of characters

Apostrophes are used to enclose the character string. Within the character string, an apostrophe is represented by two consecutive apostrophes. The two consecutive apostrophes count as one character in the length of the string.

The minimum number of characters in a character constant is one; the maximum number of characters in a character constant is (2**16)-1 or 65535. The length is the number of characters in the string. Blanks are significant in a character constant. The string can contain any character in the ASCII set. An uppercase character is not equivalent to its lowercase counterpart.

Character positions in a character constant are numbered consecutively as 1, 2, 3, and so forth, up to the length of the constant. The length of the character constant is significant in all operations in which the constant is used. The length must be greater than zero.

Examples of valid character constants:

`'ABC'`	
`'123'`	
`'Year''s'`	Represents the string Year's

Examples of invalid character constants:

`'ABC`	Terminating apostrophe is missing
`"ABC"`	Boolean constant not a valid character constant
`'Year's'`	Apostrophes within string must be doubled

Variables

A variable represents a value that can be changed repeatedly during program execution. Variables are identified by a symbolic name of one through seven letters or digits (ANSI allows only six), beginning with a letter. A variable generally represents a location in memory (although compiler optimizations may preclude the assignment of certain variables to memory locations). A variable must be defined before being referenced for its value. The types of variables are integer, real, double precision, complex, boolean, logical, and character. Variables are typed by default according to the first letter of the variable name. A variable is of type integer if the first letter is I, J, K, L, M, or N and is of type real if the first letter is any other letter. Implicit and explicit typing of variables is described in chapter 3, Specification Statements.

Integer Variables

An integer variable is a variable that is typed explicitly, implicitly, or by default as integer. An integer variable occupies one storage word. The range restrictions for integer variables are the same as for integer constants. See chapter 3 for restrictions on integer variables and constants used in DO statements.

Examples:

```
ITEM1  jsum  J

Nsum   N72   K2S04
```

Real Variables

A real variable is a variable that is typed explicitly, implicitly, or by default as real. The valid values are

-10**1232 through -10**(-1234)

0

10**(-1234) through 10**1232

with approximately 14 significant digits of precision. A real variable occupies one storage word.

Examples:

```
AVAR  RESULT  BETA

Sum3  total2  XXXX
```

Double Precision Variables

A double precision variable is a variable that is typed explicitly or implicitly as double precision. The valid values for a double precision variable are

-10**1232 through -10**(-1234)

0

10**(-1234) through 10**1232

with approximately 29 significant digits of precision. Double precision variables occupy two consecutive storage words. The first word contains the more significant part of the number and the second part contains the less significant part.

Example:

```
DOUBLE PRECISION OMEGA, X, IOTA
```

The variables OMEGA, X, and IOTA are declared double precision.

Complex Variables

A complex variable is a variable that is typed explicitly or implicitly as complex. A complex variable occupies two storage words; each word contains a real number. The first word contains the real part of the number and the second word contains the imaginary part.

Example:

```
COMPLEX ZETA, MU, LAMBDA
```

The variables ZETA, MU, and LAMBDA are declared complex.

Logical Variables

A logical variable is a variable that is typed explicitly or implicitly as logical. A logical variable occupies one storage word. Logical variables contain one of the symbols T or F, representing the logical values true and false.

Example:

```
LOGICAL L33, PRAVDA, VALUE
```

The variables L33, PRAVDA, and VALUE are declared logical.

Boolean Variables (CDC Extension)

A boolean variable is a variable that is typed explicitly or implicitly as boolean. A boolean variable occupies one storage word. Boolean string, octal, or hexadecimal values are generally assigned to boolean variables.

Example:

```
BOOLEAN HVAL, zzz, r34
```

The variables HVAL, zzz, and r34 are declared boolean.

Character Variables

A character variable is a variable that is typed explicitly or implicitly as character. You can specify the length of a character variable when the variable is typed as character. Refer to the description of the CHARACTER statement in chapter 3 for more information on specifying the length of character variables.

Example:

```
CHARACTER NAM*15, C3*3
```

The variable NAM is 15 characters long and the variable C3 is three characters long.

Arrays

A FORTRAN array is a set of elements identified by a single name. The name is composed of one through seven letters and digits and begins with a letter. Each array element is referenced by the array name and a subscript. The type of the array elements is determined by the array name in the same manner as the type of a variable is determined by the variable name. The array name can be typed explicitly with a type statement, implicitly with an IMPLICIT statement, or by default typing. The array name and its dimensions must be declared in a DIMENSION, COMMON, or type statement.

When an array is declared, the declaration of array dimensions has the following form:

array (*d,...,d*)

array Array name.

d Specifies the bounds of an array dimension in the form

lower:**upper**

where lower and upper are as follows:

lower Optionally specifies the lower bound of the dimension. The lower bound can be an integer or boolean expression with a positive, zero, or negative value. If lower: is omitted, the lower bound defaults to 1.

upper Specifies the upper bound of the dimension. The upper bound can be an integer or boolean expression with a positive, zero, or negative value. The upper bound must be greater than or equal to the lower bound. In the case of an assumed-size array, the upper bound of the last dimension can be specified as *.

Arrays can have one through seven dimensions.

The dimension bounds can be positive, zero, or negative. If the lower bound is omitted, it defaults to 1. In this case, the upper bound must be positive. The general rule is that the upper bound must always be greater than or equal to the lower bound. The size of each dimension is indicated by the distance between the lower bound and upper bound.

A dimension bound expression must not include nonintrinsic function references or array element references. If a boolean expression is used for the lower or upper bound of a dimension, the value of the expresion is converted to integer; that is, the value is INT(expression). The expression must not contain boolean variables or boolean intrinsic function references.

Arguments to intrinsic function references may contain other intrinsic function references, constants, or symbolic constants of any type acceptable to the intrinsic function. Arguments to intrinsic function references may also contain integer variables.

The presence of a variable in a dimension bound expression makes the size of the array adjustable. The presence of an asterisk as the upper bound of of the last dimension specifies that the array is an assumed-size array. An assumed-size array can be used only in a subroutine or function, as described under Procedure Communication in chapter 7.

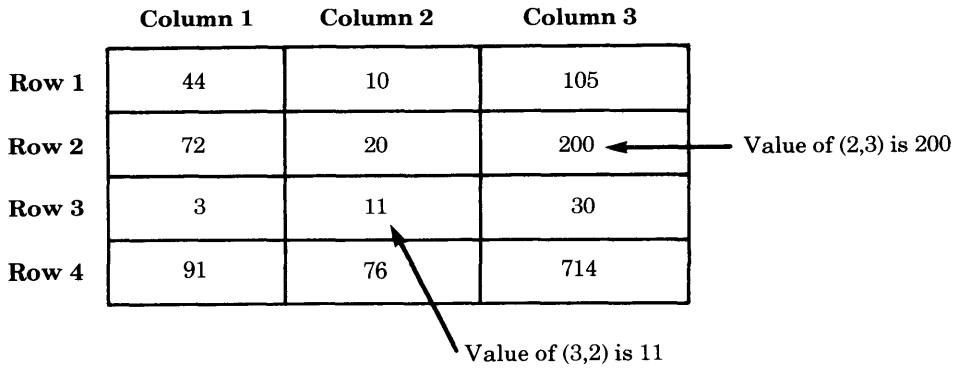
The following examples show storage patterns for a one-dimensional, two-dimensional, and three-dimensional array respectively. Arithmetic values are shown for the array elements, but an array can be any data type. Array elements are stored in ascending locations by positions. The first subscript value increases most rapidly; the last subscript value increases least rapidly.

Example of a one-dimesional array:

```
DIMENSION RX(0:5)
```

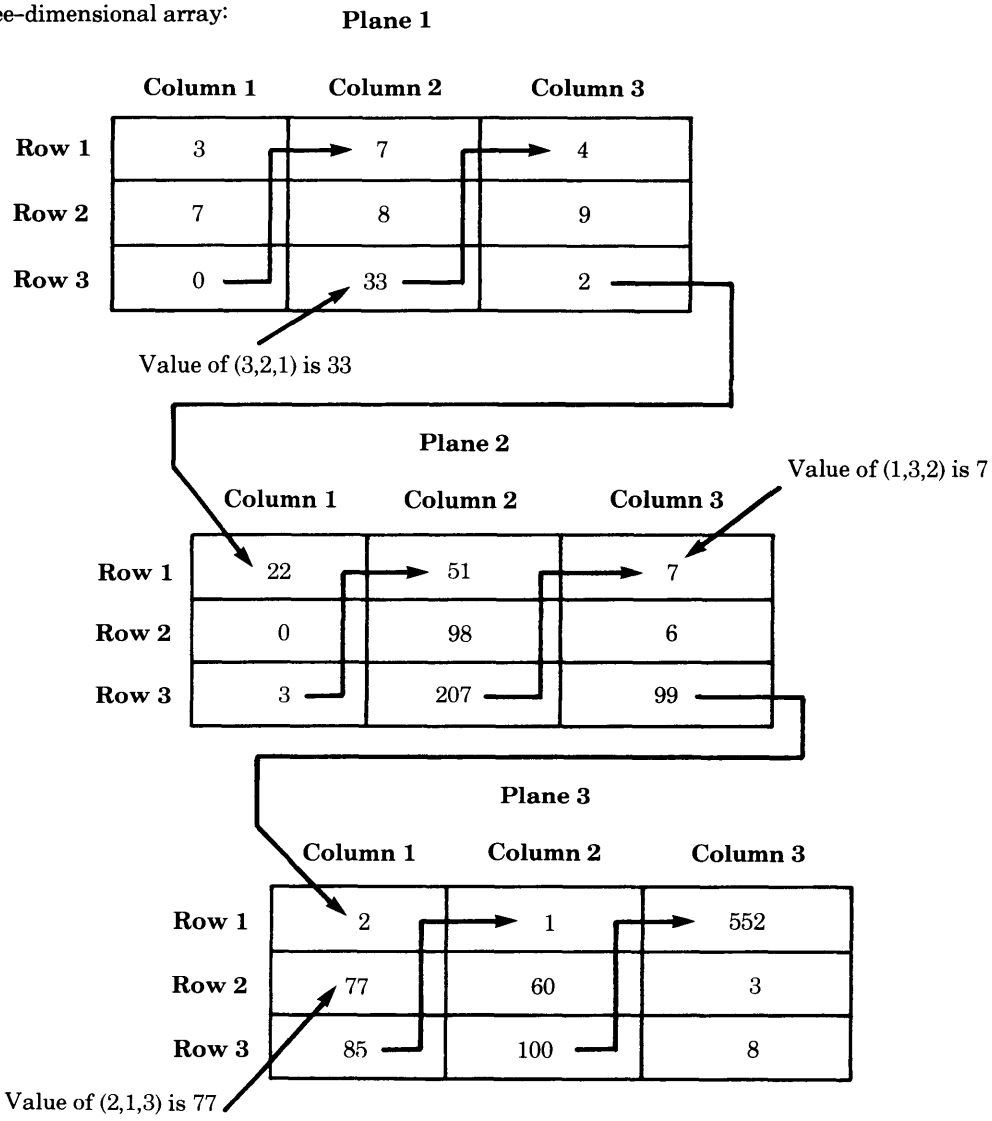
element 0 10.0
element 1 55.0
element 2 11.2
element 3 72.6
element 4 91.9
element 5 7.1

Example of a two-dimensional array:



The array has four rows and three columns, for a total of 12 elements.

Example of a three-dimensional array:



The array has six rows, six columns, and three planes, for a total of 108 elements.

Array Storage

The elements of an array have a specific storage order, with elements of any array stored as a linear sequence of storage words. The first element of the array begins with the first storage position and the last element ends with the last storage word or character storage position.

The span (number of storage words) of an array dimension is given by (u - l + 1) where u is the upper dimension bound and l is the lower dimension bound. The number of words reserved for an array is determined by the type of the array and the spans of its dimensions. An array of type integer, boolean, real, or logical occupies n words of storage, where n is the product of the spans of all dimensions. An array of type complex or double precision occupies 2*n words. An array of type character occupies n*len bytes, where len is the length in characters of an array element.

A noncharacter array must not exceed (2**28)-1 words in length. ((2**28)-1 = 268,435,455.) A character array must not exceed (2**31)-1 bytes. ((2**31)-1 = 2,147,483,647.)

Array References

Array references can be references to complete arrays or to specific array elements. A reference to a complete array is simply the array name. A reference to a specific element involves the array name followed by a subscript specification. An array element reference is also called a subscripted array name.

A reference to the complete array references all elements of the array in the order in which they are stored. For example,

```
DIMENSION XT(3)
DATA XT/1.,2.,3./
PRINT*, XT
```

references the entire array XT in the DATA statement and the PRINT statement.

A reference to an array element references a specific element and has the following form:

array(e,...,*e*)

array	Array name.
e	Subscript expression that is an integer, real, double precision, complex, or boolean expression. (ANSI allows only an integer expression.)

When referencing an array element, you must specify a subscript value for each dimension in the array. Array element references are not legal unless a value is supplied for each dimension. There can be up to seven dimensions in an array element.

An array element reference specifies the name of the array followed by a list of subscript expressions enclosed in parentheses. Each subscript expression can be an integer, real, double precision, complex, or boolean expression. Each subscript expression is evaluated and converted as necessary to integer. A subscript expression can contain function references and array element references; however, evaluation of a function reference must not alter the value of any other subscript expression in the array element reference. (The compiler does not diagnose this condition, however.) The function evaluation also must not alter the value of any other entity in the same subscript expression. For example, in the statement

```
ARY(2,IFUNC(J)+K)
```

IFUNC must not alter the value of K.

Each value after conversion to integer must not be less than the lower bound or greater than the upper bound of the dimension. If the array is an assumed-size array, where the upper bound of the last dimension is specified as an asterisk, the value of the subscript expression must not exceed the actual size of the dimension. For example, in the statements

```
DIMENSION AR(100)
.
.
.
CALL SUB (AR)

SUBROUTINE SUB (BB)
DIMENSION BB(*)
.
.
.
BB(I + J - K)
```

the value of I + J - K must not exceed 100.

If a subscript expression value in an array element reference falls outside the dimension bounds, the results are undefined. (A runtime check for range violations is provided. The check is optional and can be suppressed by the RUNTIME CHECKS parameter on the FORTRAN command.) For each array element reference, evaluation of the subscript expressions yields a value for each dimension and a position relative to the beginning of the array.

The position of an array element is calculated as shown in the following table:

Dimensions	Position of Array Element
1	1 + (s1- j1)
2	1 + (s1 -j1) + (s2 - J2) * n1
3	1 + (s1 - j1) + (s2 - J2) * n1 + (s3 - J3) * n2 * n1
.	
.	
.	
7	1 + (s1 - j1) + (s2 - J2) * n1 + (s3 - J3) * n2 * n1 + (s4 - J4) * n3 * n2 * n1 + (s5 - J5) * n4 * n3 * n2 * n1 + (s6 - J6) * n5 * n4 * n3 * n2 * n1 + (s7 - J7) * n6 * n5 * n4 * n3 * n2 * n1

ji Lower bound of dimension i.

ki Upper bound of dimension i.

ni Size of dimension i. If the lower bound is one, ni + ki. Otherwise ni + (ki - ji + 1).

si Value of the subscript expression specified for dimension i.

The position indicates the relative position of an array element.

Example:

```
INTEGER DZ(12)
.
.
.
DZ(6) = 79
```

The array element reference DZ(6) refers to the element at position 6 in the array, that is, (1 + (6 - 1)).

Example:

```
COMMON /CHAR/ CQ
CHARACTER*5 CQ(6,4)
.
.
.
CQ(6,3) = 'RUN'
```

The array element reference CQ(6,3) refers to the element at position 18, that is, (1 + (6 - 1) + (3 - 1) * 6). The character storage position is 86, that is, 1 + (element position - 1) * character length.

Character Substrings

When a character variable or character array is declared, you can subsequently define and reference the entire string or specific parts of the string. You reference a part of a character string by using a substring reference. Character variables and arrays are declared with the CHARACTER statement described in chapter 3.

Substring References

If the name of a character entity is used in a reference, the value is the current value of the entire entity.

Example:

```
CHARACTER*6 S1, S2
DATA S1/'PEARLS'/
S2 = S1
```

S1 is a reference to the full string 'PEARLS'.

A reference to part of the string is written as a character substring reference. A character substring reference has the form:

char *(first:last)*

char	Character variable, character array name, or character array element reference.
first	Optional integer, real, double precision, complex, or boolean expression specifying the position of the first character of the substring. If omitted, default value is 1. ANSI allows an integer expression only.
last	Optional integer, real, double precision, complex, or boolean expression specifying the position of the last character in the substring. If omitted, value defaults to the length of the string. ANSI allows an integer expression only.

The expression specifying the first character position in the substring is evaluated and converted as necessary to integer. The expression can contain array element references and function references, but evaluation of a function reference must not alter the value of other entities in the substring reference. If the specification of first is omitted, the value is one; all characters from one to the value of the specification of last are included in the substring.

The specification of the last character position in the substring is an expression subject to the same rules as the specification of first. If last is omitted, the value is the length of the string; all characters from the specified first position to the end of the string are included in the substring. For a string length len, the values of first and last are restricted as follows:

$1 \leq first \leq last \leq len$

For example, sustring references to the string S1 with the value 'PEARLS' could be any of the following:

`S1(1:3)`	Value 'PEA'
`S1(3:4)`	Value 'AR'
`S1(4:)`	Value 'RLS'
`S1(:4)`	Value 'PEAR'
`S1(:)`	Value 'PEARLS'

Note that the substring reference S1(:) has the same effect as the reference S1 because all characters in the string are referenced.

If a substring expression exceeds the bounds declared in the CHARACTER statement the results are undefined. The RUNTIME CHECKS parameter on the FORTRAN command provides a runtime check on substring bounds violations.

Substrings and Arrays

If a substring reference is used to select a substring from an array element of a character array, the combined reference includes specification of the array element followed by specification of the substring. For example:

```
CHARACTER*8 ZS(5)
CHARACTER*4 RSEN
.
.
.
ZS(4)(5:6) = 'FG'
RSEN = ZS(1)(:4)
```

The first reference refers to characters 5 and 6 in element 4 of array ZS. The second reference refers to the first 4 characters of the first element of array ZS.

Statement Order

The order of various statements within the program unit is shown in table 2-1.

Table 2-1. Statement Order

Statement				
PROGRAM, SUBROUTINE, FUNCTION, or BLOCK DATA				Comments and compiler directives
IMPLICIT	PARAMETER (must precede first reference)	FORMAT†	ENTRY†† (except within range of block IF or DO loop)	
REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, CHARACTER, BOOLEAN (Type specification statements)				
DIMENSION, EQUIVALENCE, COMMON, SAVE, EXTERNAL, INTRINSIC (Specification statements)				
Statement function definition†	NAMELIST† (must precede first reference)			
Assignment, DO, CONTINUE, IF, ELSE, ELSEIF, ENDIF, GOTO, ASSIGN, CALL, RETURN, PAUSE, STOP (Executable statements)†				
DATA				
OPEN, CLOSE, INQUIRE, READ, WRITE, PRINT, PUNCH, BUFFER IN, BUFFER OUT, ENCODE, DECODE, REWIND, BACKSPACE, ENDFILE (Executable I/O statements)†				
END				

†Cannot be used in a BLOCK DATA subprogram.
††Cannot be used in a main program or BLOCK DATA subprogram.

Within each group, you can order statements as necessary, but you must order the groups as shown. Statements that can appear anywhere within more than one group are shown on the right in boxes that extend vertically across more than one group.

A PROGRAM statement can appear only as the first statement in a main program. The first statement of a subroutine, function, or block data subprogram is respectively in a SUBROUTINE statement, FUNCTION statement, or BLOCK DATA statement. The END statement is the last statement of any program unit.

Comments can appear anywhere within a program unit. Note that a comment following the END statement is considered part of the next program unit.

FORMAT statements can appear anywhere in a program unit.

ENTRY statements can appear anywhere in a program unit, subject to two restrictions. An ENTRY statement cannot appear within the range of a DO loop (between the DO statement and the final statement of the DO loops) or within a block IF structure (between the IF statement and the ENDIF statement). The ENTRY statement cannot be used in the main program unit because an alternate entry point would have no meaning.

Specification statements in general precede the executable statements in a program unit. The nonexecutable specification statements describe characteristics of quantities known in the program unit; executable statements describe the actions to be taken.

All specification statements must precede all DATA statements, NAMELIST statements, statement function definitions, and executable statements. Within the specification statements, all IMPLICIT statements must precede all other specification statements except PARAMETER statements. PARAMETER statements can appear anywhere among the specification statements, but each PARAMETER statement must precede any references to symbolic constants defined by that PARAMETER statement.

All statement function definitions must precede all executable statements in a program unit. Statement function definitions must not appear in block data subprograms.

DATA statements can appear anywhere among statement function definitions and executable statements.

NAMELIST statements can appear anywhere among statement function definitions and executable statements. Note that each NAMELIST statement defining a namelist group must appear before the first reference to that namelist group. Also note that NAMELIST statements must not appear in block data subprograms.

Executable statements must follow all specification statements and any statement function definitions. Executable statements such as assignment, flow control, or I/O statements can appear in whatever order required in the program unit. Executable statements cannot appear in block data subprograms.

The END statement must be the last statement of each program unit.

Specification Statements 3

Specification statements are nonexecutable statements that specify the characteristics of symbolic names used in a FORTRAN program. Specification statements must appear before all DATA statements, NAMELIST statements, statement function statements, and executable statements in the program unit.

DATA statements are not specification statements but are described in this chapter.

The specification statements are as follows:

IMPLICIT

DIMENSION

PARAMETER

EQUIVALENCE

COMMON

SAVE

EXTERNAL

INTRINSIC

Type (INTEGER, REAL, DOUBLE PRECISION, COMPLEX, BOOLEAN (CDC Extension), LOGICAL, CHARACTER)

The IMPLICIT and type statements specify the data type of symbolic names. Default typing of names takes place unless the type statements are used to change the data type of specific names. The IMPLICIT statement changes the default typing of names. Any IMPLICIT statements must precede all other specification statements, except PARAMETER statements.

The DIMENSION statement specifies the number of dimensions in an array and the bounds for each dimension.

The PARAMETER statement gives a symbolic name to a constant. PARAMETER statements can appear anywhere among the specification statements, but each symbolic constant must be defined in a PARAMETER statement before the first reference to the symbolic constant.

The EQUIVALENCE and COMMON statements provide for the sharing of storage. EQUIVALENCE allows variables within a program unit to share storage locations; COMMON defines blocks of storage to be shared by multiple program units.

The SAVE statement preserves the values of variables after execution of a RETURN or END statement in a subprogram. Variables that would otherwise become undefined remain defined and can be used in any subsequent executions of the same subprogram.

The EXTERNAL and INTRINSIC statements control the recognition of function names. The EXTERNAL statement specifies that a function name refers to a user-written function rather than an intrinsic function or a variable (in an actual argument list). The INTRINSIC statement specifies that a function name refers to an intrinsic function (supplied by the compiler or a runtime library) rather than a user-written function.

If any specification statement appears after the first executable statement, DATA statement, NAMELIST statement, or statement function statement, the compiler issues a fatal diagnostic.

DATA statements give initial values to variables. DATA statements must appear after all specification statements in the program unit. DATA statements can appear anywhere among the statement function definitions and executable statements. Usually, DATA statements are placed after the specification statements but before the statement function definitions and executable statements. A variable is considered undefined until a value is assigned with a DATA statement, input statement, or assignment statement. You should always define a variable before the first reference to the variable. Use of undefined variables in expressions can cause run time errors or unpredictable results.

Type Statements

Each variable, array, symbolic constant, statement function, or external function name has a type. Those entities can be typed as integer, real, double precision, complex, boolean, logical, or character. The name of a main program, subroutine, or block data subprogram has no type property, and cannot be typed.

If you do not specify explicit typing, then default typing occurs. The type of each symbolic name is implied by the first letter of the name. The letter I, J, K, L, M, or N implies type integer, and any other letter implies type real.

Implicit typing is controlled by the IMPLICIT statement. The IMPLICIT statement allows you to alter the default typing of names. The default type of a name is determined by the first letter of the name. The IMPLICIT statement specifies a different typing according to the first letter of each name. One or more IMPLICIT statements can be included in each program unit.

Explicit typing defines the types of individual names. The INTEGER, REAL, DOUBLE PRECISION, COMPLEX, BOOLEAN, LOGICAL, and CHARACTER statements are explicit type statements. An explicit type statement can also be used to supply dimension information for an array.

Intrinsic functions are typed by default and need not appear in any explicit type statement in the program. Explicitly typing a generic intrinsic function name does not remove the generic properties of the name. IMPLICIT statements do not change the type of any intrinsic function. Intrinsic functions are described in chapter 8.

INTEGER Statement

The INTEGER statement declares a variable, array, symbolic constant, function name, or dummy procedure name to be type integer. This statement has the form:

INTEGER name, ..., *name*

name A name that is explicitly typed as integer. Each name has one of the forms:

var

A variable, symbolic constant, function, or function entry name.

array*(d,...,d)*

An array name with optional dimension bounds specification. If no dimension bounds are specified, then the enclosing parentheses are omitted. (See DIMENSION statement.)

Examples:

```
INTEGER ITEM1, NSUM, JSUM

integer a72, h2so4

INTEGER M5(2)
```

REAL Statement

The REAL statement declares a variable, array, symbolic constant, function name, or dummy procedure name to be type real. The REAL statement has the form:

REAL name, ..., *name*

name A name that is explicitly typed as real. Each name has one of the forms:

var

A variable, symbolic constant, function, or function entry name.

array*(d,...,d)*

An array name with optional dimension bounds specification. If no dimension bounds are specified, the enclosing parentheses can be omitted.

Examples:

```
REAL IVAR, NSUM3, RESULT

real total2, BETA, XXXX

REAL TR(10,5)
```

DOUBLE PRECISION Statement

The DOUBLE PRECISION statement declares a variable, array, symbolic constant, function name, or dummy procedure name to be type double precision. The DOUBLE PRECISION statement has the form:

DOUBLE PRECISION name, ..., *name*

name A name that is explicitly typed as double precision. Each name has one of the forms:

var

A variable, symbolic constant, function, or function entry name.

array*(d,...,d)*

An array name with optional dimension bounds specification. If no dimension bounds are specified, the enclosing parentheses can be omitted.

Examples:

```
DOUBLE PRECISION DPROD, DEIGV

double precision rmat(10,10)
```

COMPLEX Statement

The COMPLEX statement declares a variable, array, symbolic constant, function name, or dummy procedure name to be type complex. The COMPLEX statement has the form:

COMPLEX name, ..., *name*

name A name that is explicitly typed as complex. Each name has one of the forms:

var

A variable, symbolic constant, function, or function entry name.

array*(d,...,d)*

An array name with optional dimension bounds specification. If no dimension bounds are specified, the enclosing parentheses can be omitted.

Examples:

```
COMPLEX CPVAR

COMPLEX RES(5,5)
```

BOOLEAN Statement (CDC Extension)

The BOOLEAN statement declares a variable, array, symbolic constant, function name, or dummy procedure name to be type boolean. The BOOLEAN statement has the form:

BOOLEAN name, ..., *name*

name A name that is explicitly typed as boolean. Each name has one of the forms:

var

A variable, symbolic constant, function, or function entry name.

array*(d,...,d)*

An array name with optional dimension bounds specification. If no dimension bounds are specified, the enclosing parentheses can be omitted.

Examples:

```
BOOLEAN ALABEL, QMASK

boolean label(14)
```

LOGICAL Statement

The LOGICAL statement declares a variable, array, symbolic constant, function name, or dummy procedure name to be type logical. The LOGICAL statement has the form:

LOGICAL name, ..., *name*

name A name that is explicitly typed as logical. Each name has one of the forms:

var

A variable, symbolic constant, function, or function entry name.

array*(d,...,d)*

An array name with optional dimension bounds specification. If no dimension bounds are specified, the enclosing parentheses can be omitted.

Logical variables contain one of the symbols T or F representing the values true or false, respectively.

Example:

```
LOGICAL SWITCH, TEST
```

CHARACTER Statement

The CHARACTER statement declares a variable, array, symbolic constant, function name, or dummy subprogram name to be type character. The CHARACTER statement has the form:

CHARACTER*len* **name**, ..., *name*

name A name that is explicitly typed as character. Each name has one of the forms:

var
var*len
array*(d,...,d)*
array*(d,...,d)***len**

where var, array, d, and len are as follows:

var A variable, function, symbolic constant, or function entry name.

array An array name.

d Optional dimension bounds specification. If no dimension bounds are specified, the enclosing parentheses can be omitted. (See DIMENSION statement.)

***len** Specifies the length (decimal number of characters) of the item, and can be an unsigned nonzero integer constant; an extended integer constant expression, enclosed in parentheses, with a positive nonzero value; or an asterisk enclosed in parentheses. The default length is one character.

A length specification immediately following the keyword CHARACTER applies to each entity not having its own length specification. A length specification immediately following an entity is the length specification only for that entity. Note that for an array, the length specified is for each array element. If a length is not specified for an entity, either explicitly or by an IMPLICIT statement, the length is one. The unit of length for CHARACTER is characters.

Example:

```
CHARACTER A*3, B(10)*(12+3*2)
```

The example defines a character variable A that is 3 characters long and a character array B that has 10 elements, each of which is 18 characters long.

You can specify the length of a character dummy argument in a function or subroutine as (*). A character string with length (*) is called an assumed-length character string, and is described in chapter 7.

The length of a type character external function name in a FUNCTION or ENTRY statement can be specified as (*). When a reference to such a function is executed, the function has the length specified in the referencing program unit.

The length specified for a character function, in the program unit that references the function, must be an extended integer constant expression and must agree with the length specified in the function. Note that there is always agreement of length if the length (*) is specified in the function.

If you specify a length of (*) for a symbolic constant of type character, the constant has the length of its corresponding extended character constant expression in a PARAMETER statement. If the length specification is a symbolic constant, it must be enclosed in parentheses.

Example:

```
PARAMETER (N=5)
CHARACTER*(N) AB, CD*(N+3)
```

If you omit the parentheses in the CHARACTER statement, the compiler cannot detect where the length specification ends and the variable name begins (blanks do not function as delimiters), and an error message is issued. For example, the statements

```
PARAMETER (N=5, NA=6, NAB=7)
CHARACTER*N AB
```

could be interpreted in several ways, including:

Length specification N, variable name AB
Length specification NA, variable name B
Length specification NAB, no variable name

Example:

```
CHARACTER*10 ASTR, ABC(5), XR*20
```

The variable ASTR and each element of the array ABC are 10 characters long. The variable XR is 20 characters long.

Example:

```
CHARACTER AR*5, BR*8
.
.
.
CALL ZC(BR)
.
.
.
END
SUBROUTINE ZC(STR)
CHARACTER STR*(*)
```

In the example, the variable STR has length 8 when subroutine ZC is called. If subroutine ZC is called with variable AR passed, the variable STR has length 5. See Procedure Communication in chapter 7.

Character substrings are described in chapter 2.

IMPLICIT Statement

The IMPLICIT statement changes or confirms the default typing of symbolic names according to the first letter of the names. The IMPLICIT statement has the form:

IMPLICIT type(ac,...,*ac*), ..., *type(ac*,...,*ac)*

type One of the keywords INTEGER, REAL, DOUBLE PRECISION, COMPLEX, BOOLEAN, LOGICAL, CHARACTER, OR CHARACTER*len.

ac A single letter, or a range of letters represented by the first and last letter separated by a hyphen, indicating which variables are implicitly typed.

len Specifies the length of type character entities, and can be an unsigned nonzero integer constant, or a positive extended integer constant expression enclosed in parentheses.

The IMPLICIT statement establishes the type of variables, arrays, symbolic constants, statement functions and external functions (but not intrinsic functions) that begin with the specified letter or range of letters. Note that since uppercase and lowercase letters are equivalent, implicitly typing a letter of one case has the same effect for the letter of the other case.

The IMPLICIT statements in a program unit must precede all other specification statements except PARAMETER statements. An IMPLICIT statement in a function or subroutine subprogram affects the type associated with dummy arguments and the function name, as well as other variables in the subprogram. Explicit typing of a variable name or array element in a type statement or FUNCTION statement overrides an IMPLICIT specification.

The specified single letters or ranges of letters specify the entities to be typed. A range of letters has the same effect as writing a list of the single letters within the range. The same letter can appear as a single letter, or be within a range of letters, only once in all IMPLICIT statements in a program unit.

For entities of type character, you can also specify an implicit length. If you do not specify the length, it defaults to one. The length must have a positive nonzero value. The specified length applies to all entities implicitly typed as character.

Example:

```
IMPLICIT CHARACTER*20 (M, X-Z)
```

The default typing is changed for names beginning with the letters M, X, Y, or Z. Names beginning with M are typed as character rather than integer, and names beginning with X, Y, or Z are character rather than real.

Note that any explicit typing with a type statement is effective in overriding both the default typing and any implicit typing.

Example:

```
IMPLICIT LOGICAL (L)
INTEGER L, LX, TT
```

The names L and LX are explicitly typed as integer. All other names beginning with L are implicitly typed as logical. The name TT is explicitly typed as integer and does not take the default type real.

DIMENSION Statement

The DIMENSION statement declares symbolic names as array names and specifies the bounds of each array. The DIMENSION statement has the form:

DIMENSION array(d,...,d), ..., *array(d,...,d)*

array Array name.

d Specifies the bounds of a dimension in one of the forms:

upper
upper:lower

where upper and lower are as follows:

upper Upper bound of the dimension; a dimension bound expression in which all constants, symbolic constants, and variables are type integer or boolean. (See Arrays, chapter 2.)

lower Lower bound of the dimension; a dimension bound expression in which all constants, symbolic constants, and variables are type integer or boolean. If only the upper bound is specified, the lower bound defaults to 1. (See Arrays, chapter 2.)

A DIMENSION statement can declare more than one array. Dummy argument arrays specified within a subprogram can have adjustable dimension specifications. A further explanation of adjustable dimension specifications appears under Procedure Communication in chapter 7.

An array can be declared only once within a program unit. Note that dimension information can be specified in COMMON statements and type statements. The dimension information defines the array dimensions and the bounds for each dimension.

The description of arrays in chapter 2 covers the properties of arrays, the storage of arrays, and array references.

Example:

```
REAL NIL
DIMENSION NIL(6,2,2)
```

NIL is an array containing 24 real elements. The following statement is equivalent:

```
REAL NIL(6,2,2)
```

Example:

```
COMPLEX BETA
DIMENSION BETA(2,3)
```

BETA is an array containing six complex elements.

Example:

```
CHARACTER*8 XR
DIMENSION XR(0:4)
```

XR is an array containing five character elements, and each element has a length of eight characters. A reference to the third and fourth characters of the second element would be XR(1)(3:4).

PARAMETER Statement

The PARAMETER statement gives a symbolic name to a constant. This statement has the form:

PARAMETER (p=e, ..., *p=e*)

p A symbolic name

e An extended constant expression

If the symbolic name of a constant is of type integer, real, double precision, complex, or boolean, the corresponding expression must be an extended arithmetic or boolean constant expression. If the symbolic name is of type character or logical, the corresponding expression must be an extended character constant expression or logical constant expression. (Expressions are described in chapter 4.) Each symbolic name becomes defined with the value of the expression that appears to the right of the equals, according to the rules for assignment. Any symbolic constant that appears in an expression e must have been previously defined in the same or a different PARAMETER statement in the program unit.

A symbolic constant can be defined only once in a program unit, and can identify only the corresponding constant. The type of a symbolic constant can be specified by an IMPLICIT statement or type statement before the first appearance of the symbolic constant in a PARAMETER statement. If the length of a symbolic character constant is not the default length of one, the length must be specified in an IMPLICIT statement or type statement before the first appearance of the symbolic constant. The easiest way to do this is to explicitly type the symbolic constant as character with length (*). The actual length of the constant is then determined by the length of the string defining it in the PARAMETER statement. The length must not be changed by another IMPLICIT statement or by subsequent statements.

Once defined, a symbolic constant can appear in the program unit in the following ways:

In an expression in any subsequent statement

In a DATA statement as an initial value or a repeat count

In a complex constant as the real or imaginary part

In a C$ directive as a primary in an expression, or as a parameter value

A symbolic constant cannot appear in a format specification.

Example:

```
PARAMETER (ITER = 20, START = 5)
CHARACTER CC*(*)
PARAMETER (CC = '(I4, F10.5)')
.
.
.
DATA COUNT/START/
.
.
.
DO 410 J = 1, ITER
.
.
.
READ CC, IX, RX
```

The symbolic constant START is used to assign an initial value to variable COUNT, the symbolic constant ITER is used to control the DO loop, and the symbolic constant CC is used to specify a character constant format specification.

COMMON Statement

The COMMON statement defines areas of storage to be available to the program unit in which the statement appears and associates variables and arrays with the defined area of storage. The areas of storage declared in a COMMON statement are called common blocks. The COMMON statement has the form:

COMMON */name/***nlist**, ..., */name/nlist*

name Name identifying a named common block; name can be omitted, in which case the common block is called blank (unlabeled) common. If the first specification is for blank common, the slashes can also be omitted. The comma separating /name/ from the preceding nlist is optional.

nlist List of entities to be included in the common block. The entities are separated by commas and have one of the forms:

var	Variable name
arr	Array name
arr(d,...,d)	Array name with declared dimensions

COMMON blocks provide a means of associating entities in different program units. The use of common blocks enables different program units to define and reference the same data without using arguments, and to share storage locations. Within a program unit, an entity in a common block is known by a specific name. Within another program unit, the same data can be known by a different symbolic name.

A particular variable name or array name can appear only once in any COMMON statement within the program unit. Function or entry names cannot be included in COMMON statements. In a subprogram, names of dummy arguments cannot be included in COMMON statements.

Within a program unit, declarations of common blocks are cumulative. The nlist following each successive appearance of a particular common block name (or no name for blank common) adds more entities to that common block and is treated as a continuation of the specification. Variables and arrays are stored in the order in which they appear in the common specifications.

If any character variable or character array is included in a common block, all entities in the common block must be type character. You can use a common block name within a program unit as a variable or array name, without conflict.

The maximum number of common blocks in an executable program, including blank common and all named common, is 500. The maximum size of each common block is 536,870,912 storage words (for character data, 4,294,967,296 characters).

The actual size of any common block is the number of storage words required for the entities in the common block, plus any extensions associated with the common block by EQUIVALENCE statements. Extensions can only be made by adding storage words at the end of the common block. (See the description of the EQUIVALENCE statement in this chapter.) A blank common block can be treated as having a different size in separate program units. The length of a common block, other than blank common, must not be increased by a subprogram using the block unless (in CDC FORTRAN only) the subprogram is loaded first. If a program unit does not use all locations reserved in a common block, unused variables can be inserted in the COMMON declaration to ensure proper correspondence of common areas.

Entities in named common blocks can be initially defined by a DATA statement in a block data subprogram, or (in CDC FORTRAN only) by a DATA statement in any program unit. Entities in blank common cannot be initially defined. After an entity in a named common block has been initially defined, the value is available to any subprogram in which the named common block appears. If the entity is reinitialized in a later subprogram, a loader diagnostic is issued.

Entities in blank common remain defined at all times and do not become undefined upon execution of a RETURN or END statement in a subprogram. Entities in named common become undefined upon execution of a RETURN or END in a subprogram, unless the SAVE statement is used or the FORCED SAVE option is selected on the FORTRAN command. The SAVE statement is described later in this chapter; the FORCED SAVE option is described in chapter 10.

Example:

```
COMMON A, B
COMMON /XT/ C, D, E
.
.
.

SUBROUTINE P(Q, R)
COMMON /XT/ F, G, H
.
.
.

FUNCTION T(U)
COMMON Y, Z
```

The entities C, D, and E in the main program are in the common block named XT. The same storage words are known by the names F, G, and H in subroutine P. The entities A and B in the main program are in blank common. The same storage words are known by the names Y and Z in function T.

Example (non-ANSI):

```
COMMON JCOUNT
.
.
.
JCOUNT = 6
.
.
.

FUNCTION AB(A)
COMMON /C/ STX(4)
DATA STX/1., 2., 2., 1./
```

Since an entity in blank common cannot be initially defined with a DATA statement, an assigment statement must be used to define the value of JCOUNT. In function AB, a DATA statement can be used to define initial values for the elements of array STX in the common block named C. Note that JCOUNT is not common to function AB.

Example:

```
CHARACTER*15 D, E
COMMON /CVAL/ D, E
DATA D, E/'TEST', 'PROD'/
```

The common block named CVAL contains character variables. The variables D and E are initially defined in a DATA statement.

Example:

```
COMMON /SUM/ A, B(20)
.
.
.

SUBROUTINE GR
COMPLEX FR(10)
COMMON /SUM/ X, FR
```

The common block SUM in the main program is declared to contain the variable A and the array B. In the subroutine GR, the same storage words are associated with X and the array FR. Even if X is not used in the subroutine, X holds a place so that array FR matches the placement of array B. Note also that array FR is complex. The elements B(1) and B(2) are known in GR as FR(1); B(3) and B(4) are FR(2); and so forth. The specification of common block SUM accounts for 21 storage words.

EQUIVALENCE Statement

The EQUIVALENCE statement specifies the sharing of storage by two or more entities in a program unit. The EQUIVALENCE statement has the form:

EQUIVALENCE (nlist), ..., *(nlist)*

nlist A list of variable names, array names, array element names, or character substring names. The names are separated by commas. Each nlist establishes an equivalence class. Each subscript or substring expression in the list must be an extended integer constant expression.

Equivalencing causes association of the entities that share the storage. Equivalencing associates entities within a program unit, while common blocks associate entities across program units. You cannot equivalence entities in common blocks to one another; however, you can equivalence an entity not explicitly declared to be in common to an entity in common without conflict.

If the equivalenced entities are of different data types, equivalencing does not cause type conversion. If a variable and an array are equivalenced, the variable does not acquire array properties and the array does not lose the properties of an array. You can equivalence an entity of type character only to another entity of type character. The lengths of the equivalenced character entities can be different.

Each nlist specification must contain at least two names of entities to be equivalenced. In a subprogram, names of dummy arguments cannot appear in the list. Function and entry names cannot be included in the list. Equivalencing specifies that all entities in the list share the same first storage word. For character entities, equivalencing specifies that all entities in the list share the same first character storage position. Equivalencing can indirectly cause the association of other entities, for instance when an EQUIVALENCE statement interacts with a COMMON statement.

If an array element is included in nlist, the number of subscript expressions must match the number of dimensions declared for the array name. If an array name appears in the list, the effect is as if the first element of the array had been included in the list. Any subscript expression must be an extended integer constant expression. For character entities, any substring expression must be an extended integer constant expression.

Example:

```
DIMENSION Y(4), B(3,2)
EQUIVALENCE (Y(1), B(3,1))
EQUIVALENCE (X, Y(2))
```

Storage is shared so that six storage words are needed for Y, B, and X. The associations are:

```
        B(1,1)
        B(2,1)
Y(1)    B(3,1)
Y(2)    B(1,2)   X
Y(3)    B(2,2)
Y(4)    B(3,2)
```

Example:

```
CHARACTER A*5, C*3, D(2)*2
EQUIVALENCE (A,D(1)), (C,D(2))
```

Storage is shared so that five character storage positions are needed for A, C, and D. The associations are:

```
A(1:1)  D(1)(1:1)
A(2:2)  D(1)(2:2)
A(3:3)  D(2)(1:1)     C(1:1)
A(4:4)  D(2)(2:2)     C(2:2)
A(5:5)                C(3:3)
```

You can equivalence variables of different data types, except for character data. The equivalencing associates the first storage word of each entity. For example,

```
REAL TR(4)
COMPLEX TS(2)
EQUIVALENCE (TR,TS)
```

causes the following associations:

TR(1) TS(1)-real part
TR(2) TS(1)-imaginary part
TR(3) TS(2)-real part
TR(4) TS(2)-imaginary part

Equivalencing must not reference array elements in a way that conflicts with the storage sequence of the array. You cannot specify the same storage unit as occurring more than once in the storage sequence. For example,

```
REAL FA(3)
EQUIVALENCE (FA(1),B), (FA(3),B)
```

is illegal. Also, the normal storage sequence of array elements cannot be interrupted to make consecutive storage words no longer consecutive. For example,

```
REAL BZ(7), CZ(5)
EQUIVALENCE (BZ,CZ), (BZ(3),CZ(4))
```

is also illegal.

The interaction of COMMON and EQUIVALENCE statements is restricted in two ways:

An EQUIVALENCE statement must not attempt to associate two different common blocks in the same program unit. For example,

```
COMMON /LT/ A, T
COMMON /LX/ S, R
EQUIVALENCE (T,S)
```

is not legal.

An EQUIVALENCE statement must not cause a common block to be extended by adding storage words before the first storage word of the common block. On the other hand, a common block can be extended through equivalencing if storage words are added at the end of the common block. For example,

```
COMMON /X/ A
REAL B(5)
EQUIVALENCE (A, B(4))
```

is not legal, whereas

```
COMMON /X/ A
REAL B(5)
EQUIVALENCE (A, B(1))
```

can be used to extend the common block.

SAVE Statement

The SAVE statement causes the definition status of entities to be retained after the execution of a RETURN or END statement in a subprogram. The SAVE statement has the form:

SAVE *a, ..., a*

a Optional variable name, array name, or common block name enclosed in slashes. The same name must not appear more than once. If no names are specified, all variables, arrays and common data are saved.

A SAVE statement is optional and has no effect in a main program or in any subprogram compiled with OPT=DEBUG or OPT=LOW specified on the FORTRAN command.

You can also use the FORCED_SAVE parameter on the FORTRAN command to save program entities. Selecting the FORCED_SAVE parameter is equivalent to specifying a SAVE statement in every subprogram compiled.

Dummy argument names, procedure names, and names of entities in a common block must not appear in the SAVE statement. A common block name has the effect of specifying all of the entities in the common block. A SAVE statement with no list is treated as though it contained the names of all allowable items in the program unit. If a common block name is specified in a SAVE statement in a subprogram, the common block name must be specified by a SAVE statement in every subprogram in which the common block appears.

Execution of a RETURN statement or an END statement within a subprogram causes the entities within the subprogram to become undefined, except in the following cases:

Entities specified by SAVE statements do not become undefined.

Entities in blank common do not become undefined.

Entities that have been defined in a DATA statement (and not redefined) do not become undefined.

Entities in a named common block that appears in the subprogram and in at least one other program unit that references that subprogram do not become undefined.

Entities in a subprogram compiled with OPT=DEBUG or OPT=LOW specified on the FORTRAN command.

If a local variable or array that is specified in a SAVE statement and is not in a common block is defined in a subprogram at the time a RETURN or END statement is executed, that variable or array remains defined with the same value at the next reference to the subprogram.

Within a subprogram, an entity in a common block can be defined or undefined, depending on the definition status of the associated storage. If you specify a named common block in a SAVE statement in a subprogram and the entities in the common block are defined, the common block storage remains defined at the time a RETURN or END statement is executed and is available to the next program unit that specifies the named common block. The common block storage can become undefined or redefined in another program unit.

The following example illustrates the SAVE statement:

```
PROGRAM MAIN
COMMON /C1/ G, H
CALL XYZ
.
.
.
END
```

```
SUBROUTINE XYZ
COMMON A, D, F
COMMON /C1/ GVAL, HVAL
SAVE
DATA JCOUNT /5/
X = 6.5
.
.
.
RETURN
END
```

The SAVE statement in subroutine XYZ has the effect of saving the value of X as 6.5 for any later invocations of the subroutine. Saving of certain other values does not depend on the presence of the SAVE statement. The three entities in blank common remain defined. The two entities in common block C1 remain defined because common block C1 appears in the referencing program unit. Finally, since JCOUNT is initially defined and not redefined in the subroutine, JCOUNT remains defined for any later invocations of the subroutine.

EXTERNAL Statement

The EXTERNAL statement identifies a name as representing an external function or subroutine and permits such a name to be used as an actual argument. The EXTERNAL statement has the form:

EXTERNAL name, ..., *name*

name Name of an external function or subroutine, dummy function or subroutine, or block data subprogram

A symbolic name can appear only once in all of the EXTERNAL statements of a program unit. If an external subprogram name is an actual argument in a program unit, it must appear in an EXTERNAL statement in the program unit. A statement function name must not appear in an EXTERNAL statement.

If an intrinsic function name appears in an EXTERNAL statement in a program unit, the name becomes the name of some external function or subroutine. The intrinsic function with the same name cannot be referenced in the program unit.

In the following example, the name SQRT is declared as external. The function reference SQRT(X) is therefore taken to reference the user-written function SQRT rather than the intrinsic function SQRT.

```
SUBROUTINE ARGR
EXTERNAL SQRT
.
.
.
Y = SQRT(X)
.
.
.
END

FUNCTION SQRT(XVAL)
.
.
.
END
```

In the following example, the names LOW and HIGH are declared as external. In one call to subroutine AR, LOW is passed as an actual argument and the function reference FUNC(VAL) is equivalent to LOW(VAL). In the second call to subroutine AR, the function reference FUNC(VAL) is equivalent to HIGH(VAL).

```
SUBROUTINE CHECK
EXTERNAL LOW, HIGH
.
.
.
CALL AR(LOW, VAL)
.
.
.
CALL AR(HIGH, VAL)
.
.
.
RETURN
END

SUBROUTINE AR(FUNC, VAL)
VAL = FUNC(VAL)
.
.
.
RETURN
END

REAL FUNCTION LOW(X)
.
.
.
END

REAL FUNCTION HIGH(X)
.
.
.
END
```

INTRINSIC Statement

The INTRINSIC statement identifies a name as representing an intrinsic function, and enables use of an intrinsic function name as an actual argument. The INTRINSIC statement has the form:

INTRINSIC fun, ..., *fun*

fun An intrinsic function name

Appearance of a name in an INTRINSIC statement declares the name as an intrinsic function name. If you use an intrinsic function name as an actual argument in a program unit, it must appear in an INTRINSIC statement in the program unit. You must not use the following intrinsic function names as actual arguments:

Type conversion functions BOOL, CHAR, CMPLX, DBLE, FLOAT, ICHAR, IDINT, IFIX, INT, REAL, and SNGL

Lexical relationship functions LGE, LGT, LLE, and LLT

Largest/smallest value functions AMAX0, AMAX1, AMIN0, AMIN1, DMAX1, DMIN1, MAX, MAX0, MAX1, MIN, MIN0, MIN1

Logical and masking functions AND, OR, XOR, NEQV, EQV, COMPL (these functions are CDC extensions)

The appearance of a generic intrinsic function name in an INTRINSIC statement does not remove the generic properties of the name.

An intrinsic name can appear only once in all INTRINSIC statements in a program unit. Note that a symbolic name must not appear in both an EXTERNAL and an INTRINSIC statement in the program unit.

In the following example, the name SQRT is declared intrinsic in subroutine DC and passed as an argument to subroutine SUBA. Within SUBA, the reference FNC(A) references the intrinsic function SQRT.

```
SUBROUTINE DC
INTRINSIC SQRT
.
.
.
CALL SUBA(X, Y, SQRT)
.
.
.
END

SUBROUTINE SUBA(A, B, FNC)
B = FNC(A)
.
.
.
```

In the following example, the names SIN and COS are declared as intrinsic and can therefore be passed as actual arguments. In the first call to subroutine AR, the reference FUNC(VAL) is equivalent to SIN(VAL); in the second call, FUNC(VAL) is equivalent to COS(VAL). In each case, the intrinsic function is referenced.

```
SUBROUTINE CHECK
INTRINSIC SIN, COS
.
.
.
CALL AR(SIN,VAL)
.
.
.
CALL AR(COS,VAL)
.
.
.
END

SUBROUTINE AR(FUNC,VAL)
VAL = FUNC(VAL)
.
.
.
```

DATA Statement

The DATA statement provides initial values for variables, arrays, array elements, and substrings. The DATA statement has the form:

DATA nlist/clist/, ..., *nlist/clist/*

nlist A list of names to be initially defined. Each name is one of the following:

A variable name
An array name
An array element name
A character substring name
An implied DO list

clist A list of constants or symbolic constants, separated by commas, specifying the initial values. Each item in the list has one of the forms:

c
r*c
r(c,...,c) (CDC Extension)

where

c is a constant or symbolic constant.

r is a repeat count that is an unsigned nonzero integer constant or symbolic constant. The repeat count repeats the constant or list of constants enclosed in parentheses.

The DATA statement is nonexecutable and can appear anywhere after the specification statements in a program unit.

Entities that are initially defined by DATA statements are defined when the program begins execution. Entities that are not initially defined, and not associated with an initially defined entity, are undefined at the beginning of execution of the program.

You must not initialize a variable, array element, or substring more than once in separate program units. (An attempt to do so results in a loader error.) If two entities are associated, only one can be initially defined by a DATA statement.

If an entity is intially defined more than once in a DATA statement, the last definition overrides any previous ones.

Names of dummy arguments, functions, and entities in blank common (including any entities associated with an entity in blank common) cannot be initially defined. Entities in a named common block can be initially defined within a block data subprogram, or (in CDC FORTRAN only) within any program unit in which the named common block appears.

For each list nlist, you must specify the same number of items in the corresponding list clist. A one-to-one correspondence exists between the items specified by nlist and the constants specified by clist. The first item of nlist corresponds to the first constant of clist, the second item to the second constant, and so forth. If an unsubscripted array name appears as an item in nlist, you must specify a constant in clist for each element of the array. The values of the constants are assigned according to the storage order of the array.

Elements of an array which are not explicitly defined in a DATA statement remain undefined. For example:

```
DIMENSION RAY(3)
DATA RAY(2)/0./
```

RAY(1) and RAY(3) are undefined.

Character Data Initialization

For initialization by a DATA statement, a character item in nlist must correspond to a character constant in clist. The character item becomes initially defined according to the following rules:

If the length of the character item in nlist is greater than the length of the corresponding character constant, the additional character positions in the item are initially defined as blanks.

If the length of the character item in nlist is less than the length of the corresponding character constant, the additional characters in the constant are ignored.

Note that initial definition of a character item causes definition of all character positions of the item. Each character constant initially defines exactly one character variable, array element, or substring.

Example:

```
CHARACTER STR1*6, STR2*3
DATA STR1/'ABCDE'/
DATA STR2/'FGHJK'/
```

The character variables STR1 and STR2 are initially defined. Variable STR1 is set to 'ABCDE ', with the sixth character position defined as blank. Variable STR2 is set to 'FGH', with the fourth and fifth characters of the constant ignored.

Expressions and Assignment Statements 4

Expressions are composed of operands and operators that define an operation to be performed. Expressions are evaluated to a specific value. They are used in assignment statements to assign values to variables and array elements.

Expressions

Expressions are formed from a combination of operators, operands, and parentheses. Expressions can be arithmetic, character, relational, logical, or (in CDC FORTRAN only) boolean expressions. Arithmetic, boolean, character, relational, and logical expressions are described separately. The relational expressions are not fully independent and are used as parts of logical expressions.

Expressions are composed of variable or constant operands and operators that define an operation to be performed on the operands. An expression that contains only constant operands is a constant expression. A constant expression in which each operand is a constant or any of the intrinsic functions listed in this chapter with constant arguments is an extended constant expression.

Arithmetic Expressions

An arithmetic expression is used to perform an arithmetic computation. It consists of a sequence of unsigned constants, symbolic constants, variables, array elements, and function references separated by operators and parentheses. An arithmetic expression has the form:

aexp

aexp An arithmetic expression in one of the forms:

term	aexp + bprim
+term	aexp - bprim
-term	bprim + term
aexp + term	bprim - term
aexp - term	bprim + bprim
+ bprim	bprim - bprim
- bprim	

term An arithmetic term in one of the forms:

fact
term * fact
term / fact
term * bprim
term / bprim
bprim * fact
bprim * bprim
bprim / bprim

fact An arithmetic factor in one of the forms:

prim
prim ** fact
bprim ** fact
prim ** bprim
bprim ** bprim

bprim A boolean primary, as described for boolean expressions

prim An arithmetic primary; one of the following entities:

Unsigned arithmetic constant
Arithmetic symbolic constant
Arithmetic variable
Arithmetic array element reference
Arithmetic function reference
Arithmetic expression enclosed in parentheses

An arithmetic entity is one whose data type is integer, real, double precision, or complex.

An arithmetic expression can be an unsigned arithmetic constant, symbolic arithmetic constant, arithmetic variable reference, arithmetic array element reference, or arithmetic function reference. More complicated arithmetic expressions can be formed by using one or more arithmetic or boolean operands together with arithmetic operators and parentheses. Arithmetic operands identify values of type integer, real, double precision, or complex.

The arithmetic operators are:

** Exponentiation
* Multiplication
/ Division
+ Addition or identity
- Subtraction or negation

For two operands A and B, the arithmetic operators have the following meaning:

A ** B	Exponentiate A to the power B. B is called the exponent of A.
A * B	Multiply A and B.
A / B	Divide A by B.
A + B	Add A and B
+ B	Same as B.
A - B	Subtract B from A.
- B	Negate B.

Each of the operators **, /, and * operates on a pair of operands and is written between the two operands. Each of the operators + and - either operates on a pair of operands and is written between the two operands, or operates on a single operand and is written preceding that operand.

Examples of valid arithmetic expressions:

```
3.78542     (A-B)*F + C/D**E     J**I

+AVAL       C*D/E                A*(-B)
```

Examples of invalid arithmetic expressions:

`B*-A`	Adjacent operators not permitted
`(F+(X**Y)`	Right parenthesis missing

An arithmetic constant expression contains only the following operands:

Arithmetic constants

Symbolic arithmetic constants

Arithmetic constant expressions enclosed in parentheses

Boolean constants

Symbolic boolean constants

Boolean constant expressions enclosed in parentheses.

An exponent used with an exponentiation operator must be of type integer or boolean. If the exponent e is of type boolean, the value used is INT(e). Note that variable, array element, and function references are not allowed.

An arithmetic constant expression is an arithmetic expression in which each primary is an arithmetic or boolean constant, symbolic arithmetic or boolean constant, or arithmetic or boolean constant expression enclosed in parentheses. Any exponents that appear are of type integer. Note that arithmetic constant expressions do not contain variable, array element, or function references.

An extended arithmetic constant expression contains only arithmetic or boolean constants, symbolic arithmetic or boolean constants, references to any of the intrinsic functions in the following list with extended constant expressions as actual arguments, or extended arithmetic constant expressions enclosed in parentheses. The allowable intrinsic functions are as follows:

ABS	CCOS	DDIM	ERFC	MIN
ACOS	CEXP	DEXP	EXP	MIN0
AIMAG	CLOG	DIM	FLOAT	MIN1
AINT	CMPLX	DINT	IABS†	MOD
ALOG	COMPL	DLOG	ICHAR	NEQV
ALOG10	CONJG	DLOG10	IDIM	NINT
AMAX0	COS	DMAX1	IDINT	OR
AMAX1	COSD	DMIN1	IDNINT	REAL
AMIN0	COSH	DMOD	IFIX	SHIFT
AMIN1	CSIN	DNINT	INDEX	SIGN
AMOD	CSQRT	DPROD	INT	SIN
AND	DABS	DSIGN	ISIGN	SIND
ANINT	DACOS	DSIN	LEN	SINH
ASIN	DASIN	DSINH	LOG	SNGL
ATAN	DATAN	DSQRT	LOG10	SQRT
ATAN2	DATAN2	DTAN	MASK	TAN
ATANH	DBLE	DTANH	MAX	TAND
BOOL	DCOS	EQV	MAX0	TANH
CABS	DCOSH	ERF	MAX1	XOR

†Valid only if the fixed collation weight table is in effect. (Collation tables are described in chapter 9.)

An integer constant expression is an arithmetic constant expression or a boolean constant expression in which each constant or symbolic constant is of type integer or boolean. If a boolean constant expression e appears, the value used is INT(e). Note that variable, array element, and function references are not allowed.

Examples of integer constant expressions:

```
3     -3+4     R"A"

-3    0"74"    R"AB".AND. 48
```

An extended integer constant expression is an extended arithmetic or boolean constant expression in which each operand is an integer or boolean constant, a symbolic integer or boolean constant, a reference to one of the intrinsic functions in the following list with extended constant expressions as actual arguments, or an extended integer constant expression enclosed in parentheses. The allowable intrinsic functions are as follows:

ABS†	IABS	INDEX	MAX0	NEQV
AND	ICHAR††	INT	MAX1	NINT
BOOL	IDIM	ISIGN	MIN†	OR
COMPL	IDINT	LEN	MIN0	SHIFT
DIM†	IDNINT	MASK	MIN1	SIGN†
EQV	IFIX	MAX†	MOD†	XOR

†Valid only with integer or boolean arguments.
††Valid only if the fixed collation weight table is in effect. Collation weight tables are discussed in chapter 9.

A set of rules establishes the interpretation of an arithmetic expression that contains two or more operators. A precedence among the arithmetic operators determines the order in which the operands are combined:

** Highest
* and / Intermediate
+ and - Lowest

For example, in the expression

```
- A ** 2
```

the exponentiation operator (**) has precedence over the negation operator (-). The operands of the exponentiation operator are combined to form an expression used as the operand of the negation operator. The above expression is the same as the expression:

```
- (A ** 2)
```

You can alter the default order of evaluation of subexpressions within an expression by enclosing the subexpression in parentheses. The subexpressions within the parentheses are always evaluated before being combined with other operands in the expression. For example, in the expression

```
A + B/C
```

B is divided by C and the result is added to A. However, in the expression

```
(A + B) / C
```

A is added to B and the result is divided by C.

Parenthetical subexpressions can be nested within an expression. For example

```
(A + (C/D)) * B
```

is a valid expression.

In an expression containing nested subexpressions, the subexpression within the innermost pair of parentheses is evaluated first. Note that each left parenthesis must have a corresponding right parenthesis.

Successive exponentiations are combined from right to left. For example,

```
2**3**2
```

has the same interpretation as

```
2**(3**2)
```

Two or more multiplication or division operators are combined from left to right.

Two or more addition or subtraction operators are combined from left to right. Note that arithmetic expressions containing two consecutive arithmetic operators, such as A**-B or A+-B, are not permitted. However, expressions such as A**(-B) and A+(-B) are permitted.

Subexpressions containing operators of equal precedence are evaluated from left to right. The compiler may reorder individual operations that are mathematically associative or commutative to perform optimizations such as removal of repeated subexpressions. The mathematical results of the reordering are correct but the specific order of evaluation is indeterminate. For example, the expression A/B*C is guaranteed to algebraically equal (AC)/B, not A/(BC), but the specific order of evaluation by the compiler is indeterminate. (You can inhibit reordering of operators of equal precedence by specifying the EXPRESSION_EVALUATION=CANONICAL parameter on the FORTRAN command.)

The data type of an arithmetic expression containing one or more arithmetic operators is determined from the data types of the operands. Integer expressions, real expressions, double precision expressions, and complex expressions are arithmetic expressions whose values are of type integer, real, double precision, and complex, respectively. When the operator + or - operates on a single operand, the data type of the resulting expression is the same as the data type of the operand unless the operand is of type boolean, in which case the type of the resulting expression is integer.

When an arithmetic operator operates on a pair of arithmetic operands of the same type, the result has the same type as the operands. If the operands are of different types, the expression is known as a mixed mode expression. The data types of the results of mixed mode expressions are given in the following tables:

Resulting Data Type for X1**X2:

Type of x1	Type of x2	x1 Value Used	x2 Value Used	Resulting Data Type
Integer	Integer	x1	x2	Integer
Integer	Real	REAL(x1)	x2	Real
Integer	Double Precision	DBLE(x1)	x2	Double Precision
Integer	Complex	CMPLX (REAL(x1),0.)	x2	Complex
Real	Integer	x1	x2	Real
Real	Real	x1	x2	Real
Real	Double Precision	DBLE(x1)	x2	Double Precision
Real	Complex	CMPLX(x1,0.)	x2	Complex
Double Precision	Integer	x1	x2	Double Precision
Double Precision	Real	x1	DBLE(x2)	Double Precision
Double Precision	Double Precision	x1	x2	Double Precision
Double Precision	Complex	CMPLX(SNGL(x1),0.)	x2	Complex
Complex	Integer	x1	x2	Complex
Complex	Real	x1	CMPLX(x2,0.)	Complex
Complex	Double Precision	x1	CMPLX(SNGL(x2),0.)	Complex
Complex	Complex	x1	x2	Complex

Resulting Data Type for X1+X2, X1-X2, X1*X2, or X1/X2:

Type of x1	Type of x2	x1 Value Used	x2 Value Used	Resulting Data Type
Integer	Integer	x1	x2	Integer
Integer	Real	REAL(x1)	x2	Real
Integer	Double Precision	DBLE(x1)	x2	Double Precision
Integer	Complex	CMPLX(REAL(x1),0.)	x2	Complex
Real	Integer	x1	REAL(x2)	Real
Real	Real	x1	x2	Real
Real	Double Precision	DBLE(x1)	x2	Double Precision
Real	Complex	CMPLX(x1,0.)	x2	Complex
Double Precision	Integer	x1	DBLE(x2)	Double Precision
Double Precision	Real	x1	DBLE(x2)	Double Precision
Double Precision	Double Precision	x1	x2	Double Precision
Double Precision	Complex	CMPLX(SNGL(x1),0.)	x2	Complex
Complex	Integer	x1	CMPLX(REAL(x2),0.)	Complex
Complex	Real	x1	CMPLX(x2,0.)	Complex
Complex	Double Precision	x1	CMPLX(SNGL(x2),0.)	Complex
Complex	Complex	x1	x2	Complex

If two arithmetic operands are of different types, the operand that differs in type from the result of the operation is converted to the type of the result. The operator then operates on a pair of operands of the same type. The exception to this is an operand of type real, double precision, or complex raised to an integer power; the integer operand is not converted. If the value of J is negative, the interpretation of I**J is the same as the interpretation of 1/(I**ABS(J)), which is subject to the rules for integer division. For example, 2**(-3) has the value of 1/(2**3), which is zero.

A boolean operand in an exponentiation operation is converted to integer. For the + - * and / operations, if two operands are of different type and one type is boolean, the result has the type of the other operand. If both operands are of type boolean, the result has type integer. The result of the operator + or the operator - operating on a single boolean operand is of type integer. A boolean operand is converted to the type of the result, and the operation is performed on the converted operand. (Combining boolean operands with operands that are neither boolean nor integer can lead to unexpected results.)

One operand of type integer can be divided by another operand of type integer to yield an integer result. The result is the signed nonfractional part of the mathematical quotient. For example, (-10)/4 yields -2; the result is formed by discarding the fractional part of the mathematical quotient -2.5.

Character Expressions

A character expression consists of a sequence of character operands joined by concatenation operators. A character expression has the form:

cexp

cexp — A character expression in one of the forms:

cprim
cexp // cprim

cprim — A character primary; one of the following:

Character constant
Character symbolic constant
Character variable
Character array element
Character substring reference
Character function reference
A character expression

// — Concatenation operator

The value of a character expression is a character string. Evaluation of a character expression produces a result of type character. The simplest form of a character expression is a single operand of type character. You can form more complicated character expressions by using two or more character operands together with character operators and parentheses.

The result of a concatenation operation is a character string concatenated on the right with another string and whose length is the sum of the lengths of the strings. For example, the value of 'AB' // 'CDE' is the string 'ABCDE'.

The operands of a character expression must identify values of type character. Except in a character assignment statement, a character expression must not involve concatenation of an operand whose length specification is an asterisk in parentheses, unless the operand is a symbolic constant.

In a character expression containing two or more operands, the operands are combined from left to right to interpret the expression. For example, the interpretation of the character expression

```
'AB' // 'CD' // 'EF'
```

is the same as the interpretation of the character expression

```
('AB' // 'CD') // 'EF'
```

The value of the preceding expression is the same as that of the constant 'ABCDEF'. Note that parentheses have no effect on the value of a character expression. Thus, the expression

```
'AB' // ('CD' // 'EF')
```

has the same value as the preceding expressions.

A character constant expression is a character expression in which each operand is a character constant, a symbolic character constant, or a character constant expression enclosed in parentheses. Note that variable, array element, substring, and function references are not allowed.

An extended character constant expression is a character expression in which each operand is a character constant, a reference to the CHAR intrinsic function with an extended integer constant expression as the actual argument, or an extended character constant expression enclosed in parentheses.

Relational Expressions

Relational expressions can appear only within logical expressions. Evaluation of a relational expression produces a logical result with a true or false value. A relational expression has the form:

rexp

rexp A relational expression in one of the forms:

aexp rop aexp
aexp rop bprim
bprim rop aexp
bprim rop bprim
cexp rop cexp

rop One of the following relational operators:

.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

aexp An arithmetic expression

bprim A boolean primary, as described for boolean expressions

cexp A character expression

A relational expression is used to compare the values of two arithmetic or boolean expressions, or of two character expressions. You cannot use a relational expression to compare the value of an arithmetic expression with the value of a character expression. For two operands A and B, the relational operators have the following meaning:

A .LT. B Is A less than B?

A .LE. B Is A less than or equal to B?

A .EQ. B Is A equal to B?

A .NE.B Is A not equal to B?

A .GT. B Is A greater than B?

A .GE. B Is A greater than or equal to B?

You can use a complex operand only when the relational operator is .EQ. or .NE.

An arithmetic relational expression has the logical value true only if the values of the operands satisfy the relation specified by the operator. If the two arithmetic expressions are of different types, or both are boolean, the value of the relational expression

X1 relop X2

is the value of the expression

((X1) - (X2)) relop 0

where 0 (zero) is of the same type as the expression. Note that the comparison of a double precision value and a complex value is not permitted.

Examples of valid relational expressions:

`J .LT. ITEM`	Is J less than ITEM?
`580.2 .gt. var`	Is 580.2 greater than VAR?
`B .GE. (2.7, 59E3)`	Real part of complex operand is used.
`C .LT. 1.5D4`	Most significant part of double precision operand is used.

Example of invalid relational expression:

`A .GT. 720 .LT. 900`	Two relational operands are not permitted.

A character relational expression has the logical value true only if the values of the operands satisfy the relation specified by the operator. The character expression X1 is considered to be less than X2 if the value of X1 precedes the value of X2 in the collating sequence; X1 is greater than X2 if the value of X1 follows the value of X2 in the collating sequence. Note that the collating sequence in use determines the result of the comparison. The default collating sequence is the ASCII collating sequence as shown in appendix D. Also refer to Collation Control in appendix F.

Character relational expressions in PARAMETER and conditional compilation (C$ IF) statements are always evaluated using the ASCII sequence.

If the operands are of unequal length, the shorter operand is treated as if it had been extended on the right with blanks to the length of the longer operand.

Logical Expressions

Evaluation of a logical expression produces a result of type logical, with a value of true or false. A logical expression has the form:

lexp

lexp A logical expression in one of the forms:

```
ldis
lexp .EQV. ldis
lexp .NEQV. ldis
lexp .XOR. ldis
```

ldis A logical disjunction in either of the forms:

```
lterm
ldis .OR. lterm
```

lterm

lprim A logical primary; one of the following:

Logical constant
Logical symbolic constant
Logical variable
Logical array element reference
Logical function reference
Logical expression enclosed in parentheses

The logical operators are shown in the following table:

Operator	Representing	Use of Operator	Meaning
.NOT.	Logical negation	.NOT. A	Complement A
.AND.	Logical conjunction	A .AND. B	Boolean product of A and B
.OR.	Logical inclusive disjunction	A .OR. B	Boolean sum of A and B
.EQV.	Logical equivalence	A .EQV. B	Test for A logically equivalent to B
.NEQV.	Logical nonequivalence	A .NEQV. B	Test for A not logically equivalent to B
.XOR.	Logical exclusive disjunction (CDC extension)	A .XOR. B	Boolean difference of A and B

A logical constant expression contains only logical constants, symbolic logical constants, relational expressions that contain only constant expressions, or logical constant expressions enclosed in parentheses. Note that logical constant expressions do not contain variable, array element, or function references.

An extended logical constant expression contains only logical constants, symbolic logical constants, relational expressions that contain only extended constant expressions, and references to any of the intrinsic functions in the following list with extended character constant expressions as actual arguments, or extended logical constant expressions enclosed in parentheses. The allowable intrinsic functions are as follows:

LGE LGT LLE LLT

The simplest form of a logical expression is a single logical operand with no operators. More complicated logical expressions can be formed by using one or more logical operands together with logical operators and parentheses.

Examples (L, M, and Z are logical variables):

```
.NOT. L

.NOT. (X .GT. Y)

x .gt. y .and. .not. z
```

Examples of invalid relational expressions:

```
.AND. M
```
.AND must be preceded by a logical expression.

```
L .AND. .OR. M
```
.AND. must be separated from .OR. by a logical expression.

A set of rules establishes the interpretation of a logical expression that contains two or more logical operators. A precedence among the logical operators determines the order in which the operands are to be combined, unless the order is changed by the use of parentheses. The precedence of the logical operators is:

.NOT.	Highest
.AND.	
.OR.	
.EQV. or .NEQV. or .XOR.	Lowest

For example, in the expression

```
A .OR. B .AND. C
```

the .AND. operator has higher precedence than the .OR. operator; therefore, the interpretation is the same as

```
A .OR. (B .AND. C)
```

When a logical expression contains two or more .AND. operators; two or more .OR. operators; or two or more .EQV., NEQV., or .XOR. operators, the logical quantities are combined from left to right.

The value of a logical operation involving any logical operator is shown in the following table:

x1	x2	.NOT.x2	x1.AND.x2	x1.OR.x2	x1.EQV.x2	x1.NEQV.x2	x1.XOR.x2
.TRUE.	.TRUE.	.FALSE.	.TRUE.	.TRUE.	.TRUE.	.FALSE.	.FALSE
.TRUE.	.FALSE.	.TRUE.	.FALSE.	.TRUE.	.FALSE.	.TRUE.	.TRUE.
.FALSE.	.TRUE.	.FALSE.	.FALSE.	.TRUE.	.FALSE.	.TRUE.	.TRUE.
.FALSE.	.FALSE.	.TRUE.	.FALSE.	.FALSE.	.TRUE.	.FALSE.	.FALSE.

Boolean Expressions (CDC Extension)

A boolean expression is formed with logical operators and boolean or arithmetic operands. Evaluation of a boolean expression produces a result of type boolean. A boolean expression has the form:

bexp

bexp A boolean expression in one of the forms:

bdis
bexp .EQV. bdis
bexp .EQV. aexp
aexp .EQV. bdis
aexp .EQV. aexp
bexp .NEQV. bdis
bexp .NEQV. aexp
aexp .NEQV. bdis
aexp .NEQV. aexp
bexp .XOR. bdis
bexp .XOR. aexp
aexp .XOR. bdis
aexp .XOR. aexp

bdis A boolean disjunct in one of the forms:

bterm
bdis .OR. bterm
bdis .OR. aexp
aexp .OR. bterm
aexp .OR. aexp

bterm A boolean term in one of the forms:

bfact
bterm .AND. bfact
bterm .AND. aexp
aexp .OR. bterm
aexp .OR. aexp

bfact A boolean factor in one of the forms:

bprim
.NOT. bprim
.NOT. aexp

aexp An arithmetic expression

bprim A boolean primary; one of the following:

Unsigned boolean constant
Boolean symbolic constant
Boolean variable
Boolean array element reference
Boolean function reference

Examples:

```
X .AND. Z"FFFF"

(X + Y) .OR. R

P .AND. Q .OR. S
```

A boolean constant expression is a boolean expression that contains only boolean constants, symbolic boolean constants, boolean constant expressions enclosed in parentheses, arithmetic constants, symbolic arithmetic constants, or arithmetic constant expressions enclosed in parentheses.

An extended boolean constant expression is a boolean constant expression that contains only boolean or arithmetic constants, symbolic boolean or arithmetic constants, references to any of the intrinsic functions listed for arithmetic constant expressions with extended constant expressions as actual arguments, or extended boolean constant expressions enclosed in parentheses.

Boolean quantities are combined from left to right when a boolean expression contains two or more AND. operators, two or more .OR. operators, or two or more .EQV., .NEQV., or .XOR. operators.

If an operand is of type integer, real, double precision, or complex, it is converted to boolean and the operation is performed on the converted operand.

A boolean operator determines each bit value of the result it yields independently of other bits of the result. Each bit value of the result is determined from the corresponding bit values of the operands. At each bit position, the bit in the result is determined as shown in the following table:

		Corresponding Bit in Result of					
Each Bit in x1	**Corresponding Bit in x2**	**.NOT.x2**	**x1.AND.x2**	**x1.OR.x2**	**x1.EQV.x2**	**x1.NEQV.x2**	**x1.XOR.x2**
0	0	1	0	0	1	0	0
0	1	0	0	1	0	1	1
1	0	1	0	1	0	1	1
1	1	0	1	1	1	0	0

General Rules for Expressions

The order in which operands are combined using operators is determined by:

1. Use of parentheses
2. Precedence of the operators
3. Right-to-left interpretation of exponentiations
4. Left-to-right interpretation of multiplications and divisions
5. Left-to-right interpretation of additions and subtractions in an arithmetic expression
6. Left-to-right interpretation of concatenations in a character expression
7. Left-to-right interpretation of .AND. operators
8. Left-to-right interpretation of .OR. and .NOT. operators
9. Left-to-right interpretation of .EQV., NEQV., and .XOR. operators in a logical expression or boolean expression

Precedences have been established among the arithmetic and logical operators. There is only one character operator. No precedence is established among the relational operators. The precedences among the operators are:

Arithmetic	Highest
Character	
Relational	
Logical	Lowest

An expression can contain more than one kind of operator. For example, the logical expression

```
L .OR. A + B .GE. C
```

where A, B, and C are of type real, and L is of type logical, contains an arithmetic operator, a relational operator, and a logical operator. This expression would be interpreted as:

```
L .OR. ((A + B) .GE. C)
```

Any variable, array element, function, or character substring referenced in an expression must be defined at the time the reference is made. An integer operand must be defined with an integer value rather than a statement label value. Note that if a character string or substring is referenced, all of the referenced character positions must be defined at the time the reference is executed.

You must not specify any arithmetic operation whose result is not mathematically defined; for example, dividing by zero, or raising a zero value to a zero-valued or negative-valued power.

A function reference in a statement must not alter the value of any other entity within the statement in which the function reference appears. The execution of a function reference in a statement must not alter the value of any entity in common that affects the value of any other function reference in that statement. However, execution of a function reference in the expression of a logical IF statement can affect entities in the statement that is executed when the value of the expression is true. If a function reference causes definition of an actual argument of the function, that argument or any associated entities must not appear elsewhere in the same statement. For example, the statements

```
A(I) = F(I)
Y = G(X) + X
```

are prohibited if the reference to F defines I, or the reference to G defines X.

All of the operands of an expression are not necessarily evaluated if the value of the expression can be determined otherwise. For example, in the logical expression

```
X .GT. Y .OR. L(Z)
```

where X, Y, and Z are real, and L is a logical function, the function reference L(Z) need not be evaluated if X is greater than Y. If a statement contains a function reference in a part of an expression that need not be evaluated, all entities that would have become defined in the execution of that reference become undefined at the completion of evaluation of the expression containing the function reference. In the example above, evaluation of the expression causes Z to become undefined if L defines its argument.

If a statement contains more than one function reference, the functions can be evaluated in any order, except for a logical IF statement and a function argument list containing function references. For example, the statement

```
Y = F(G(X))
```

where F and G are functions, requires G to be evaluated before F is evaluated.

Any expression contained in parentheses is always treated as an entity. For example, in the expression A*(B*C), the product of B and C is evaluated and then multiplied by A; the mathematically equivalent expression (A*B)*C is not used.

Assignment Statements

An assignment statement consist of a variable followed by an equal sign followed by an expression. (In the case of multiple assignment statements, multiple equal signs and multiple expressions are allowed.) When the assignment statement is executed, the expression is evaluated and the result is stored in the variable. If the variable has been previously defined, the new value replaces the current value. There are five types of assignment statements:

Arithmetic

Logical

Statement label (with the ASSIGN statement as described in chapter 5)

Character

Boolean (CDC Extension)

Arithmetic Assignment Statement

An arithmetic assignment statement has the form:

v = exp

v	A variable or array element of type integer, real, double precision, or complex
exp	An arithmetic or boolean expression

After evaulation of the arithmetic expression exp, the result is converted to the type of v in the following way:

Integer	INT (exp)
Real	REAL (exp)
Double precision	DBLE (exp)
Complex	CMPLX (exp)

The result is then assigned to v, and v is defined or redefined with that value.

Examples:

```
A = A + 1.0

WAGE = PAY - TAX

VAR = VALUE + (7/4) * 32
```

In the following example, a value is assigned to the complex variable C:

```
COMPLEX C
PARAMETER (PAR1=1., PAR2=2.)
.
.
.
C = (PAR1,PAR2)
```

Example of invalid arithmetic assignment statement:

```
B + C = X - 5.1
```
An expression must not appear to the left of the equal sign.

Character Assignment Statement

A character assignment statement has the form:

v = exp

v A character variable, array element, or substring

exp A character expression

The character expression exp is evaluated, and the result is then assigned to v. None of the character positions being defined in v can be referenced in exp.

The variable v and expression exp can have different lengths. If the length of v is greater than the length of exp, the effect is as though exp were extended to the right with blank characters until it is the same length as v. If the length of v is less than the length of exp, the effect is as though exp were truncated from the right until it is the same length as v.

Example:

```
CHARACTER MO*3, DAY*3, DATE*10
.
.
.
DATE = MO//DAY//'1981'
```

The character variables MO and DAY and the string 1981 are concatenated into a single 10-character string and stored into the variable DATE.

Only as much of the value of exp must be defined as is needed to define v. In the example

```
CHARACTER A*2, B*4
A = B
```

the assignment A=B requires that the substring B(1:2) be defined. It does not require that the substring B(3:4) be defined. If v is a substring, exp is assigned only to the substring. The definition status of substrings not specified by v is unchanged.

Logical Assignment Statement

A logical assignment statement has the form:

v = exp

v A logical variable or array element

exp A logical expression

The logical expression is evaluated and the result is then assigned to v. Note that e must have a value of either true or false.

Example:

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

LOG2 is assigned the value false because I is not equal to zero.

Boolean Assignment Statement (CDC Extension)

A boolean assignment statement has the form:

v = exp

v A boolean variable or array element

exp A boolean or arithmetic expression

The boolean or arithmetic expression exp is evaluated. If exp is an arithmetic expression, the result used is BOOL(exp). The result is then assigned to v.

Example:

```
A = B .AND. Z"FF"
```

The lower 8 bits of B are stored in A; the upper 56 bits of A contain zeros.

Multiple Assignment Statement (CDC Extension)

A multiple assignment statement has the form:

v = ...= *v* =**exp**

v A variable, array element, or character substring.

exp An arithmetic, boolean, character, relational, or logical expression. The type of exp must ensure that v = exp is valid for each v specified.

Execution of a multiple assignment statement causes the evaluation of the expression exp. After any necessary conversion, the assignment and definition of the rightmost v with the value of exp occurs. Assignment and definition of each additional v occurs in right-to-left order. The value assigned to each v is the value of the v immediately to its right, after any necessary conversion.

Example:

```
X = Y = Z = 10.0 * SUM(I)
```

This statement is equivalent to:

```
X = 10.0 * SUM(I)
Y = 10.0 * SUM(I)
Z = 10.0 * SUM(I)
```

Flow Control Statements 5

The flow control statements include the GO TO statements, IF statements, DO statements, CONTINUE statement, PAUSE statement, STOP statement, and END statement. The CALL and RETURN statements, while sometimes considered to be flow control statements, are described in chapter 7.

GO TO Statements

The three types of GO TO statements are unconditional GO TO, computed GO TO, and assigned GO TO. The ASSIGN statement is used in conjunction with the assigned GO TO and is therefore described in this chapter.

Unconditional GO TO Statement

The unconditional GO TO statement transfers control to the executable statement identified by the specified label. The unconditional GO TO statement has the form:

GO TO label

label Label of an executable statement

The labeled statement must appear in the same program unit as the GO TO statement.

Example:

```
      GO TO 20
10    A = 0.0
20    A = A + 1.0
      .
      .
      .
```

When the statement GO TO 20 is reached, control transfers to statement 20. Note that the statement following the GO TO can be reached only as a result of another flow control statement.

Computed GO TO Statement

The computed GO TO statement transfers control to the executable statement identified by one of the specified labels. The computed GO TO statement has the form:

GO TO (label,...,*label*), **exp**

label Label of an executable statement that appears in the same program unit as the GO TO statement.

exp An arithmetic or boolean expression. (ANSI allows an integer expression only.) The comma preceding exp is optional.

The label selected is determined by the value of the expression. If exp has a value of one, control transfers to the statement identified by the first label in the list; if exp has a value of i, control transfers to the statement identified by the ith label in the list. The value of exp is truncated and converted to integer, if necessary.

If the value of exp is less than one or greater than the number of labels in the list, execution continues with the statement following the computed GO TO.

Example:

```
GO TO (10,20,30,20) L
```

The next statement executed is:

```
10 if L = 1

20 if L = 2

30 if L = 3

20 if L = 4
```

Example:

```
M = 4
GO TO (100,200,300) M
A = B + C
```

Execution continues with the statement A = B + C because the value of M is greater than the number of labels enclosed in parentheses.

ASSIGN Statement

The ASSIGN statement assigns a statement label to an integer variable. The ASSIGN statement has the form:

ASSIGN label TO iv

label Label of an executable or FORMAT statement

iv Integer variable

The value assigned to iv represents the label of an executable or a FORMAT statement. The labeled statement must appear in the same program unit as the ASSIGN statement. When defined by an ASSIGN statement, iv can be referenced only in an assigned GO TO statement or input/output format specifier.

The assignment must be made prior to execution of the assigned GO TO statement or the input/output statement that references assigned label sl.

Example:

```
ASSIGN 10 TO LSWIT
GO TO LSWIT(5, 10, 15, 20)
```

Control transfers to the statement labeled 10.

Example:

```
ASSIGN 24 TO IFMT
WRITE (2, IFMT) A,B
```

The variables A and B are formatted according to the FORMAT statement labeled 24.

Assigned GO TO Statement

The assigned GO TO statement transfers control to the executable statement last assigned to iv by the execution of a prior ASSIGN statement. The assigned GO TO statement has the form:

GO TO iv, *(label, ..., label)*

iv Integer variable. The comma following iv is optional.

label Label of an executable statement that appears in the same program unit as the assigned GO TO statement. The list of labels, including parentheses, can be entirely omitted.

At the time of execution of an assigned GO TO statement, the variable iv must be defined with a statement label of an executable statement that appears in the same program unit. The list of statement labels is optional. All labels in a statement label list must be in the same program unit as both the ASSIGN and assigned GO TO statements. Also, if the optional list is present, iv must be defined with one of the labels in the list.

Example:

```
      ASSIGN 50 TO JUMP
10    GO TO JUMP(20,30,40,50)
20    CONTINUE
      .
      .
      .
30    CAT = ZERO + HAT
      .
      .
      .
40    CAT = ZERO + RAT
      .
      .
      .
50    CAT = ZERO + BAT
```

Statement 50 is executed immediately after statement 10.

IF Statements

The IF statement evaluates an expression and conditionally transfers control or executes another statement, depending on the outcome of the test. The types of IF statements are as follows:

Arithmetic IF
Logical IF
Block IF

The ELSE, ELSE IF, and END IF statements are used in conjunction with a block IF statement, and are described in this chapter after the block IF statement.

Arithmetic IF Statement

The arithmetic IF statement evaluates an arithmetic expression and conditionally transfers control to another statement. The arithmetic IF statement has the form:

IF (exp) label1, label2, label3

exp	Integer, real, double precision, or boolean expression
label1, label2, label3	Labels of executable statements that appear in the same program unit as the arithmetic IF statement

The arithmetic IF statement transfers control to the statement labeled label1 if the value of exp is less than zero, to the statement labeled label2 if it is equal to zero, or to the statement labeled label3 if it is greater than zero. If exp is type boolean, INT(exp) is used.

Example:

```
      IF (I - N) 3, 4, 5
3     ISUM = J + K
      GO TO 4
5     CALL ERROR1
4     PRINT*, ISUM
      STOP
```

If I is less than N, control transfers to statement 3; if I is equal to N, control transfers to statement 4; if I is greater than N, control transfers to statement 5.

Logical IF Statement

The logical IF statement allows for conditional execution of a statement. The logical IF statement has the form:

IF (exp) stat

exp	Logical expression
stat	Any executable statement except a DO, block IF, ELSE, ELSE IF, END, END IF, or another logical IF statement

If the value of exp is true, statement stat is executed. If the value of exp is false, stat is not executed; execution continues with the next statement.

Example:

```
      IF (P .AND. Q) RES = 7.2
50    TEMP = ANS*Z
```

If P and Q both have the logical value true, RES is set to 7.2; otherwise, RES is unchanged. In either case, statement 50 is executed.

Example:

```
      IF (A .LT. B) GO TO 50
20    Z = T + R
      .
      .
      .
50    Z = Z + 1.0
```

If A is less than B, control transfers to statement 50; otherwise, execution continues with statement 20.

Block IF Statement

The block IF statement provides for conditional execution of a block of executable statements. The block IF statement has the form:

IF (exp) THEN

exp Logical expression

The block IF statement is used with the END IF and, optionally, the ELSE and ELSE IF statements to form block IF structures.

If the logical expression exp has the value true, execution continues with the next executable statement. If exp has the value false, control transfers to an ELSE or ELSE IF statement or, if no ELSE or ELSE IF statements are present, to an END IF statement.

ELSE Statement

The ELSE statement provides for alternate execution of a block of statements within a block IF structure. The ELSE statement has the form:

ELSE

An ELSE statement allows you to construct a block IF structure with one alternate if-block.

An ELSE statement can have a statement label; however, the label cannot be referenced in any other statement.

ELSE IF Statement

The ELSE IF statement provides for alternate execution of a block of statements within a block IF structure, depending on the result of a conditional test. The ELSE IF statement has the form:

ELSE IF (exp) THEN

exp Logical expression

The ELSE IF statement enables you to form a block IF structure with more than one alternate if-block.

An ELSE IF statement can have a statement label; however, the label cannot be referenced by any other statement.

The effect of executing an ELSE IF statement is the same as for a block IF statement.

END IF Statement

The END IF statement terminates a block IF structure. The END IF statement has the form:

END IF

For each block IF statement there must be a corresponding END IF statement. An END IF statement can have a statement label; however, the label cannot be referenced by any other statement.

Block IF Structures

Block IF structures provide for alternative execution of blocks of statements. A block IF structure begins with a block IF statement, ends with an END IF statement and, optionally, includes one ELSE or one or more ELSE IF statements. Each block IF, ELSE, and ELSE IF statement is followed by an associated block of executable statements called an if-block.

The simplest form of a block IF structure is as follows:

IF (exp) THEN

if-block

END IF

In this structure, if exp has the value true, execution continues with the first statement in the if-block. If exp has the value false, control transfers to the statement following the END IF statement. The if-block can contain any number of statements, including block IF statements.

Control can be transferred out of an if-block from inside the if-block. However, control cannot be transferred into an if-block from outside the if-block. An if-block can be entered only through its associated block IF, ELSE IF, or ELSE statement. You cannot transfer control directly to an ELSE, ELSE IF, or END IF statement. However, you can transfer control directly to a block IF statement.

When execution of the statements in an if-block has completed, and if control has not been transferred outside the if-block, execution continues with the statement following END IF.

An example of a simple block IF structure is as follows:

```
IF (I .EQ. 0) THEN
    X = X + DX
    Y = Y + DY
END IF
PRINT 10,X,Y
```

If I is zero, the two succeeding statements are executed; otherwise, the statements are not executed. In either case, execution continues with the statement following END IF.

Note that in the preceding example, and in all succeeding examples, statements within block IF structures are indented. Although this is not a requirement, it is recommended to improve clarity.

A block IF structure can contain one ELSE statement to provide an alternate path of execution within the structure. The general form of a block IF structure containing an ELSE statement is as follows:

IF (exp) THEN

if-block-1

ELSE

if-block-2

END IF

In this structure, if exp has the value true, execution continues with the first statement in if-block-1. When execution of the statements in if-block-1 has completed, and if control has not been transferred out of if-block-1, execution continues with the statement following END IF.

If exp has the value false, control transfers to the first statement in if-block-2. If the last statement in if-block-2 does not transfer control, execution continues with the statement following END IF.

A block IF statement can have at most one associated ELSE statement.

Example:

```
IF (A .LT. B) THEN
    X = A + B
    XSUM = XSUM + X
ELSE
    Y = A*B
    YSUM = YSUM + Y
ENDIF
PRINT 15, X,Y
```

If A is less than B, new values for X and XSUM are calculated; otherwise, new values for Y and YSUM are calculated. In either case, the values of X and Y are printed.

A block IF structure can contain one or more ELSE IF statements to provide for alternate execution of additional if-blocks. This capability allows you to form block IF structures containing a number of possible execution paths depending on the outcome of the associated IF tests. The general form of a block IF structure with two ELSE IF statements is as follows:

IF (exp1) THEN

if-block-1

ELSE IF (exp2) THEN

if-block-2

ELSE IF (exp3) THEN

if-block-3

END IF

In a block IF structure with ELSE IF statements, the initial block IF statement and each ELSE IF or ELSE statement has an associated if-block. At most one if-block in this structure is executed (if no nested levels appear). Each logical expression is evaluated until one is found that has the value true. Control then transfers to the first statement of the associated if-block. When execution of the if-block has completed, and if control has not been transferred outside the if-block, execution continues with the statement following END IF. If none of the logical expressions has the value true and no ELSE statement appears, no if-blocks are executed; execution continues with the statement following END IF. Note that in a block IF structure containing ELSE or ELSE IF statements (and no nested levels), at most one if-block is executed, even if more than one of the specified conditions is satisfied.

If an ELSE statement appears, it must follow the if-block of the last ELSE IF statement. If no logical expression in the block IF statement or ELSE IF statements has the value true, control transfers to the statement following ELSE.

Control can transfer out of a block IF structure from inside any if-block; however, control cannot transfer from one if-block to another if they are at the same nesting level.

The following example illustrates a block IF structure with two ELSE IF statements and an ELSE statement:

```
IF (N .EQ. 2) THEN
    X = 1.0
    Y = 2.0
ELSE IF (N .EQ. 3) THEN
    X = X + 10.0
    Y = Y + 10.0
ELSE IF (N .LT. 0) THEN
    X = 0.0
    Y = 0.0
ELSE
    CALL ERRSUB
ENDIF
```

Nested Block IF Structures

Block IF structures can be nested; that is, any if-block within a structure can itself contain block IF structures. Within a nesting hierarchy, control can transfer from an inner level structure into an outer level structure; however, control cannot transfer from an outer level structure into an inner level structure. The general form of a nested block-IF structure with two levels of nesting is as follows:

```
IF (exp1) THEN
   .
   .
   .
   IF (exp2) THEN
      .
      .
      .
   END IF
      .
      .
      .
END IF
```

Note that nested block IF statements cannot share END IF statements; each block IF statement requires its own associated END IF statement.

Example:

```
IF (X .GT. Y) THEN
    Y = Y + YINCR
    IF (K .EQ. J) THEN
        XT = X
        YT = Y
    ELSE
        K = K + 1
    END IF
ELSE
    X = X + XINCR
END IF
```

The preceding structure contains two levels of nesting. Each level contains a block IF and an ELSE statement. The inner structure is executed only if X is greater than Y. The inner structure contains a legal branch to the outer structure.

DO Statement

The DO statement is used to specify repeated execution of a group of statements. The DO statement has the form:

DO label, var=exp1,exp2,*exp3*

label Label of an executable statement that is to be the final statement of the DO loop. The comma between label and var is optional.

var Integer, real, or double precision variable, called the DO variable.

exp1 Initial parameter.

exp2 Terminal parameter.

exp3 Optional increment parameter; default is 1. If exp3 is omitted, the preceding comma must also be omitted.

exp1, exp2, and exp3 are called indexing parameters; they can be integer, real, double precision, or boolean expressions.

The DO statement and the group of statements to be repeated are referred to as a DO loop. The DO statement determines the range of the DO loop (that is, the statements to be included in the loop) and the number of times the DO loop is to be repeated.

DO Loops

The final statement of a DO loop is an executable statement that must physically follow and reside in the same program unit as its associated DO statement. The final statement must not be an unconditional GO TO, assigned GO TO, arithmetic IF, block IF, ELSE IF, ELSE, END IF, RETURN, STOP, END, or DO statement. If the final statement is a logical IF statement, it can contain any statement except a DO, block IF, ELSE IF, ELSE, END IF, END, or another logical IF.

The range of a DO loop consists of all the executable statements following the DO statement up to and including the final statement. Execution of a DO statement causes the following sequence of operations to occur:

1. The expressions exp1, exp2, and exp3 are evaluated and, if necessary, converted to the type of the DO variable var.
2. DO variable var is assigned the value of exp1.
3. The iteration count is established; this value is determined by the following expression:

   MAX(INT((m2-m1+m3)/m3),mtc)

   m1, m2, m3

   Values of the expressions exp1, exp2, and exp3 respectively, after conversion to the type of var.

   mtc

   Minimum trip count; in CDC FORTRAN, mtc has a value of either one or zero, and is established by the ONE_TRIP_DO parameter on the FORTRAN command or the C$ DO directive. A zero trip count is prohibited in ONE_TRIP_DO mode, and the results of executing such a loop are undefined. In ANSI FORTRAN, mtc has a value of one.
4. If the iteration count is not zero, the range of the DO loop is executed. If the iteration count is zero, execution continues with the statement following the terminal statement of the DO loop; the DO variable retains its most recent value.
5. DO variable var is incremented by the value of exp3.
6. The iteration count is decremented by one.

Steps 4 through 6 are repeated until the iteration count attains a value of zero.

The iteration count of a DO loop must not exceed (2**29)-1, and the following conditions must be satisfied:

| m1 + m3 | < (2**63)-1

| m2 + m3 | < (2**63)-1

NOTE

When type real or double precision indexing parameters are used, accumulated roundoff error in the indexing calculations can cause a loop to execute more times than expected.

If a DO loop appears within an if-block, the DO loop must be entirely contained within that if-block. If a block IF statement appears within the range of a DO loop, the corresponding END IF statement must also appear within the range of that DO loop.

If a DO loop executes zero times, the DO variable value is equal to m1. Otherwise, the value is the most recent value of the DO variable plus the increment parameter value. When control transfers out of a DO loop, the DO variable retains its most recent value.

You can enter a DO loop only through its DO statement, unless (CDC FORTRAN only) you are reentering a loop from the extended range of that loop. (See Extended Range DO Loops.)

Example:

```
      DO 10 J=1,11,3
      IF (A(J) .LE. A(J+1)) ITEMP = A(J)
   10 A(J) = A(J+1)
      PRINT 100, A
```

The statements following DO, up to and including statement 10, are executed four times (J will equal 1, 4, 7, and 10). The PRINT statement is then executed. When the loop is complete, J will equal 13.

Example:

```
      DO 10 I=5,1,-1
   10 IF (X .GT. B(I)) B(I) = 0.0
      PRINT 500, B
```

This example illustrates the use of a negative increment parameter. The loop compares X with elements 5, 4, 3, 2, and 1 respectively, of array B. After the DO loop has completed, array B is printed. The DO variable I has a value of zero when the DO loop has completed.

Example:

```
      DO 20 I=1,200
      IF (I .GE. IVAR) GO TO 10
   20 A(I) = 0.0
   10 PRINT 100, A
```

A legal exit from the DO loop is made when the value of the DO variable I is equal to IVAR.

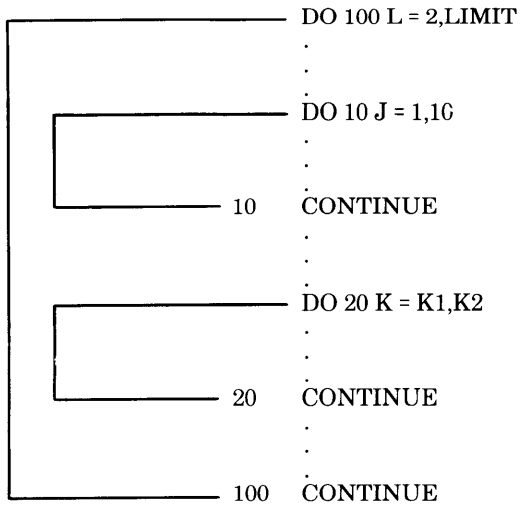
Nested DO Loops

When a DO loop entirely contains another DO loop, the inner loop is called a nested DO loop. The range of a DO statement can include other DO statements providing each inner DO statement and its final statement are within the containing DO loops.

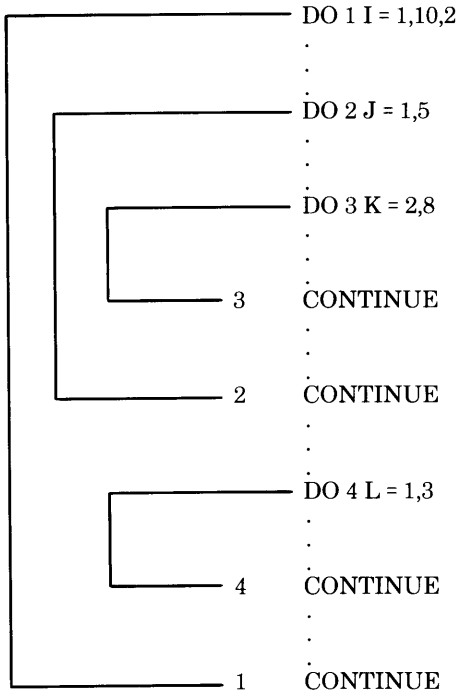
The final statement of an inner DO loop must be either the same as the final statement of the outer DO loop or must occur before it. If more than one DO loop has the same final statement, a transfer to that statement can be made only from within the range (or extended range in CDC FORTRAN) of the innermost DO.

The following examples show some possible DO loop nests. Note that loops can be completely nested or can share a final statement.

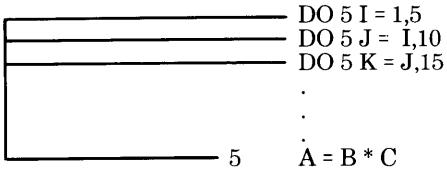
Example 1



Example 2



Example 3



The following example illustrates a nested DO loop:

```
      DO 200 I=1,10
      A(I) = A(I) + 1.0
          DO 100 J = 1,20
100       IF (A(I) .GE. B(J)) A(I) = 0.0
200   CONTINUE
```

The outer loop is executed 10 times. On each pass through the outer loop, the inner loop is executed 20 times, Thus, the inner loop is executed a total of 200 times. Note that the inner loop is indented. Although this is not a requirement, it can improve program clarity.

In the following example, all elements of array A are set to zero:

```
DIMENSION A(3,4,5)
DO 10 I=1,5
DO 10 J=1,4
DO 10 K=1,3
10 A(K,J,I) = 0.0
```

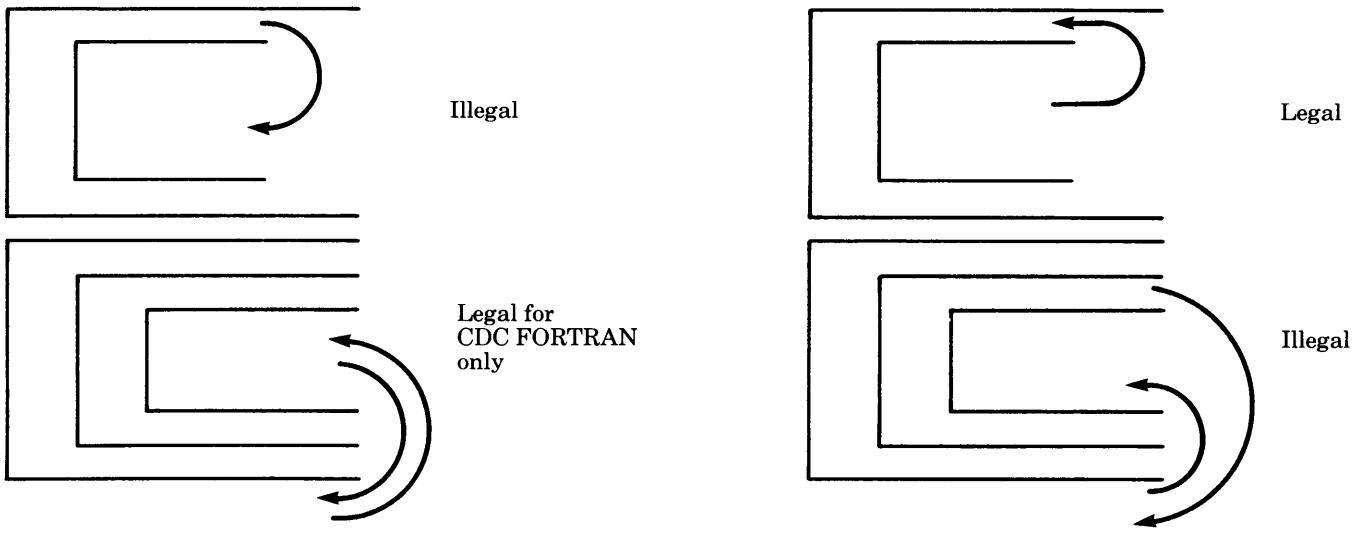
A DO loop can be initially entered only through the DO statement. However, a loop can be reentered from its extended range (CDC extension; see Extended Range DO Loops).

When you use an IF or GO TO statement to bypass one or more inner loops, the bypassed loops require a different final statement. For example,

```
      DO 10 I=1,100
      IF (I .GT. IVAR) GO TO 10
          DO 20 J=1,10
          .
          .
          .
20        CONTINUE
10    CONTINUE
```

In this example, the inner and outer loops cannot share a final statement.

A transfer from an outer DO loop into an inner DO loop is not allowed, unless the transfer is from the extended range of the inner loop. However, a transfer from an inner DO into outer DO is allowed because such a transfer is still within the range of the outer DO loop. A transfer back into an inner DO loop from the extended range of that loop is allowed (CDC extension; see Extended Range DO Loops). Subprograms can be called from within a DO loop. Legal and illegal transfers within a nest of DO loops are shown in the following illustration:



Extended Range DO Loops (CDC Extension)

Under certain conditions, control can be transferred out of a loop and can subsequently be tranferred back into the loop. The statements executed outside the loop constitute the extended range of the loop. Control can transfer from the extended range back into a loop only if the following conditions are satisfied:

The DO variable is not redefined outside the loop.

The program unit containing the loop has not been exited by a RETURN, STOP, or END statement.

The iteration count of the loop is nonzero.

If any of these conditions are not satisfied, the loop cannot be reentered except through its DO statement.

When DO loops are nested, an inner loop can be exited and subsequently reentered subject to the preceding restrictions.

CONTINUE Statement

The CONTINUE statement performs no operation. This statement has the form:

CONTINUE

CONTINUE is an executable statement that can be placed anywhere in the executable statement portion of a source program without affecting the sequence of execution. The CONTINUE statement is often used as the last statement of a DO loop. It can end a DO loop when a GO TO or arithmetic IF would normally be the last statement of the loop. For example, the following DO loop is illegal because it ends with an arithmetic IF statement:

```
4    DO 5 I=1,100
        A(I) = A(I) + DELT
5    IF (I-J) 4, 6, 4
```

This loop executes correctly if it is changed to the following:

```
     DO 5 I=1,100
        A(I) = A(I) + DELT
        IF (I-J) 5, 6, 5
5    CONTINUE
```

If a CONTINUE statement does not have a label, an informative diagnostic is issued.

PAUSE Statement

The PAUSE statement causes program execution to temporarily stop. This statement has the form:

PAUSE *n*

n A character constant of 1 through 57 characters (CDC limitation), or a string of 1 through 5 decimal digits.

When a PAUSE statement is executed in a batch program, execution stops and PAUSE n is displayed on the operator console. The operator can continue or terminate the program with an entry from the console.

In an interactive program, execution stops and PAUSE n is displayed at your terminal. You can terminate the program by entering the break sequence defined for the terminal. Any other entry causes execution to continue.

Examples:

```
PAUSE 'EXAMPLE TWO'

PAUSE 45321
```

STOP Statement

The STOP statement ends program execution. This statement has the form:

STOP *n*

n A character constant of 1 through 57 characters (CDC limitation), or a string of 1 through 5 decimal digits.

When a STOP statement is encountered during execution, STOP n is displayed in the job log file, the program terminates, and control returns to the operating system. If you omit n, no message is displayed. A program unit can contain more than one STOP statement.

Examples:

```
STOP

STOP 'PROGRAM HAS ENDED'
```

END Statement

The END statement indicates the end of the program unit to the FORTRAN compiler. This statement has the form:

END

Every program unit must have an END statement as the last statement.

The END statement can be labeled. If control flows into or branches to an END statement in a main program, execution stops. If control flows into or branches to an END statement in a function or subroutine, action is the same as if a RETURN statement were encountered.

An END statement cannot be continued; it must be completely contained on an initial line. A line following an END statement is considered to be the first line of the next program unit, even if it has a continuation character in position 6.

Example:

```
      GO TO 15
      .
      .
      .
15    END
```

RETURN Statement

The RETURN statement is described in chapter 7, Program Units.

CALL Statement

The CALL statement is described in chapter 7, Program Units.

Input/Output 6

The FORTRAN input/output operations involve reading records from files and writing records to files. External files reside on an external storage device, such as a disk. Internal files reside in memory. The term file, as used in this manual, refers to an external file. Internal files are described later in this chapter under Internal Input/Output.

Introduction to Input/Output

This section presents some terms and concepts you should know before using the FORTRAN input/output statements.

Using Files

An external file is a collection of data that begins at the beginning-of-information (BOI) and ends at the end-of-information (EOI). A file with V type records can contain one or more partition boundaries. (Partition boundaries are not allowed on files with F type records.) A partition boundary and the end-of-information are recognized as the end-of-file by the FORTRAN input/output statements. The ENDFILE statement writes a partition boundary.

Throughout the remainder of this chapter, the term end-of-file means either end-of-partition or end-of-information.

A record is generally considered to be the amount of data transferred by a single input or output operation. Execution of any input or output statement transfers at least one record. (Formatted input/output statements can transfer more than one record.)

File Names

Every file referenced in a FORTRAN program must have a valid NOS/VE file name. A NOS/VE file name consists of 1 through 31 characters chosen from among A through Z (lowercase letters are equivalent), 0 through 9, @, _, #, and $; the first character must not be a digit. The names are local file names (that is, existing files must reside in the $LOCAL catalog, and new files are created in the $LOCAL catalog).

File Attributes

Every NOS/VE file has associated with it a set of file attributes. These attributes completely describe the structure and processing limitations of the file. The file attributes are stored in an internal table that is created and maintained by NOS/VE. These attributes include file organization, record type, record length, and many others. Files read and written by the FORTRAN input/output statements are provided with default values for the file attributes, and in most cases, with the exception of the FORTRAN file interface subprograms, FORTRAN programmers need not be concerned with the file attributes or with the internal operations of NOS/VE. Certain file attributes can be overridden by a SET_FILE_ ATTRIBUTE (SETFA) command executed before the first time the file is opened. If your application requires you to specify the attributes of a file, you should refer to the discussion of I/O implementation in appendix E. This appendix describes the attributes of FORTRAN files and indicates which ones can be overridden by a SETFA command.

File Access Methods

FORTRAN provides two methods of accessing records in a file. The access methods are sequential access and random access.

In a sequential access file, records are written in sequential order and must be read in the same order in which they were written. You can access a sequential record only by reading sequentially until the desired record is found.

Sequential files are read and written by FORTRAN READ and WRITE statements, BUFFER IN and BUFFER OUT statements, and PRINT and PUNCH statements. You can perform formatted, unformatted, list directed, buffer, and namelist input/output on sequential files.

To create a sequential access file, or to reference an existing one, you can specify the ACCESS='SEQUENTIAL' specifier on the OPEN statement for the file. However, if you omit this specifier, the file is assumed to be a sequential access file.

Random access files provide a quicker method of access to a specific record than sequential access files. Records on a random access file can be read or written in any order by specifying a record key or a record number. Random access files eliminate the need for sequentially searching a file for a particular record.

FORTRAN provides three ways of performing random access input/output. The first way is to use the direct access file capability. Direct access files are processed by standard FORTRAN READ and WRITE statements. To create a direct access file, or to reference an existing one, you must specify ACCESS='DIRECT' on the OPEN statement for the file. You can use direct access files for formatted or unformatted input/output. You cannot use direct access files for namelist, buffer, or internal input/output. Direct access files are described in this chapter under Direct Access Files.

A second method of random access is provided by the mass storage subroutines. These files, often referred to as mass storage files, must be processed by the mass storage subroutines; they cannot be processed by READ and WRITE statements.

A third method of random access is provided by the FORTRAN-callable file interface subprograms. These subprograms enable you to perform operations on files having indexed sequential organization. Records on indexed sequential files can be accessed directly by record key, without the need for a more time-consuming sequential search. Indexed sequential files provide an efficient and flexible method of input/output for applications requiring a random access capability. The file interface subprograms, along with the background information required to use those subprograms, are described in this chapter under File Interface Subprograms.

If you decide to use one of the random access methods, note that the FORTRAN direct access capability is an ANSI standard feature, whereas the mass storage and file interface capabilities are CDC extensions, and will inhibit program portability. Also, direct access files have fixed length records, while both mass storage and file interface allow variable length records. The file interface capability offers the most flexibility.

Opening Files

Before you reference a new or existing file in a FORTRAN program, the file must be opened. The opening process prepares the file for input/output and establishes certain file properties, such as unit/file association, record length, access method, and so forth.

You can open a file explicitly by declaring it in an OPEN statement. (Mass storage files require an OPENMS call. Indexed sequential files requre an OPENM call.) Alternatively, for sequential files only, you can simply reference a unit in an input/output statement, in which case the file associated with that unit is automatically opened and default values are provided for the various file properties.

After you have finished reading or writing a file, you can close the file by specifying a CLOSE statement (CLOSMS for mass storage files; CLOSEM for indexed sequential files). The CLOSE statement performs various file completion operations. If you do not explicitly close a file, it is automatically closed either when the program terminates or when the job terminates.

Input/Output Units

All files are referenced in FORTRAN input/output statements by means of an associated unit. For example, the statement

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

specifies unit number 2. The read operation is performed on the file associated with unit 2.

You can associate a file with a unit through parameters on the OPEN statement or with a CREATE_FILE_CONNECTION command prior to execution. If you reference a unit number that has not been associated with a file through an OPEN statement or system command, the file name defaults to TAPEn, where n is the unit number you specified. For example, if a program contains the statement

```
READ (UNIT=9, FMT=150), X, Y
```

and unit 9 is not associated with a file name on an OPEN statement, then a file named TAPE9 is read.

You can also specify unit names in input/output statements using boolean L format. (This capability is provided mainly for compatibility with previous systems.) In this case, the file name is the same as the unit name. File names specified in this manner are limited to seven characters. For example,

```
READ (L"AFILE", 100) A, B
```

reads from a file named AFILE.

A unit specification of the form L"TAPEn" is the same as specifying unit number n. That is, the file associated with unit n is used or, if no file is associated with unit n, the file name defaults to TAPEn. For example, the following statements are equivalent:

```
READ (UNIT=L"TAPE1", FMT=5) AV, BV

READ (UNIT=1, FMT=5) AV, BV
```

Both statements read from file TAPE1 (assuming unit 1 has not been associated with a file name in an OPEN statement).

FORTRAN provides several standard units for use in input/output statements. If you specify an asterisk for the unit identifier in an input/output statement, or if you omit the list of specifiers entirely, the unit defaults to one of the standard units. The standard units have a default association with a standard system file. The system files do not contain data, but are connected to physical files that contain data. The standard units, and the default files associated with those units, are as follows:

Standard Unit	Standard System File	Physical File	Description
INPUT	$INPUT	INPUT	In an interactive job, data is read from the terminal. In a batch job, INPUT is a null file.
OUTPUT	$OUTPUT	OUTPUT	In an interactive job, data is written to the terminal. In a batch job, data is printed at job termination.
PUNCH	PUNCH	PUNCH	Data is written to file PUNCH.
_	$NULL	NULL	Data written to file $NULL is discarded.

Standard system files cannot be redefined (that is, opened with STATUS='NEW' on the OPEN statement), but they can be reconnected to different physical files.

Input/Output Restrictions

Mixing types of operations on the same file can destroy file integrity. In particular, files processed by mass storage or indexed sequential subroutines should be processed only by those subroutines.

For every formatted, list directed, namelist, or unformatted READ, you can check for end-of-file by using the END= or IOSTAT= specifier in the READ statement. If an end-of-file (end-of-partition) is encountered and a test is not included, the program terminates with a fatal error.

Record length on print files should not exceed the length of a printer line (usually 133 or 137 characters). The default maximum record length is 150 characters. The first character is always used for printer control and is not printed. The second character appears in the first print position. Printer control characters are listed in this chapter under Format Specification.

Input/Output Statement Specifiers

Specifiers are used in input/output statements to select various processing options. The input/output specifiers have the following keyword=value forms:

UNIT=u

Specifies the FORTRAN unit or internal file to be used. The unit name is derived from the unit identifier u, which can be one of the following:

An asterisk implying unit INPUT in a READ statement and unit OUTPUT in a WRITE statement. The default file for unit INPUT is $INPUT; the default file for unit OUTPUT is $OUTPUT.

The name of a character variable, array, array element, or substring identifying an internal file.

An integer or boolean expression having the following characteristics:

INT(u) has a value in the range 0 through 999. The compiler associates these numbers with unit names of the form TAPEu.

or

BOOL(u) is an ASCII coded name in boolean L format (left-justified with binary zero fill). This is the unit name. If this name is of the form TAPEk, where k is an integer in the range 0 through 999 with no leading zero, it is equivalent to the integer k for the purpose of identifying external units. A valid unit name consists of one through seven letters or digits beginning with a letter. (Uppercase and lowercase letters are equivalent.)

The characters UNIT= can be omitted, in which case u must be the first item in the list of specifiers.

File names default to the unit name unless a different file name has been specified using execution command file name substitution, PROGRAM statement equivalencing, or an OPEN statement.

FMT=fn

Specifies a format to be used for formatted input/output; fn can be one of the following:

A statement label identifying a FORMAT statement in the program unit containing the input/output statement.

A character array, array element, or variable containing the format specification.

A noncharacter array containing the format specification.

A character expression. (Note that a character constant is permitted.)

An integer variable that has been assigned the statement number of a FORMAT statement by an ASSIGN statement.

An asterisk, indicating list directed I/O.

A namelist group name.

The characters FMT= can be omitted, in which case the format specifier must be the second item in the list of specifiers, and the first item must be the unit specifier without the characters UNIT=.

REC=rn

Specifies the number of the record to be read or written in the file; rn must be a positive nonzero integer. Valid only if the unit is open for direct access.

END=sl

Specifies the label of an executable statement to which control transfers when an end-of-file is encountered during an input operation.

ERR=sl

Specifies the label of an executable statement to which control transfers if an error condition is encountered during input/output processing.

IOSTAT=ios

Specifies an integer variable into which one of the following values is returned after the input/output operation is complete:

< 0 End-of-file encountered

= 0 Operation completed normally

> 0 Error number in the range 1 through 999.

All runtime errors under NOS/VE are identified by a six-digit condition code. For FORTRAN errors, the condition code is determined by adding the base number 585000 to the error number. Runtime errors are listed by condition code in the Diagnostic Messages for NOS/VE manual, which provides descriptions and suggested action for the errors.

iolist

Input/output list specifying items to be transmitted (described under Input/Output Lists).

Input/Output Lists

The list portion of an input/output statement specifies the items to be read or written and the order of transmission. The input/output list can contain any number of items. List items are read or written sequentially from left to right.

If no list appears on input, one or more records are skipped. If no list appears on formatted output, only information completely contained within the FORMAT statement, such as character strings, is transmitted. If no list appears on unformatted output, a null (empty) record is transmitted.

A list item can be a variable name, an array name, an array element name, a character substring name, or an implied DO list. On output, a list item can also be a character, boolean, logical, or arithmetic expression. No expression in an input/output list can reference a function if such reference would cause any input/output operations to be executed or would cause the value of any element of the input/output statement to be changed. List items are separated by commas.

An array name without subscripts in an input/output list specifies the entire array in the order in which it is stored. The entire array (not just the first word of the array) is read or written. You cannot use assumed-size array names in input/output lists. (Assumed-size array element names are permitted.)

Subscripts in an input/output list can be any valid subscript (as described in chapter 2).

The following input/output statements illustrate typical input/output lists:

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

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

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

WRITE (2, 202) DELTA

WRITE (4, 102) DELTA (5*J+2, 5*I-3, 5*K), C, D(I+7)
```

On formatted input or output, the input/output list is scanned and each item in the list is paired with the edit descriptor provided by the FORMAT statement. After one item has been input or output, the next edit descriptor is taken together with the next element of the list, and so on until the end of the list. For example, the statements

```
      READ (5, 20) L, M, N
20    FORMAT (I3, I1, I7)
```

read the input record

```
100223456712
```

as follows:

100 is read into the variable L under the specification I3.

22 is read into the variable M under the specification I1.

3456712 is read into N under the specification I7.

On unformatted input or output, the list items are transmitted between memory and the storage device with no formatting.

Implied DO List in I/O List

An implied DO list in an input/output list has the following form:

(dlist, i=exp1, exp2, *exp3*)

The values of the DO variable i and the iteration count are established in the same way as for DO loops. In an input statement, the DO variable must not appear as a list item in dlist. The range of an implied DO list is the list of elements in dlist.

When an implied DO list appears in an input/output list, the items in dlist are specified once for each iteration of the implied DO, with appropriate substitution of values for any occurrence of the DO variable. For example, the following statements have the same effect:

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

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

Example:

```
      READ 100, (A(I), I=1,2)
100   FORMAT (F10.3)
```

These statements read two records, each containing a value for A. This example is equivalent to the following:

```
      READ 100, A(1), A(2)
100   FORMAT (F10.3)
```

The value of the DO variable i must not be redefined within the range of the implied DO list by a READ statement. For example, the following statement is illegal:

```
READ *, (I,A(I), I=1, 10)
```

Changes to the values of exp1, exp2, and exp3 have no effect upon the execution of the implied DO. However, their values can be changed in a READ statement if they are outside the range of the implied DO, and the change does have effect. For example, the statement

```
READ 100, K, (A(I), I=1, K)
```

transmits a value to K. That value is then used as the terminal parameter of the implied DO.

You can use an implied DO list to transmit a list item more than one time. For example, the list (B, K=1, 5) causes the list item B to be transmitted five times.

Example:

```
WRITE (3, 20) (CAT, DOG, RAT, I=1,10)
```

This statement writes CAT, DOG, and RAT 10 times each.

A variable cannot be used as a DO variable more than once in the same implied DO nest, but iolist items can appear more than once. The value of a DO variable within an implied DO list is defined within that DO list. When the implied DO has completed, the DO variable retains the first value to exceed the upper limit.

Implied DO lists can be nested; that is, the iolist in an implied DO list can itself contain implied DO lists. The first (innermost) DO variable varies most rapidly, and the last (outermost) DO variable varies least rapidly. For example, a nested implied DO with two levels has the form:

((list, v1=exp1, exp2, *exp3*), **v2=exp4, exp5**, *exp6*)

Nested implied DO lists are executed in the same manner as nested DO statements.

Example:

```
      DIMENSION VECT(3, 4, 7)
      READ (3, 100) VECT
  100 FORMAT (I6)
```

This sequence is equivalent to the following:

```
DIMENSION VECT(3,4,7)
READ (3, 100) (((VECT(I, J, K), I=1,3), J=1,4), K=1,7)
```

Example:

```
READ (1, 100) ((E(I,J), J=1,3), I=1,3)
```

This statement transmits nine elements into the array E in the order: E(1,1), E(1,2), E(1,3), E(2,1), E(2,2), E(2,3), E(3,1), E(3,2), E(3,3):

Example:

```
      READ (5, 100) (VECTOR (I), I=1, 10)
  100 FORMAT (F7.2)
```

These statements read data (consisting of one data item per record) into the elements 1 through 10 of the array VECTOR. The following statements have the same effect (but are less efficient):

```
      DO 40 I = 1,10
   40 READ (5, 100) VECTOR (I)
  100 FORMAT (F7.2)
```

In this example, numbers are read, one from each record, into the elements VECTOR(1) through VECTOR(10) of the array VECTOR. The READ statement is encountered each time the implied DO list is executed; and a new record is read for each element of the array.

If the format specification F7.2 is changed to 4F7.2, only three records are read by the first example; the second example still reads 10 records. Both examples read 10 values.

Formatted Input/Output

For formatted input/output, a format specifier must be present in the input/output statement. The input/output list is optional. Each formatted input/output statement transfers one or more records. Each record is zero or more characters in length. The formatted input/output statements are READ, WRITE, PRINT, and PUNCH.

Formatted READ

The formatted READ statement transmits data from a storage device to internal storage, and converts the data according to a format specification. This statement has the forms:

READ (*UNIT*=**u,** *FMT*=**fn,** *IOSTAT=ios, ERR=sl, END=sl*) *iolist*

READ fn, *iolist*

The UNIT=, FMT=, IOSTAT=, ERR=, and END= specifiers are described earlier in this chapter under Input/Output Statement Specifiers. The iolist is described under Input/Output Lists.

The first form transmits data from the file associated with unit u to storage locations named in iolist according to FORMAT specification fn. The second form performs the same operation for unit INPUT. The number of items in the list and the FORMAT specifications must conform to the record structure on the input unit. If the list is omitted, a record will be bypassed. (Slash descriptors in the format specification will cause additional records to be bypassed.)

Each execution of a READ statement transmits at least one record. The FORMAT statement determines when a new record will be read. For example, the statement

```
    READ (5, 100) (VEC(I), I=1,10)
100 FORMAT (5F7.2)
```

reads data (consisting of one data item per record) into the first 10 elements of array VEC.

You should specify the END= or IOSTAT= specifier to test for an end-of-file (end-of-partition or end-of-information). If neither is specified, and an an end-of-file is encountered, the program terminates with a fatal error. A fatal error also occurs if you attempt to read a unit after an END= or IOSTAT= specifier has returned an end-of-file condition for that unit. Records following an end-of-partition can be read by issuing a CLOSE followed by an OPEN on the file or by calling the EOF function.

Example:

```
READ (4, 200) A, B, C
```

This statement reads data from unit 4 into the variables A, B, and C according to the specifications in the format statement labeled 200.

Example:

```
READ 5, X, Y, Z
```

This statement reads data from unit INPUT (unit specifier omitted) to the variables X, Y, and Z according to the specifications in the format statement labeled 5.

Example:

```
READ (2, 100, ERR=16, END=18)
```

This statement reads data from the file associated with unit 2 into variables A and B according to format 100. If an error occurs during the read, control transfers to the statement labeled 16; if an end-of-file is encountered, control transfers to the statement labeled 18.

Example:

```
READ (2, '(2f10.4)') a, b
```

This statement reads data into a and b according to the format specification 2f10.4.

Formatted WRITE

The formatted WRITE statement transfers data from the storage locations named in the input/output list to the file associated with the unit specified by u. This statement has the form:

WRITE (*UNIT*=**u**, *FMT*=**fn**, *ERR=sl, IOSTAT=ios*) *iolist*

The UNIT=, FMT=, ERR=, and IOSTAT= specifiers are described earlier in this chapter under Input/Output Statement Specifiers. The iolist is described under Input/Output Lists.

The data is converted from internal format to coded format according to the format specification fn.

Each execution of a WRITE statement transmits at least one record. The FORMAT statement determines when a new record will be transmitted. If the iolist is omitted, a null record is written.

Example:

```
WRITE (UNIT=4, FMT=50) A, B
```

This statement writes data from variables A and B to unit 4 according to the FORMAT statement labeled 50. The keywords UNIT= and FMT= in the unit and format specifiers are optional.

Example:

```
WRITE (*, 12) L, M, S(3)
```

This statement writes data from variables L, M, and S(3) to unit OUTPUT (as indicated by an asterisk in place of the unit specifier) according to the format statement labeled 12.

Example:

```
WRITE (4, 50, ERR=200) A, B
```

This statement is identical to the first example except that if an error occurs during the write, control transfers to statement 200.

Example:

```
WRITE (2, '(3E12.4)') XVAR, YVAR
```

This statement writes variables XVAR and YVAR to the file associated with unit 2 according to the specification 3E12.4. This example illustrates the inclusion of a format specification in the WRITE statement.

Formatted PRINT

The PRINT statement transfers information from the storage locations named in the input/output list to unit OUTPUT according to the specified format. This statement has the form:

PRINT fn, *iolist*

iolist is optional. The format specification fn is described earlier in this chapter under Input/Output Statement Specifiers. The iolist is described under Input/Output Lists.

Example:

```
PRINT 4, A, B, N
```

This statement prints data from A, B, and N according to the format statement labeled 4.

Example:

```
PRINT '(3E14.4)', X1, X2, X3
```

This statement prints data from X1, X2, and X3 according to the format specification 3E14.4.

Formatted PUNCH (CDC Extension)

The PUNCH statement transfers data from the specified storage locations to the unit PUNCH. This statement has the form:

PUNCH fn, *iolist*

The format specification fn is described earlier in this chapter under Input/Output Statement Specifiers. The iolist is described under Input/Output Lists.

Example:

```
PUNCH 5, A, B, C, ANSWER
```

This statement writes data from variables A, B, C, and ANSWER according to the FORMAT statement labeled 5.

Example:

```
      PUNCH 30
30    FORMAT (' GOOD MORNING')
```

This statement writes according to the FORMAT statement labeled 30. Since no iolist is specified, no data is transferred and converted from variables.

Format Specification

Format specifications are used in conjunction with formatted input/output statements to produce output or read input that consists of strings of ASCII characters. On input, data is converted from a specified format to an internal representation. On output, data is converted from an internal representation to the specified format and written as records of ASCII characters. Format specifications are identified by the FMT= specifier on an input/output statement. This specifier can identify one of the following:

The statement label of a FORMAT statement.

An integer variable which has been assigned the statement label of a FORMAT statement (see ASSIGN Statement, chapter 5).

A character array name or any character expression, except one involving assumed-length character entities. The value of the expression must be a valid format specification.

The name of a noncharacter array that contains a format specification.

FORMAT Statement

FORMAT is a nonexecutable statement which specifies the formatting of data to be read or written with formatted input/output. This statement has the form:

sl FORMAT (item, ..., *item*)

sl Statement label

item One of the following:

A nonrepeatable edit descriptor

A repeatable edit descriptor optionally preceded by a repeat count

A list of repeatable or nonrepeatable edit descriptors, enclosed in parentheses, and optionally preceded by a repeat count

The FORMAT statement is used with formatted input and output statements, and it can appear anywhere in a program unit after the PROGRAM, FUNCTION, or SUBROUTINE statement. An example of a READ statement and its associated FORMAT statement is as follows:

```
      READ (5, 100) INK, NAME, AREA
  100 FORMAT (10X, I4, I1, F7.2)
```

The format specification consists of a sequence of edit descriptors enclosed in parentheses. Blanks are not significant except in H, quote, and apostrophe edit descriptors.

Generally, each item in an input/output list is associated with a corresponding edit descriptor in a FORMAT statement. The FORMAT statement specifies the external format of the data and the type of conversion to be used. Editing complex variables always requires two floating-point (D, E, F, or G) edit descriptors; the two descriptors can be different. Double precision variables correspond to one floating-point edit descriptor. The D edit descriptor corresponds to exactly one list item.

The type of conversion should correspond to the type of the variable in the input/output list. The FORMAT statement specifies the type of conversion for the input data, with no regard to the type of the variable which receives the value when reading is complete. For example, the statements

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

read a floating-point number into the variable N, which could cause unpredictable results if N is referenced later as an integer.

Character Format Specification

A format specification can also be a character expression, or a character array that contains a format specification. (The character expression must not involve concatenation of an operand whose length specification is an asterisk in parentheses unless the operand is a symbolic constant.) The character expression is substituted for the FORMAT statement number in the FMT= specifier of the READ or WRITE statement. The form of these format specifications is the same as for FORMAT statements without the keyword FORMAT. Any character information beyond the terminating parenthesis is ignored. The initial left parenthesis can be preceded by blanks. For example, the sequence

```
CHARACTER FORM*11
DATA FORM/'(I3,2E14.4)'/
READ (2, FMT=FORM, END=50) N, A, B
```

is equivalent to

```
      READ (2,FMT=100,END=50) N, A, B
  100 FORMAT (I3,2E14.4)
```

The preceding examples can also be expressed as

```
READ (2,FMT='(I3, 2E14.4)',END=50) N, A, B
```

or

```
CHARACTER FORM*(*)
PARAMETER (FORM='(I3,2E14.4)')
READ (2,FMT=FORM,END=50) N, A, B
```

If a format specification is contained in a character array, the specification may occupy two or more contiguous array elements. Only the array name need be specified in the input/output statement; all information up to the closing parenthesis is considered to be part of the format specification. For example, the statements

```
CHARACTER AR(2)*10
DATA AR/'(10X,2I1,1','0X,F6.2)'/
READ (5, AR) I, J, X
```

read data into I, J, and X according to the format specification contained in the character array elements AR(1) and AR(2). These statements are equivalent to

```
    READ (5, 100) I, J, X
100 FORMAT (10X, 2I1, 10X, F6.2)
```

Noncharacter Format Specification (CDC Extension)

You can place format specifications in a noncharacter array. The rules for noncharacter format specifications are the same as for character format specifications.

Execution-Time Format Specification

Format specifications can be read dynamically at execution time. The format can be read under the A specification and stored in a character array, variable, or array element; or it can be included in a DATA statement. Formats can also be generated by the program at execution time. (Note, however, that execution-time format specifications cannot be interpreted by the compiler, and are therefore less efficient.)

If you use an array to store a format specification, its type can be other than character. In either case, the format must consist of a list of descriptors and editing characters enclosed in parentheses, but without the keyword FORMAT and the statement label.

The name of the entity containing the specification is used in place of the FORMAT statement number in the associated input/output statement. The name specifies the location of the format information. For example, the input string

```
(E7.2,G20.5,F7.4,I3)
```

can be read and subsequently referenced as follows:

```
CHARACTER F*30
READ (2, '(A)') F
WRITE (3, F) A, B, C, N
```

The preceding example produces the same output as the following statements:

```
    WRITE (3, 10) A, B, C, N
10  FORMAT (E7.2, G20.5, F7.4, I3)
```

A program can create a format specification at execution time. For example, the following statements define a format specification containing a printer control character; if the variable PRTFLG is zero, the printer control character is removed:

```
CHARACTER FMT*9
DATA FMT/'(1X, 3I10)'/
IF (PRTFLG .EQ. 0) FMT (2:4)='    '
WRITE (2, FMT) I, J, K
```

If PRTFLG is zero, the program produces the same result as WRITE (2, '(3I10)') I, J, K.

Edit Descriptors

Format specifications are composed of edit descriptors, which specify the data conversions to be performed. Tables 6-1 and 6-2 list the edit descriptors and give a brief description of each. The descriptors listed in table 6-1 can be preceded by an unsigned nonzero decimal integer indicating the number of times the descriptor is to be repeated (as described later in this chapter under Repeatable Edit Descriptors).

Table 6-1. Repeatable Edit Descriptors

Descriptor	Descriptor Type	Description
Ew.d	Numeric	Floating-point with exponent.
Ew.dEe	Numeric	Floating-point with explicitly specified exponent length.
Fw.d	Numeric	Floating-point without exponent.
Dw.d	Numeric	Floating-point with exponent.
Gw.d	Numeric	Floating-point with or without exponent.
Gw.dEe	Numeric	Floating-point with or without exponent (if exponent is present, exponent length is explicitly specified).
Iw	Numeric	Decimal integer.
Iw.m	Numeric	Decimal integer with minimum number of digits.
Lw	Logical	Logical
A	Character	Character with data-dependent length.
Aw	Character or Boolean	Character or boolean with specified length.
Rw	Boolean	Boolean conversion.
Ow	Boolean	Octal integer.
Ow.m	Boolean	Octal integer with leading zeros and minimum number of digits.
Zw	Boolean	Hexadecimal integer.
Zw.m	Boolean	Hexadecimal with leadings zeros and minimum number of digits.

Table 6-2. Nonrepeatable Edit Descriptors

Descriptor	Descriptor Type	Description
SP	Numeric output control	Plus signs (+) produced.
SS		Plus signs (+) suppressed.
S		Plus signs (+) suppressed.
nX	Tabulation control	Position forward.
Tn		Position to column n.
TRn		Position forward n columns.
TLn		Position backward n columns.
nH	Character output	Output character string.
"		Output character string.
'		Output character string.
:	Format control	Terminates format control if no more items in iolist.
/	End of record	Indicates end of current input or output record.
kP	Scale factor	Scaling for numeric editing.
BN	Numeric input control	Blanks ignored.
BZ		Blanks treated as zeros.

The following symbols, representing information that you must supply, are used in discussion of the edit descriptors:

- w Nonzero unsigned integer constant specifying the field width in number of character positions in the external record. This width includes any leading blanks, + or - signs, decimal point, and exponent.
- d Unsigned integer constant specifying the number of digits to the right of the decimal point within the field. On output all numbers are rounded to d digits.
- e Nonzero unsigned integer constant specifying the number of digits in the exponent; the value of e cannot exceed six.
- m Unsigned integer constant specifying the minimum number of digits to be output.
- k Integer constant scale factor (used with P descriptor).
- n Positive nonzero decimal integer. The meaning of this value depends on the particular edit descriptor.

You must specify the field width w for all edit descriptors except A.

Field separators are used to separate descriptors and groups of descriptors. The format field separators are the slash, the comma, and the colon. (The slash is also used to specify demarcation of formatted records; the colon terminates format control if more items are in the iolist.)

Leading blanks are not significant in numeric input conversions; other blanks in numeric conversions are ignored unless BLANK='ZERO' was specified for the file on an OPEN statement or a BZ edit descriptor is in effect. You can omit plus signs. An all-blank field is considered to be zero, except for logical input, where an all-blank field is considered to have the logical value false.

For the E, F, G, and D input conversions, a decimal point in the input field overrides the decimal point specification of the edit descriptor.

The output field is right-justified for all output conversions. If the number of characters produced by the conversion is less than the field width, leading blanks are inserted in the output field unless w.m is specified, in which case leading zeros are produced as necessary. If the number of characters produced by an output conversion exceeds the field width, asterisks are inserted throughout the field.

Complex data items are converted on input or output as two independent floating-point quantities. The format specification for a complex data item uses two edit descriptors. For example, the statements

```
      COMPLEX A
      WRITE (6, 10) A
   10 FORMAT (F7.2, E8.2)
```

write the real part of A according to F7.2 format and the imaginary part according to E8.2 format.

Different types of data can be read by the same FORMAT specification. For example, the statement

```
   10 FORMAT (I5, F15.2)
```

specifies two conversions: the first of type integer, and the second of type real.

The statements

```
      CHARACTER R*4
      READ (5,15) NO, NONE, INK, A, B, R
   15 FORMAT (3I5, 2F7.2, A4)
```

read three integer values, two real values, and one character string.

Repeatable Edit Descriptors

Certain edit descriptors can be repeated by prefixing the descriptor with a nonzero unsigned integer constant specifying the number of repetitions required. The repeatable edit descriptors are A, D, E, F, G, I, L, O, R, and Z. The other edit descriptors cannot be repeated. The repeatable edit descriptors correspond to iolist items, and cause conversion of data between internal representation and coded format. For example, the following statements are equivalent:

```
100 FORMAT (3I4,2E7.3)

100 FORMAT (I4, I4, I4, E7.3, E7.3)
```

A group of descriptors can be repeated by enclosing the group in parentheses and prefixing it with the repetition factor. If no integer precedes the left parenthesis, the repetition factor is 1. For example, the statement

```
1   FORMAT (I3, 2(E15.3, F6.1, 2I4))
```

is equivalent to the following specification (provided that the number of items in the input/output list does not exceed the number of repeatable edit descriptors):

```
1   FORMAT (I3, E15.3, F6.1, I4, I4, E15.3, F6.1, I4, I4)
```

A maximum of nine levels of parentheses is allowed in addition to the parentheses required by the FORMAT statement.

If there are fewer items in the input/output list than indicated by the format conversions in the format specification, the excess conversions are ignored.

If the total number of items in the input/output list exceeds the number of format conversions encountered when the final right parenthesis in the FORMAT specification is reached, the converted items are transmitted and a new record is read or written. The format specification is then scanned to the left for a right parenthesis. If none is found, the scan stops when the beginning of the format specification is reached. If a right parenthesis is found, however, the scan continues to the left until the field separator (slash, comma, or colon) which precedes the left parenthesis pairing the right parenthesis is reached. Transmission resumes with formatting proceeding to the right until either the output list is exhausted or the final right parenthesis of the FORMAT statement is encountered. For example, in the sequence

```
      READ (5, 100) I, A, J, B, K, C, L
  100 FORMAT (I7, (F12.7, I3))
```

I is input with format I7, A is input with F12.7, and J is input with I3. The format specification is exhausted (the right parenthesis has been reached); a new record is read, and the specification (F12.7,I3) is rescanned. B is input with format F12.7, K with I3, and from a third record, C with F12.7, and L with I3.

A repetition factor can be used to indicate multiple end-of-record slashes; the specification n(/) causes n-1 lines to be skipped on output. For example, the statement

```
2   FORMAT (' VALUES',4(/),' X   Y')
```

causes the following record to be output:

```
VALUES
(blank line)
(blank line)
(blank line)
X   Y
```

Following are descriptions of the repeatable edit descriptors.

A Descriptor

The A descriptor can be used with an input/output list item of type character or noncharacter. This descriptor has the forms:

A (Character data only) Aw

Input

If w is less than the length of the list item, the input quantity is stored left-justified in the item; the remainder of the item is filled with blanks. If w is greater than the length of the item, the rightmost characters are stored and the remaining characters are ignored.

If w is omitted, the length of the field is equal to the length of the list item; in this case you cannot use A with a noncharacter list item. If w is omitted and the list item is a character array name, then the entire array is read; the length of the field is the length of an array element.

Following are some examples of A input.

Example:

```
      CHARACTER A*9
      READ (5, 100) A
100 FORMAT (A7)
```

Input Record:

```
EXAMPLE
```

In Location A:

```
EXAMPLE△△
```

Example:

```
      CHARACTER B*10
      READ (5, 200) B
200 FORMAT (A13)
```

Input Record:

```
SPECIFICATION
```

In Location B:

```
CIFICATION
```

Example:

```
      CHARACTER Q*8, P*12, R*9
      READ (5, 10) Q, P, R
10    FORMAT (A8, A12, A5)
```

Input Record:

```
THISΔISΔI ANΔEXAMPLEΔI I ΔKNOW
```

(Bars mark input fields as specified by the FORMAT statement.)

In Storage:

```
P: THISΔISΔ
Q: ANΔEXAMPLEΔI
R: ΔKNOWΔΔΔΔ
```

Output

If w is less than the length of the list item, the leftmost characters in the item are output. For example, the statements

```
CHARACTER A*6
A = 'SAMPLE'
PRINT '(1X, A4)', A
```

print the following:

```
SAMP
```

If w is greater than the length of the list item, the characters are output right-justified in the field, with blanks on the left. For example, if the format specification in the preceding example is changed to (1X,A12), output is as follows:

```
ΔΔΔΔΔΔSAMPLE
```

If w is omitted, the length of the output field is the same as the length of the character list item.

A Descriptor for Noncharacter List Items

The A descriptor, when used for noncharacter list items, has the form:

Aw

When the A descriptor is used with a noncharacter list item, the field width specifier, w, must appear; w characters are converted.

On input, if w is less than or equal to 8 (for real, integer, logical, or boolean list items) or 16 (for double precision or complex list items), the w characters of the input value are converted to character code and stored left-justified in the item with blank fill on the right. If w is greater than 8 (for real, integer, logical, or boolean list items) or 16 (for double precision or complex list items), the rightmost 8 or 16 characters of the input value are converted and stored.

Example:

```
      READ (1,99) X, Y, Z
99    FORMAT (3A6)
```

Input record:

```
123.4Δ I 586.25 I Δ1.E04
```

(Bars mark input fields as specified by the FORMAT statement.)

In storage:

```
X: 123.4ΔΔΔ
Y: 586.25ΔΔ
Z: Δ1.E04ΔΔ
```

A left-justified character string is stored in each of the variables X, Y, and Z.

On output, if w is less than or equal to 8 (for real, integer, logical, or boolean list items) or 16 (for double precision or complex list items), the internal value is treated as a string of character coded data, and the leftmost w characters are written to the output record. If w is greater than 8 (for real, integer, logical, or boolean list items) or 16 (for double precision or complex list items), the output value is right-justified in the field and preceded by blanks.

D Descriptor

The D descriptor specifies conversion between an internal double precision real number and an external floating-point number written with an exponent. This descriptor has the form:

Dw.d

Input

D editing corresponds to E editing and can be used to input the same forms as E.

The subfields of a D input field are the same as for E input except that you can specify either D or E for the exponent.

Output

Type D conversion is used to output double precision values. D conversion corresponds to E conversion except that D replaces E at the beginning of the exponent subfield. If the value being converted is indefinite, an I is printed in the field; if it is infinite (out-of-range), an R is printed.

The specification Dw.d produces output in the following format:

s.aD+ee

s.a+eee

- s — Minus sign if the number is negative, or omitted if the number is positive (subject to control by S, SS, and SP descriptors)
- a — One or more most significant digits
- ee — Two digit exponent
- eee — Three digit exponent

The first form is used for values where the magnitude of the exponent is less than 100; the second form is used for numbers where the magnitude of the exponent is in the range 100 through 999.

E Descriptor

The E descriptor specifies conversion between an internal real or double precision value and an external number written with an exponent. This descriptor has the forms:

Ew.d Ew.dEe

Input

An E input field consists of an integer subfield, a fraction subfield, and an exponent subfield.

The integer subfield begins with a + or - sign, a digit, or a blank; and it can contain a string of digits. The integer subfield is terminated by a decimal point, E, +, - or the end of the input field.

The fraction subfield begins with a decimal point and terminates with an E, +, - or the end of the input field. It can contain a string of digits.

The exponent subfield can begin with E (or e), +, or -. When it begins with E, the + is optional between E and the string of digits in the subfield. For example, the following are valid equivalent forms for the exponent 3:

E+ 03 e 03 e03 E3 +3

A nonblank input field must contain an integer subfield or a fraction subfield, or both, and may contain an exponent subfield. An input field containing only an exponent subfield is diagnosed as a fatal error. An input field consisting entirely of blanks is interpreted as zero.

The range, in absolute value, of permissible values is approximately 10**(-1234) through 10**1232. Numbers below the range are treated as zero; numbers above the range cause a fatal error message.

Examples of valid subfield combinations are as follows:

+1.6327E-04	Integer-fraction-exponent
-32.7216	Integer-fraction
+328+5	Integer-exponent
.629E-1	Fraction-exponent
+136	Integer only
136	Integer only
.07628431	Fraction only

The width w of an E descriptor includes plus or minus signs, digits, decimal point, E, and exponent. If an external decimal point is not provided, d acts as a negative power-of-10 scaling factor. The internal representation of the input quantity is given by

i * 10**(-d) * 10**p

where i is the integer subfield and p is the exponent subfield.

For example, if the specification is E10.8, the input number 3267E+05 is converted and stored as:

3267 * 10**5 * 10**(-8) = 3.267.

If an external decimal point is provided, it overrides d; e, if specified, has no effect on input.

If the field length specified by w in Ew.d is not the same as the length of the field containing the input number, incorrect numbers might be read, converted, and stored. The following example illustrates a situation where numbers are read incorrectly, converted, and stored; yet there is no immediate indication that an error has occurred:

```
      OPEN (3, BLANK='ZERO')
      READ (3, 20) A, B, C
20    FORMAT (E9.3, E7.2, E10.3)
```

Input Record (three adjacent fields in positions 1 through 24):

```
+6.47E-01 | -2.36 | +5.321E+02
```

(Bars mark programmer's intended values.)

Numbers actually read:

```
+6.47E-01 | -2.36+5 | .321E+02
```

First, +647E-01 is read, converted and placed in location A. The second specification E7.2 exceeds the width of the second field by two characters. The number -2.36+5 is read instead of -2.36. The specification error (E7.2 instead of E5.2) caused the two extra characters to be read. The number read (-2.36+5) is a legitimate input number. Since the second specification incorrectly took two digits from the third number, the specification for the third number is now incorrect. The field .321E+02 is read. The OPEN statement specifies that trailing blanks are to be treated as zeros; therefore the number .321E+0200 is read converted and placed in location C. Here again, this is a legitimate input number which is converted and stored, even though it is not the number desired.

Following are some additional examples of Ew.d input.

Input Field	Specification	Converted Value	Remarks
+143.26E-03	E11.2	0.14326	All subfields present.
327.625	E7.2	327.625	No exponent subfield.
-.0003627+5	E11.7	-36.27	Integer subfield only a minus sign and a plus sign appears instead of E.
-.0003627E5	E11.7	-36.27	Integer subfield left of decimal contains minus sign only.
blanks	E4.1	0.	All subfields empty.

Output

The width w must be sufficient to contain digits, plus or minus signs, decimal point, E, the exponent, and blanks. Generally, the following values for w are sufficient:

$w \geq d + 6$ or $w \geq d + e + 4$ for negative numbers or positive numbers under SP control.

$w \geq d + 5$ or $w \geq d + e + 3$ for positive numbers not under SP control.

Positive numbers need not reserve a space for the sign of the number unless an SP specification is in effect. If the field is not wide enough to contain the output value, asterisks are inserted throughout the field. If the field is longer than the output value, the quantity is right-justified with blanks on the left. If the value being converted is indefinite, an I is printed in the field; if it is infinite (out-of-range), an R is printed.

The Ew.d specification produces output in the following formats:

s.aE±ee

s.a±eee

s	Minus sign if the number is negative; omitted if the number is positive
a	One or more most significant digits of the value correctly rounded
ee	Two-digit exponent
eee	Three-digit exponent

The first form is used for values where the magnitude of the exponent is less than 100; the second form is used for values where the magnitude of the exponent is in the range 100 through 999. For values where the magnitude of the exponent exceeds 999, you must explicitly specify the exponent length (at least four digits) using the form Ew.dEe.

When the specification Ew.dEe is used, the exponent is preceded by E, and the number of digits used for the exponent field not counting the letter and sign is determined by e. If value specified for e is too small for the value being output, the entire field width as specified by w will be filled with asterisks.

If an integer value is output under the Ew.d specification, results are unpredictable since the internal formats of real and integer values differ. An integer value normally does not have an exponent and will be printed, therefore, as a very small value or 0.0.

Following are some examples of Ew.d output.

Internal Value	Format Specification	Output Field
67.32	E9.3	Δ.673E+02
-67.32	E9.3	-.673E+02
67.32	E12.3	Δ Δ Δ Δ .673E+02
-67.32	E12.3	Δ Δ Δ -.673E+02

F Descriptor

The F descriptor specifies conversion between an internal real or double precision numand an external floating-point number. This descriptor has the form:

Fw.d

Input

On input, the F specification is treated identically to the E specification. Note that an exponent can appear in the input field.

Following are some examples of Fw.d input.

Input Field	Specification	Converted Value	Remarks
367.2593	F8.4	367.2593	Integer and fraction field.
.62543	F6.5	.62543	No integer subfield.
.62543	F6.2	.62543	Decimal point overrides d of Fw.d specification.
+144.15E-03	F11.2	.14415	Exponents are allowed in F input.
50000	F5.2	500.00	No fraction subfield; input number converted as 50,000x10*-2.
blanks	F5.2	0	Blanks in input field interpreted as 0.

Output

The F descriptor outputs a real number without a decimal exponent.

The plus sign is suppressed for positive numbers. If the field is too short, a field of asterisks is output. If the field is longer than required, the number is right-justified with blanks on the left. If the value being converted is indefinite, an I is printed in the field; if it is infinite (out-of-range), an R is printed.

The specification Fw.d outputs a number in the following format:

sn.n

- n Field of decimal digits
- s Minus sign if the number is negative, or omitted if the number is positive

Following are some examples of F output.

Internal Value	Format Specification	Output Field
+32.694	F6.3	32.694
+32.694	F10.3	Δ Δ Δ Δ 32.694
-32.694	F6.3	******
.32694	F4.3	.327
32.694	F6.0	Δ Δ Δ 33.

G Descriptor

The G descriptor specifies conversion between an internal real or double precision number and an external floating-point number written either with or without an exponent. This descriptor has the forms:

Gw.d Gw.dEe

Input

Input under control of the G specification is the same as for the E specification. The rules that apply to the E specification also apply to the G specification.

Output

Output under control of the G descriptor depends on the size of the floating-point number being edited. For values greater than or equal to .1 and less than 10**d the number is output under F format. For values outside this range, Gw.d output is identical to Ew.d, and Gw.dEe is identical to Ew.dEe. If the value being converted is indefinite, an I is printed in the field; if it is infinite (out-of-range), an R is printed.

If a number is output under the Gw.d specification without an exponent, four spaces are inserted to the right of the field (these spaces are reserved for the exponent field E+ee). Therefore, for output under G conversion, w must be greater than or equal to d + 6. The six extra spaces are required for sign, decimal point, and four-space exponent field. If the Gw.dEe form is used for a number output without an exponent, then e + 2 spaces are inserted to the right of the field.

Following are some examples of G output.

Internal Value	G14.8 Output Field	Format Option
.001415926535	.14159265E-02	E conversion
.8979323846	.89793238	F conversion
2643383279.	.26433833E+10	E conversion
-693.9937510	-693.99375	F conversion

I Descriptor

The I descriptor specifies integer conversion. This descriptor has the forms:

Iw **Iw.m**

Input

You can omit the plus sign for positive integers. When a sign appears, it must precede the first digit in the field. An Iw.m specification has no effect on input. The interpretation of blanks within a field depends on the BLANK= specifier in the OPEN statement (described later in this chapter). If this specifier is omitted, blanks are ignored. An all-blank field is always considered to be zero. Decimal points are not permitted. Any character other than a decimal digit, blank, or the leading plus or minus sign in an integer field on input will cause an error.

The following example illustrates Iw input. This example assumes that blanks are ignored (BLANK= specifier omitted from OPEN statement, or BLANK='NULL' specified). If BLANK='ZERO' were specified, blanks would be interpreted as zeros. In this case, J would contain -1500 and N would contain 104. The other values would not be affected.

```
      OPEN (2, BLANK='NULL')
      READ (2, 10) I, J, K, L, M, N
10    FORMAT (I3, I7, I1, I3, I1, I4)
```

Input Record:

139|ΔΔ-15ΔΔ|18|ΔΔ7|ΔΔ|1Δ4

(Bars mark input fields.)

In Storage:

I:	139	L:	7
J:	-15	M:	0
k:	18	N:	14

Output

If the integer is positive, the plus sign is suppressed unless an SP specification (described later in this chapter) is in effect. Leading zeros are suppressed.

If Iw.m is used and the output value occupies fewer than m positions, leading zeros are generated to fill up to m digits. If m=0, a zero value will produce all blanks. If m=w, no blanks will occur in the field when the value is positive, and the field will be too short for any negative value. If the field is too short, asterisks occupy the field.

Following are some examples of I output. Note that the first character of a printer output record is used for printer control and is not printed. More information on printer control appears later in this chapter.

Example:

```
      PRINT 10, I, J, K
10    FORMAT (I9, I10, I5.3)
```

In Storage:

I: -3762
J: 4762937
K: 13

Printed Output:

ΔΔΔ-3762|ΔΔΔ4762937|ΔΔ013

(Bars mark output fields.)

The first character in a printed output line is always interpreted as a printer control character and does not appear in the output.

Example:

```
      WRITE (6, 100) N, M, J
100   FORMAT (I5, I6, I9)
```

In Storage:

N: 20
M: -731450
I: 205

Printed Output:

```
ΔΔ20|******|ΔΔΔΔΔΔ205
```

(Bars mark output fields.)

Asterisks were printed in the second output field because the specification I6 was too small.

L Descriptor

The L descriptor is used to input or output logical data items. This descriptor has the form:

Lw

Input

If the first nonblank characters in the field are T or .T, the logical value true is stored in the corresponding list item, which should be of type logical. If the first nonblank characters are F or .F, the value false is stored. If the first nonblank characters are not T, .T, F, or .F, a diagnostic is printed. In CDC FORTRAN, an all blank field has the value false.

Output

Variables output under the L specification should be of type logical. A value of true or false in memory is output as a right-justified T or F with blanks on the left.

Example:

```
      LOGICAL I, J, K
      I = .TRUE.
      J = .FALSE.
      K = .TRUE.
      PRINT 5, I, J, K
5     FORMAT (1X, 3L3)
```

These statements print the following:

```
ΔΔTΔΔFΔΔT
```

O Descriptor (CDC Extension)

The O descriptor is used to input or output items in octal format. This descriptor has the forms:

Ow **Ow.m**

The form Ow.m has the same meaning as Ow on input. The octal digits include the numbers 0 through 7.

Input

An input field corresponding to an integer, real, logical, or boolean list item can contain a maximum of 22 digits. An input field corresponding to a complex or double precision list item can contain a maximum of 43 digits. Blanks are allowed and a plus or minus sign can precede the first octal digit. Blanks are interpreted as zeros and an all blank field is interpreted as zero. A decimal point is not allowed. The digits are stored right-justified within the list item, with zero fill on the left.

If an input field corresponding to an integer, real, logical, or boolean list item contains 22 digits, the leftmost two bit positions of the input field must be zero. If an input field corresponding to a complex or double precision list item contains 43 digits, the leftmost bit position of the input field must be zero.

Example:

```
      BOOLEAN P, Q, R
      READ 10, P, Q, R
10    FORMAT (O10, O12, O2)
```

Input Record:

```
3737373737|666Δ6644Δ4441-0
```

(Bars mark input fields as specified by the FORMAT statement.)

In Storage (Octal representation):

```
P:  0000000000003737373737
Q:  0000000000666066440444
R:  0000000000000000000000
```

Output

If w is less than or equal to 22 (for integer, real, logical, or boolean list items) or 43 (for complex or double precision list items), the rightmost digits are output. The output field includes leading zeros. For example, if location P contains a value with the octal representation

```
0000000000003737373737
```

and the output statements are

```
      WRITE (6, 100) P
100   FORMAT (1X, O4)
```

then the digits 3737 are output.

If w is greater than 22 (for integer, real, logical, or boolean list items), 22 digits are output right-justified with blanks on the left. For example, if the preceding format specification is changed to (1X, O24), output is as follows:

```
ΔΔ0000000000003737373737
```

If w is greater than 43 for complex or double precision list items, 43 digits are output right-justified with blanks on the left.

A negative number is output in two's complement internal form. For example, the statements

```
      I = -11
      PRINT 200, I
200   FORMAT (1X, O22)
```

produce the following output:

```
1777777777777777777765
```

The value of m is used to cause leading zeros to be written as blanks. If m is specified, up to w-m leading zeros are written as blanks. If the number cannot be output in w octal digits, a field of w asterisks is produced.

R Descriptor (CDC Extension)

The R descriptor is used with noncharacter list items. This descriptor is used to transmit the rightmost characters of a word. The R descriptor has the form:

Rw

On both input and output, the R specification is identical to the A specification unless w is less than 8 (for integer, real, logical, or boolean list items) or 16 (for complex or double precision list items).

On output, if w is less than 8 (for integer, real, logical or boolean list items) or 16 (for complex or double precision list items), the rightmost w characters of the output item are written to the output record.

Example of R input:

```
      BOOLEAN HOO, RAY
      READ (5, 600) HOO, RAY
600 FORMAT (R8, R7)
```

Input Record:

```
RESULTS△I OF△TEST
```

In Storage:

HOO: RESULTS△
RAY: 00F△TEST (Leftmost character is ASCII null, not ASCII zero)

Z Descriptor (CDC Extension)

The Z descriptor is used for hexadecimal conversion. This descriptor has the forms:

Zw Zw.m

The form Zw.m is meaningful for output only. Hexadecimal digits include the digits 0 through 9 and the letters A through F. A hexadecimal digit is represented by four bits.

Input

The input string can contain up to 16 hexadecimal digits (for an integer, real, logical, or boolean list item, or 32 hexadecimal digits for a complex or double precision list item). Embedded blanks are interpreted as zero and an all blank field is equivalent to zero. A plus or minus sign can precede the first digit. The string is stored right-justified within the list item, with zeros on the left.

Example of Z input:

```
INTEGER R, S
READ (10, '(Z10, Z4)') R, S
```

Input Record:

```
A309FFFFCC4△D1
```

In Storage (hexadecimal representation):

R: 000000A309FFFFCC
S: 00000000000040D1

Output

If w is less than 16 (for integer, real, logical, or boolean list items) or 32 (for complex or double precision list items), the rightmost w*4 bits are converted to hexadecimal and written. The output field includes leading zeros. For example, if location I contains

00000000000FB26C

then the output statement

```
WRITE(6, '(1X,Z3)') I
```

writes the digits 26C.

If w is greater than 16 for integer, real, logical, or boolean list items, the 16 hexadecimal digits are right-justified with blanks on the left.

If w is greater than 32 for complex or double precision list items, the 32 hexadecimal digits are right-justified with blanks on the left.

The value of m is used to cause leading zeros to be written as blanks. If m is specified, up to w-m leading zeros are written as blanks. If the number cannot be written in w hexadecimal digits, a field of asterisks is written.

Nonrepeatable Edit Descriptors

The nonrepeatable edit descriptors control aspects of formatting, but do not cause data conversions and are not associated with items in the iolist. The nonrepeatable edit descriptors are P, BN, BZ, S, SS, SP, H, ', ", X, T, TL TR, /, and :.

P Descriptor

The P descriptor has the form:

kP

k Signed or unsigned integer constant called the scale factor.

The P descriptor is used to change the position of a decimal point of a real number when it is input or output. The descriptor kP causes subsequent format specifications to be scaled by 10**k. Scale factors can precede D, E, F, and G format specifications or appear independently, as follows:

kPDw.d kPEw.d kPGw.d

kPEw.dEe kPFw.d kP

A scale factor of zero is established when each format specification is first referenced; it holds for all D, E, F, and G edit descriptors until another scale factor is encountered.

Once a scale factor is specified, it holds for all D, E, F, and G descriptors in that FORMAT specification until another scale factor is encountered. To nullify the effect of a scale factor for subsequent D, E, F, and G descriptors, you must specify a zero scale factor (0P). For example, in the format specification

```
15  FORMAT (2P,E14.3,F10.2,G16.2,0P,4F13.2)
```

the 2P scale factor applies to the E14.3 descriptor and also to the F10.2 and G16.2 descriptors. The 0P scale factor restores normal scaling (10**0 = 1) for the subsequent specification 4F13.2.

Input

For D, E, F, and G editing, provided that the number in the input field does not have an exponent, the number is divided by 10**k and stored. For example, if the input quantity 314.1592 is read under the specification 2PF8.4, the internal number is 314.1592 * 10**(-2) = 3.141592. However, if the input number contains an exponent, the scale factor is ignored.

Output

For F editing, the number in the output field is the internal number multiplied by 10**k. In the output representation, the decimal point is fixed; the number is either left-justfied or right-justified, depending on whether the scale factor is plus or minus. For example, the internal number -3.1415926536 can be represented on output under various scaled F specifications as follows:

Specification	Number Output
-1PF13.6	-.314159
F13.6	-3.141593
1PF13.6	-31.415927
3PF13.6	-3141.592654

For E and D editing, the effect of the scale factor KP is to shift the decimal point right k places and reduce the exponent by k. In addition, the scale factor controls the decimal normalization between the coefficient and the exponent as follows:

If k is less than or equal to zero, there will be exactly -k leading zeros and d + k significant digits after the decimal point.

If k is greater than zero, there will be exactly k significant digits to the left of the decimal point and d - k + 1 significant digits to the right of the decimal point.

For example, the number -3.1415926536 is represented on output under various Ew.d scaling as follows:

Specification	Number Output
-3P E20.4	-.0003E+04
-1P E20.4	-.0314E+02
E20.4	-.3142E+01
1P E20.4	-3.1416E+00
3P E20.4	-314.16E-02

For G editing, the effect of the scale factor is nullifie unless the magnitude of the number to be output is outside the range that permits effective use of F conversion (namely, unless the number is less than 10**(-1) or equal to or greater than 10**d). In these cases, the scale factor has the same effect as described for Ew.d and Dw.d scaling. For example, the number -.00031415926536 is represented on output under the indicated Gw.d scaling as follows:

Specification	Number Output
-3PG20.6	-.000314E+00
-1PG20.6	-.031416E-02
G20.6	-.314159E-03
1PG20.6	-3.141593E-04
3PG20.6	-314.1593E-06
5PG20.6	-31415.93E-08
7PG20.6	-3141593.E-10

The number -3.1415926536 would be output as -3.14159 under any of the preceding specifications, because that number is within the required range.

BN and BZ Descriptors

The BN and BZ descriptors can be used on input with the D, E, F, G, and I edit descriptors to specify the interpretation of blanks (other than leading blanks). In the absence of a BN or BZ descriptor, blanks in input fields are interpreted as zeros or are ignored, depending on the value of the BLANK= specifier currently in effect for the input/output unit. BLANK='NULL' is the default for input. If a BN descriptor is encountered in a format specification, all blank characters in succeeding numeric input fields are ignored; that is, the field is treated as if blanks had been removed, the remaining portion of the field is right-justified, and the field is padded with leading blanks. A field of all blanks has a value of zero.

If a BZ descriptor is encountered in a format specification, all blank characters in succeeding numeric input fields are interpreted as zeros.

For example, assuming BLANK = 'NULL', if the statement

```
READ (6, '(I3, BZ, I3, BN, I3)') I, J, K
```

reads the input record

```
1ΔΔ2ΔΔ3
```

then I, J, and K have the following values:

I =1 J = 200 K = 3

S, SP, SS Descriptors

The S, SP, and SS descriptors can be used on output with the D,E,F,G, and I descriptors to control the printing of plus (+) characters. S, SP and SS have no effect on input.

Normally, FORTRAN does not precede positive numbers by a plus sign on output. If an SP descriptor is encountered in a format specification, all succeeding positive numeric fields will contain the plus sign (w must be of sufficient length to include the sign). If an SS or S descriptor is encountered, the optional plus signs will not appear.

S, SP, and SS have no effect on plus signs preceding exponents, since those signs are always provided. For example, the statements

```
A = 10.5
B = 7.3
C = 26.0
WRITE (2, '(1X, F6.2, SP, F6.2, SS, F6.2)') A, B, C
```

print the following:

```
Δ10.50Δ+7.30Δ26.00
```

H Descriptor

The H descriptor is used to output strings of characters. This descriptor is not associated with a variable in the output list. The H descriptor has the form:

nHstring

n Number of characters in the string, including blanks.

string String of characters.

The H descriptor cannot be used on input.

Note that although using apostrophes to designate a character string precludes the need to count characters, the H descriptor may be more convenient if the string contains apostrophes. For example, the sequence

```
      A = 1.5
      WRITE (2, 30) A
30    FORMAT (6HΔLMAX=, F5.2)
```

writes the following output:

```
ΔLMAX=Δ1.50
```

Replacing the H descriptor in the preceding example with ' LMAX=' would produce the same output.

Quote (CDC Extension) and Apostrophe Descriptors

Character strings delimited by a pair of apostrophe (') or quote (") symbols can be used as alternate forms of the H specification for output. The paired symbols delimit the string. If the string is empty or invalidly delimited, a fatal compilation error occurs and an error message is printed. You cannot use the apostrophe and quote descriptors on input.

NOTE

Because the apostrophe descriptor is ANSI standard, it is preferred over the quote descriptor.

The following example shows how to write a character string that is continued on a second line:

```
      WRITE (6, 20)
20    FORMAT (' RESULT OF CALCULATIONS IS  '
     *'AS FOLLOWS')
```

These statements produce the following output:

```
RESULT OF CALCULATIONS IS AS FOLLOWS
```

An apostrophe or quote within a string delimited by the same symbol can be represented by two consecutive occurrences of the symbol. Alternatively, if a quote or apostrophe appears within a string, the other symbol can be used as the delimiter. The following example shows two ways of writing a character string that contains an apostrophe:

```
      PRINT 1
      PRINT 2
1     FORMAT ("ΔABC'DE")
2     FORMAT ('ΔDON''T')
```

These statements produce the following output:

```
ABC'DE
DON'T
```

X Descriptor

The X descriptor is used to skip character positions in an input line or output line. X is not associated with an item in the input/output list. The X descriptor has the form:

nX

n Number of character positions to be skipped from the current character position; n is a nonzero unsigned integer.

The specification nX indicates that transmission of the next character to or from a record is to occur at the position n characters forward from the current position. When an X specification causes control to pass over character positions on output, positions not previously filled during record generation are set to blanks; those already filled are left unchanged.

Example:

```
A = -342.743
B = 1.53190
J = 22
WRITE (6, '(1X, F9.4, 4X, F7.5, 4X, I3)') A, B, J
```

Output:

```
Δ-342.7430ΔΔΔΔ1.53190ΔΔΔΔ22
```

Example:

```
READ (3, '(F5.2, 3X, F5.2, 6X, F5.2)') R, S, T
```

Input record:

```
14.62ΔΔ$13.78ΔCOSTΔ15.97
```

In R, S, and T:

R: 14.62 S: 13.78 T: 15.97

T, TL, TR Descriptors

The T, TL, and TR descriptors provide for tabulation control. These descriptors have the forms:

Tn TLn TRn

n Nonzero unsigned decimal integer

When a Tn descriptor is encountered in a format specification, input or output control skips right or left to character position n; the next edit descriptor is then processed.

When a TLn descriptor is encountered, control skips backward (left) n character positions. If n is greater than or equal to the current position, control skips to the first position.

When a TRn descriptor is encountered, control skips forward (right) n positions.

Example:

```
    READ 40, A, B, C
40  FORMAT (T2, F5.2, TR5, F6.1, TR3, F5.2)
```

Input record:

```
Δ684.73ΔΔΔΔΔ2436.2ΔΔΔΔ89.14
```

Stored in A, B, and C:

A: 684.7 B: 2436.0 C: 89.0

Example:

```
    WRITE (31, 10)
10  FORMAT (T20, 'LABELS')
```

The preceding statements position to character position 20 of the output record and write the characters LABELS.

With a T, TR, or TL specification, the order of a list need not be the same as that of the input or output record, and the same information can be read more than once. For example, if the statement

```
READ (2, '(F5.2, TL5, F5.2)') A, B
```

reads the record

```
76.05
```

then both A and B contain 76.05.

When a T, TR, or TL specification causes control to pass over character positions on output, positions not previously filled during record generation are set to blanks; those already filled are left unchanged.

It is possible to destroy a previously formed field. For example, the statements

```
    PRINT 8
8   FORMAT (' DISASTERS', T5, 3H123)
```

print the following string:

```
DIS123ERS
```

Slash (End-of-Record) Descriptor

A slash in a format specification indicates the end of a record. When a slash is used to separate edit descriptors, a comma separator is allowed but not required. Consecutive slashes can be used and need not be separated from other elements by commas. One or more slashes can precede the first edit descriptor in a format specification, can follow the last edit descriptor, or can appear between edit descriptors.

On input, a slash specifies that further data comes from the next record. If a slash is the last descriptor, it causes an input record to be skipped.

On output, a slash causes subsequent data to be written to the next record. When a slash is the last descriptor, it causes a blank record to be written.

Example:

```
      DIMENSION B(3)
      READ (5, 100) IA, B
  100 FORMAT (I5/3E7.2)
```

These statements read two records; the first contains an integer number, and the second contains three real numbers.

Example:

```
      A = 46.3272
      WRITE (3, 11) A
   11 FORMAT (1X, 'NEW VALUE', //1X, F7.3)
```

Printed output:

```
NEW VALUE
(blank line)
Δ46.327
```

Each line corresponds to a formatted record. The second record is blank and produces the line spacing shown.

Colon (:) Descriptor

A colon (:) in a format specification terminates format control if there are no more items in the input/output list. The colon has no effect if there are more items in the input/output list. This descriptor is useful in forms where nonlist item edit descriptors follow list item edit descriptors; when the iolist is exhausted, the subsequent edit descriptors are not processed. For example, the statements

```
      A = 1.0
      B = 2.2
      C = 3.1
      D = 5.7
      PRINT 10, A, B, C, D
   10 FORMAT (4(F4.1, :, ','))
```

print the following record:

```
1.0, 2.2, 3.1, 5.7
```

In this example, format control terminates after the value of D is printed, and the last comma is not printed.

Printer Control Character

The first character of a printer output record is used for printer control and is not printed. It appears in other forms of output as data. For lines listed at a terminal, the SET_TERMINAL_ATTRIBUTE command allows you to specify whether printer control characters are to be recognized or disregarded.

The printer control characters are shown in table 6-3.

Table 6-3. Printer Control Characters

Character	Action
Blank	Space vertically one line, then print.
0	Space vertically two lines, then print.
1	Eject to the first line of the next page before printing.
+	No advance before printing; allows overprinting.
-	Space vertically three lines, then print.
Any other character	Refer to the SCL Language Definition manual.

Printer control characters are required at the beginning of every record to be printed, including new records introduced by means of a slash. Null records, such as those produced by successive slashes, do not require printer control characters. Printer control characters can be generated by any means.

For output directed to any device other than the line printer or terminal, printer control characters are not required.

Following are some examples of FORMAT statements in which the first character of each record is a printer control character:

```
10    FORMAT (1HO, F7.3, I1, G12.6)

20    FORMAT ('  ', I5, 'RESULT=', F8.4)

30    FORMAT ('1', I4, 2 (F7.3))

40    FORMAT (1X, I4, G16.8)
```

Unformatted Input/Output

Unformatted READ and WRITE statements do not use format specifications and do not convert data in any way on input or output. Instead, data is transferred as is between memory and the external device. (Since the data is in an internal form, it is generally not suitable for printing or terminal display.)

Each unformatted input/output statement transfers exactly one record. When a null record is written to a file with the RECORD_TYPE=FIXED attribute, such as an unformatted direct access file, the record is filled with the padding character before being written. If the record is less than MAXIMUM_RECORD_LENGTH, then the unused portions are also filled with the padding character.

Unformatted WRITE

The unformatted WRITE statement is used to output records in their internal form. This statement has the form:

WRITE (*UNIT=***u**, *IOSTAT=ios, ERR=sl*) *iolist*

Information is transferred from the items in iolist to the specified output unit with no format conversion. One record is created by an unformatted WRITE statement. If the list is omitted, the statement writes a null record on the output device. A null record contains no data but has all other properties of a legitimate record.

Example:

```
DIMENSION A(260), B(4000)
.
.
.
WRITE (10, ERR=16) A, B
```

The 4260 words of arrays A and B are written as one record in internal binary format on unit 10.

Unformatted READ

The unformatted READ statement transmits one record from the specified unit to the storage locations named in iolist. This statement has the form:

READ (*UNIT=***u**, *IOSTAT=ios, ERR=sl, END=sl*) *iolist*

No format specification is used, and the transmitted data is not converted. The information is transmitted from the designated file in the form in which it exists on the file without any conversion. If the number of words in the list exceeds the number of words in the record, an execution diagnostic results. If the number of locations specified in iolist is less than the number of words in the record, the excess data is ignored. If iolist is omitted, the unformatted READ skips one record.

You should specify the END= or IOSTAT= parameter to test for an end-of-file (end-of-partition or end-of-information). If neither is specified, and an end-of-file is encountered, the program terminates with a fatal error. A fatal error also occurs if you attempt to read a unit after an END= or IOSTAT= specifier has returned an end-of-file condition for that unit. Records following an end-of-file can be read by issuing a CLOSE followed by an OPEN on the file or by calling the EOF function.

Example:

```
READ (2, END=30, ERR=40) X, Y, Z
```

reads numbers directly into X, Y, and Z with no conversions.

List Directed Input/Output

List directed input/output involves the conversion of records according to compiler-defined formatting rules (without an explicit format specification). Each record consists of a list of values in a less restricted format than is used for formatted input/output. This type of input/output is particularly convenient when the exact form of data is not important.

List Directed Input

The list directed READ statement transmits data from the specified unit or the unit INPUT (if u is omitted or UNIT= * is specified) to the storage locations named in iolist. The list directed READ statement has the forms:

READ (*UNIT*=**u**, *FMT*=*, *IOSTAT=ios, ERR=sl*) *iolist*

READ *, iolist

Unlike formatted input, in which list items are in fixed-length fields, input items for list directed input are free-form with separators.

A list directed READ following a list directed READ that terminated in the middle of a record starts with the next data record.

You should specify the END= or IOSTAT= parameter to test for an end-of-file (end-of-partition or end-of-information). If neither is specified, and an end-of-file is encountered, the program terminates with a fatal error. A fatal error also occurs if you attempt to read a unit after an END= or IOSTAT= specifier has returned an end-of-file condition for that unit. Records following an end-of-file can be read by issuing a CLOSE followed by an OPEN on the file or by calling the EOF function.

Input data consists of a string of values separated by one or more blanks, or by a comma or slash, either of which can be preceded or followed by any number of blanks. Also, a line boundary, such as the end of a terminal line or the end of a card, serves as a value separator; however, a separator adjacent to a line boundary does not indicate a null value.

Embedded blanks are not allowed in input values, except in character values and between the components of complex numbers, as described below. The format of values in the input record is as follows:

Integers

Same format as for integer constants.

Real numbers

Any valid FORTRAN format for real or double precision numbers. In addition, the decimal point can be omitted; it is assumed to be to the right of the number if no exponent is specified, or between the number and the exponent.

Complex numbers

Two real values, separated by a comma, and enclosed by parentheses. The parentheses are not considered to be a separator. The decimal point can be omitted from either of the real values. Each of the real values can be preceded or followed by blanks.

Character values

A string of characters (which can include blanks) enclosed by apostrophes. An apostrophe can be represented within a string by two successive apostrophes with no intervening characters. Character values can only be read into character arrays, array elements, variables and substrings. If the string length exceeds the length of the list item, the string is truncated. If the string is shorter than the list item, the string is left-justified and remaining character positions are blank filled.

Logical values

An optional period, followed by a T or F, followed by optional characters that do not include slashes, blanks, or commas. (Note that the logical constants .TRUE. and .FALSE. are valid.)

You can input a boolean constant only if the corresponding list item is of type boolean. Boolean constants include:

Octal constants of the form O" ... ".

Hexadecimal constants of the form Z" ... ".

Hollerith constants containing one through eight characters and delimited by quotes. Constants of less than eight characters are left-justified with blank fill on the right. Strings of greater than eight characters cause an execution error.

In addition, real and integer values can be read into boolean variables.

An input item can be repeated by preceding the item by an integer repeat count and asterisk. For example, the input record

```
3*567.123
```

is equivalent to:

```
567.123,567.123,567.123
```

Blanks cannot immediately precede or follow the asterisk.

You can input a null in place of a constant when the current value of the corresponding list item is not to be changed. A null is indicated by a comma as the first character in the input string or by two commas separated by an arbitrary number of blanks. Nulls can be repeated by specifying an integer repeat count followed by an asterisk and any value separator. The next value begins immediately after a repeated null. You cannot use a null for either the real or imaginary part of a complex constant; however, a null can represent an entire complex constant.

When the value separator is a slash, the effect is the same as reading null values for the remaining input list items. The remainder of the current record is discarded.

Input values must correspond in type to variables in the input/output list. Note that the form of a real value can be the same as that of an integer value.

List Directed Output

The list directed output statements consist of a WRITE, a PRINT, and a PUNCH (CDC Extension) statement. These statements have the forms:

WRITE (*UNIT*=**u**, *FMT*=*****, *IOSTAT*=*ios*, *ERR*=*sl*) *iolist*

PRINT *, iolist

PUNCH *, iolist (CDC Extension)

PRINT outputs data to the unit OUTPUT. PUNCH outputs to the unit PUNCH.

Data in the locations specified by iolist is converted from internal format to coded format and transferred to the designated unit.

List directed output is consistent with the input; however, comma separators, null values, slashes, repeated constants, and the apostrophes used to indicate character values are not produced. For real or double precision variables with absolute values in the range of 10**(-6) through 10**9, an F format conversion is used; otherwise, output has 1PE format. For real values, up to 13 digits are output. For double precision values, up to 27 digits are output. Trailing zeros in the fractional part and leading zeros in the exponent are suppressed. Values are separated by blanks.

List directed output statements always produce a blank for printer control as the first character of the output record.

Logical values are output as T or F. Complex values are enclosed in parentheses with a comma separating the real and imaginary parts.

Boolean values are output in the form Z"n n ...", where n is a hexadecimal digit. Leading zeros are suppressed.

Example:

```
COMPLEX A
CHARACTER B*3
A = (7.,-1)
B='DOG'
C=123.45
PRINT*, A, B, C
```

These statements print the following record:

```
(7.,-1.)DOG123.45
```

Namelist Input/Output (CDC Extension)

Namelist input/output permits formatted input and output of groups of variables and arrays by using an identifying group name instead of a format specification. The values are converted according to compiler-defined formatting rules. The name is established by the NAMELIST statement.

NAMELIST Statement

This statement has the form:

NAMELIST /name/a, ..., *a* ... */name/a*, ..., *a*

name Name to be given to the namelist group; must be unique within the program unit.

a Variable or array name.

The NAMELIST statement is a nonexecutable statement that appears in the declarative portion of the program following any specification statements. The namelist group name identifies the succeeding list of variable or array names.

You must declare a namelist group name in a NAMELIST statement before using the name in an input/output statement. The group name can be declared only once, and it cannot be used for any purpose other than a namelist name in the program unit. It can appear in READ, WRITE, PRINT, and PUNCH statements in place of the format specifier. When a namelist group name is used, the iolist must be omitted from the input/output statement.

A variable or array name can belong to one or more namelist groups.

Namelist Input

Namelist input is performed by the namelist READ statement. This statement has the form:

READ name

READ (*UNIT*=**u,** *FMT*=**name,** *IOSTAT=ios, ERR=sl, END=sl*)

name Name of the namelist group to be read.

The UNIT=, FMT=, IOSTAT=, ERR=, and END= specifiers are described earlier in this chapter under Input/Output Statement Specifiers.

When a READ statement references a namelist group name, input data in namelist format is read from the designated file. If the specified group name is not found before an end-of-file, a fatal error occurs. If the file is empty an end-of-file condition results. This must be detected by an END= or IOSTAT= specifier or a fatal error will result. A subsequent read on the same file without an intervening positioning statement, CLOSE/OPEN, or EOF function test results in a fatal error.

Data read by a namelist READ statement must have the following namelist input group format:

$name item=value, ..., *item=value*, **$END**

name

Namelist group name.

item=value

One of the following:

v=c

vc*(i1:i2)*=**c**

array*(s)*=*r**c*, ..., *r*c*

carray(s)*(i1:i2)*=*r**c*, ..., *r*c*

The symbols v, vc, i1, i2, c, array, carray, s and r are as follows:

v	Variable name.
vc	Character variable name.
i1, i2	Integer constants representing the upper and lower bounds of a character substring.
c	Constant.
array	Array name.
carray	Character array name.
s	Array subscript in which each subscript expression is an integer constant; (s) is optional in the array(s) form, and required in the carray(s)(i1:i2) form. The number of subscript expressions must be equal to the number of dimensions in the array.
r	Unsigned nonzero integer repetition factor; if omitted, * must also be omitted.

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

A & can be substituted for either of the $ characters in the group. Since the input operation terminates when the second $ or & is encountered, the characters END can be omitted.

In each record of a namelist group, position one is reserved for printer control and must be left blank. Data items following $name (or &name) are read until another $ (or &) is encountered.

Data read by a single namelist READ statement must contain only names listed in the referenced namelist group. All items in the namelist group, or any subset of the group, can be input. Values are unchanged for items not input. Variables need not be in the order in which they appear in the defining NAMELIST statement.

Blanks must not appear:

Between $ (or &) and the group name

Within array names and variable names

Blanks can be used freely elsewhere.

You can use more than one record as input data in a namelist group. The first position of each input record is ignored. All input records containing data should end with a constant followed by a comma; however, the last record can be terminated by a $ (or &) without the final comma. Constants can be preceded by a repetition factor followed by an asterisk.

Constants can be integer, real, double precision, complex, logical, boolean, or character. Each constant must agree with the type of the corresponding input list item as follows:

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

An integer, real, or double precision constant can be used for an integer, real, double precision, or boolean input list item. The constant is converted to the type of the list item. A boolean constant cannot be used for a non-boolean list item.

Logical constants have the following forms (the lowercase forms are equivalent):

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

A character constant must have delimiting apostrophes. If a character constant occupies more than one record, each continuation of the constant must begin in column two; a complex constant has the form (c1,c2) where c1 and c2 are real constants. A character constant must extend to the end of a record preceding a continuation record. A boolean constant must be an octal constant, a hexadecimal constant, or a Hollerith constant delimited by quotes.

Blank characters appearing within noncharacter constants are ignored. The BLANK= specifier in an OPEN statement has no effect on namelist. If a constant other than a character constant contains no characters other than blanks, a fatal error results.

The following example illustrates a namelist input group:

```
Δ$AGRP                    Group name
ΔXVAL=5.0,                Real number
ΔARR=5*(1.7,-2.4),        Five complex numbers
ΔCHAR='HI THERE',         Character string
Δ$END                     Group terminator
```

Note that although the preceding example uses a separate input line for each variable or array definition, multiple definitions can be included on a single line.

Namelist Output

The namelist output statements consist of a WRITE statement, a PRINT statement, and a PUNCH statement. These statements have the following forms:

WRITE (*UNIT*=**u**, *FMT*=**name**, *IOSTAT*=*ios*, *ERR*=*sl*)

PRINT name

PUNCH name

name Name of the namelist group to be written.

The UNIT=, FMT=, IOSTAT=, and ERR= specifiers are described earlier in this chapter under Input/Output Statement Specifiers.

All variables and arrays and their values in the list associated with the namelist group name are output on the file associated with unit u, OUTPUT, or PUNCH. They are output in the order of specification in the NAMELIST statement. Output consists of at least three records. The first record is a $ in position 2 followed by the group name; the last record is a $ in position 2 followed by the characters END. Each group begins with triple spacing (a hyphen (-) inserted in the printer control position of each record).

No data appears in position 1 of any record of a namelist group. The maximum length of any record is 132 characters (unless a smaller maximum record length has been specified in the PROGRAM or OPEN statement). Logical constants appear as T or F. Elements of an array are output in the order in which they were stored.

Character constants are written with delimiting apostrophes. Boolean constants are written in the form Z"n ... n", where n is a hexadecimal digit; leading zeros are suppressed.

Records output by a namelist WRITE statement can be read later in the same program by a namelist READ statement specifying the same group name.

Following is an example of namelist input and output:

```
NAMELIST /INVAL/QUANT, COST
NAMELIST /OUTVAL/TOTAL, QUANT, COST
READ (*, INVAL)
TOTAL = QUANT * COST * 1.3
PRINT OUTVAL
```

Input Record:

```
Δ$inval quant=1.5, cost=3.02 $
```

Printed Output:

```
$OUTVAL
TOTAL =    .58889999999999E+01,
quant = .15E+01,
cost = .302E+01,
$END
```

The statement sequence defines two namelist groups, reads the first group, performs a simple calculation, and prints the second group.

Arrays in Namelist

Values can be read into an array by specifying, in the namelist input record, an array element followed by the values to be read, as follows:

array-element=constant, ..., *constant*

When data is input in this form, the constants are stored consecutively beginning with the location given by the array element. The number of constants can be less than or equal to, but must not exceed, the remaining number of elements in the array. For example:

```
INTEGER BAT(5)
NAMELIST /HAT/BAT, DOT
READ (*, HAT)
```

If the preceding statements read the following input record:

```
Δ$HAT  BAT=2,3,3*4,DOT=1.05 $END
```

the value of DOT becomes 1.05 and the array BAT is as follows:

BAT (1) 2
BAT (2) 3
BAT (3) 4
BAT (4) 4
BAT (5) 4

Buffer Input/Output (CDC Extension)

NOTE

This feature is included for compatibility with other versions of FORTRAN, and its use is not recommended. For guidelines, see appendix C.

Buffer input/output transmits data between memory and an external storage device without conversion. Buffer input/output differs from unformatted reading and writing in that READ and WRITE are associated with an input/output list. Buffer statements are not associated with a list; data is transmitted to or from a block of storage. Also, unlike unformatted READ operations, an attempt to buffer in more data than the record contains is not an error.

ENDFILE, REWIND, and BACKSPACE are valid for files processed by buffer statements. However, a file processed by buffer statements cannot be processed in the same program by direct access input/output statements, or by the mass storage subroutines.

Each buffer statement defines the location of the first and last words of the block of memory to or from which data is to be transmitted. The address of the last word must be greater than or equal to the address of the first word. The relative locations of the first and last word are defined only if they are the same variable or are in the same array, common block, or equivalence class. If the first and last words do not satisfy one of these relationships, their relative position is undefined and a fatal error might result at execution time.

If the first word and the last word are in the same common block but not in the same array or equivalence class, the operation will be successful but optimization might be degraded.

After execution of a buffer statement has been initiated, and before referencing the same file or any of the contents of the block of memory to or from which data is transferred, the status of the buffer operation must be checked by a reference to the UNIT function. This status check ensures that the data has actually been transferred and the buffer parameters for the file have been restored. If a second input/output operation is attempted on the same file without an intervening reference to UNIT, an error results.

BUFFER IN

The BUFFER IN statement has the form:

BUFFER IN (u, p) (a, b)

u	Unit identifier. See the description of the unit identifier under Input/Output Statement Specifiers.
p	This parameter is provided for compatibility with previous versions of FORTRAN; it must be present, but it is disregarded.
a	First variable or array element of the block of memory to which data is to be transmitted; cannot be type character.
b	Last variable or array element of the block of memory to which data is to be transmitted; cannot be type character.

BUFFER IN transfers one record from the file indicated by u to the block of memory beginning at a and ending at b. If the record is shorter than the block of memory, excess locations are not changed. If the record is longer than the block of memory, excess words in the record are ignored, except when the record type is fixed (RECORD_TYPE=FIXED on the SET_FILE_ATTRIBUTE command), in which case an error occurs.

The UNIT function can be used to test for an end-of-file condition after a BUFFER IN operation. After UNIT has been referenced, the number of words transferred to memory can be obtained by a call to the function LENGTH. If records do not terminate on a word boundary (in a file not written by BUFFER OUT), the exact length of the record is returned by LENGTHX in terms of words and excess bits.

Example:

```
DIMENSION CALC(51)
BUFFER IN (1,P) (CALC(1), CALC(51))
IF (UNIT(1) .GE. 0) GO TO 20
```

Data is transferred from logical unit 1 into storage beginning at the first word of the array, CALC(1), and extending through CALC(51). An error or endfile condition will transfer control to statement 20.

BUFFER OUT

The BUFFER OUT statement has the form:

BUFFER OUT (u, p) (a, b)

u Unit identifier. See the description of the unit identifier under Input/Output Statement Specifiers.

p This parameter is provided for compatibility with previous versions of FORTRAN; it must be present, but it is disregarded.

a First variable or array element of the block of memory from which data is to be transmitted; cannot be type character.

b Last variable or array element of the block of memory from which data is to be transmitted; cannot be type character.

BUFFER OUT writes one record by transferring the contents of the block of memory beginning at a and ending at b to the file indicated by u. The length of the record is given by:

lwa - fwa + 1

where lwa is the last word address of the record and fwa is the first word address. For fixed-length records (RECORD_TYPE=FIXED attribute), the record length is the length (number of characters) specified by the MAXIMUM_RECORD_LENGTH parameter on the SET_FILE_ATTRIBUTE command. If the specified length is greater than (lwa - fwa + 1) * 8, an error occurs.

The UNIT function must be referenced before another reference is made to the file or to the contents of the block of memory.

Mass Storage Input/Output (CDC Extension)

Mass storage input/output (MSIO) subroutines allow you to create, access, and modify files on a random basis without regard for their physical positioning. Each record in the file can be read or written at random without logically affecting the remaining file contents. The length and content of each record are determined by you. A random file can reside on any mass storage device.

A file processed by mass storage subroutines should not be processed by any other form of input/output.

Random Files

A randomly accessible file capability is provided by the mass storage input/output subroutines. Random files offer similar advantages to direct access files. In a random file, as in a direct access file, any record can be read, written, or rewritten directly, because the file resides on a random access mass storage device that can be positioned to any record of a file.

NOTE

Direct access files and the indexed sequential subprograms both offer the advantages of random access. The direct access capability is ANSI standard, but allows only fixed length records; indexed sequential and mass storage input/output allow both fixed and variable length records.

To permit random accessing, each record in a random file is uniquely and permanently identified by a record key. A key is a value that you select and include as a parameter on the call to read or write a record. When a record is first written, the key in the call becomes the permanent identifier for that record. The record can be retrieved later by a read call that specifies the same key, and it can be updated by a write call with the same key.

When a random file is in active use, the record key information is kept in an array in the program's execution field length. You are responsible for allocating the array space by a DIMENSION, type, or similar array declaration statement, but you must not attempt to manipulate the array contents. The array becomes the directory, or index, to the file contents. In addition to the key data, it contains the word address and length of each record in the file. The index is the logical link that enables the mass storage subroutines to associate a user call key with the location of the required record.

The index is maintained automatically by the mass storage subroutines. You must not alter the contents of the array containing the index in any manner, because to do so might result in destruction of the file contents. (In the case of a subindex, you must clear the array before using it as a subindex and, if an existing file is being reopened and manipulated, read the subindex into the array. However, individual index entries should not be altered.)

In response to a call to open the file, the mass storage subroutine automatically clears the assigned index array. The index array should be noncharacter. If an existing file is being reopened, the mass storage subroutines locate the master index in mass storage and read it into this array. Subsequent file manipulations make new index entries or update current entries. When the file is closed, the master index is written from the array to the mass storage device. When the file is reopened by the same job or another job, the index is again read into the index array provided, so that file manipulation can continue.

OPENMS

The OPENMS call opens a mass storage file and informs the system that it is a random file. This call has the form:

CALL OPENMS (u, ix, len, t)

u Unit identifier.

ix Array to contain the master index.

len Integer expression specifying the length of the master index. For a number index:

len > (number of entries) + 1

For a name index:

len > 2*(number of entries) + 1

t Integer expression specifying index type; can have one of the following values:

0 File has a number master index.
1 File has a name master index.

If a file having the name derived from u does not already exist, a new file is created.

The array ix specified in the call is automatically cleared to zeros. If an existing file is being reopened, the master index is read from mass storage into the index array.

Example:

```
DIMENSION A(11)
CALL OPENMS (5, A, 11, 0)
```

These statements open a random file named TAPE5 using an 11-word (10-entry) master index of the number type. If the file already exists, the master index is read into memory starting at location A.

WRITMS

The WRITMS call transmits data from memory to a random file. This call has the form:

CALL WRITMS (u, arr, n, k, r, s)

u Unit identifier.

arr Name of array from which data is to be written.

n Integer expression specifying the number of consecutive words to be written.

k Record key. For number index, k can be any arithmetic expression whose value is:

1 < k < len-1

For name index, k can be any boolean or integer expression. If k is an integer expression, the value BOOL(k) is used.

r Rewrite specifier; integer expression having one of the following values:

-1 Rewrite in place if new record length does not exceed old record length; otherwise, write at end-of-data.

0 No rewrite; write at end-of-data (default value).

1 Unconditional rewrite in place. A fatal error occurs if new record length exceeds old record length.

s Subindex flag specifier; integer expression having one of the following values:

Do not write subindex marker flag in index control word (default value).

1 Write subindex marker flag in index control word for this record.

The end-of-data (for r = -1 and r = 0) is defined to be immediately after the end of the data record which is closest to end-of-information. A random file record can be written in place, in which case it replaces an existing record, or it can be written at end-of-data.

The r parameter can be omitted if the s parameter is also omitted. The s parameter marks a subindex record so that it can be distinguished from a data record.

Example:

```
CALL WRITMS (3, DATA, 25, 6, 1)
```

This statement unconditionally rewrites in place a 25-word record, having an index number key of 6, from array DATA to file TAPE3. The default value is taken for the s parameter.

READMS

The READMS call transmits data from the specified random file to memory. This statement has the form:

CALL READMS (u, arr, n, k)

u Unit identifier.

arr Name of array into which record is to be read.

n Integer expression specifying the number of words to be read. If n is less than the record length, n words are read and no message is issued.

k Record key specifying the record to be read. Specified key must match one of the keys defined in a WRITMS call.

Example:

```
CALL READMS (3, MORDAT, 25, 2)
```

This statement reads the first 25 words of record 2 from unit 3 (TAPE3) into memory starting at the first word of the array MORDAT.

CLOSMS

The CLOSMS call writes the master index from memory to the file and closes the file. This call has the form:

CALL CLOSMS (u)

u Unit identifier

If CLOSMS is not explicitly called, an automatic CLOSMS occurs upon program termination for each random file opened by the program. However, CLOSMS enables you to close a file before the end of a run in order to associate a different file with the same unit.

Since new data records can overwrite the old master index, a file that has had new data records added is invalid unless the file is closed.

Example:

```
CALL CLOSMS (L"AFILE")
```

This statement closes the file AFILE.

STINDX

STINDX selects an array, other than the one specified in the OPENMS call, to be used as the current index to the file. This call has the form:

CALL STINDX (u, ixarr, len, t)

u Unit identifier.

ixarr Noncharacter array to contain the subindex.

len Integer expression specifying the length, in words, of the subindex. For a number index:

len > number of entries + 1

For a name index:

len > 2*(number of entries) + 1

t Integer expression specifying the type of index. Must have one of the following values:

0 Number index.
1 Name index.

The call permits a file to be manipulated with more than one index. For example, when you wish to use a subindex instead of the master index, STINDX is called to select the subindex as the current index. The STINDX call does not cause the subindex to be read or written; that task must be performed by explicit READMS or WRITMS calls. STINDX merely updates the internal description of the current index to the file.

Example:

```
DIMENSION SUBIX (10)
CALL STINDX (3, SUBIX, 10, 0)
```

These statements select a new index, SUBIX, for file TAPE3 with an index length of 10 (up to nine entries). The records referenced via this subindex use number keys.

Example:

```
DIMENSION MASTER (5)
CALL STINDX (2, MASTER, 5)
```

These statements select a new index, MASTER, for file TAPE2 with an index length of 5 and index type unchanged from the last index used.

Index Key Types

There are two types of index key: name and number. A name key can be specified as any nonzero boolean or integer expression. A number key must be expressed as an integer expression with a value greater than 0 and less than or equal to the length of the index in words, minus 1 word. You select the type of key by the t parameter of the OPENMS call. The key type selection is permanent. There is no way to change the key type, because of differences in the internal index structure. If you attempt to reopen an existing file with an incorrect index type parameter, a fatal error occurs. (This does not apply to subindexes chosen by STINDX calls; it is your responsibility to ensure correct index type specification in a STINDX call.) In addition, key types cannot be mixed within an index. Violation of this restriction might result in destruction of a file.

The choice between name and number keys is left entirely to you. The nature of the application may clearly dictate one type or the other. However, where possible, the number key type is preferable, because the program will execute faster and require less storage. Faster execution occurs because it is not necessary for the I/O routines to search the index for a matching key entry (as is necessary when a name key is used). Space is saved because of the smaller index array length requirement.

Master Index

The master index type for a given file is selected by the t parameter in the OPENMS call when the index is created. The type cannot be changed after the file is created; attempts to do so by reopening the file with the opposite type index are treated as fatal errors.

Subindex

The subindex type can be specified independently for each subindex. A different subindex name/number type can be specified by including the t parameter in the STINDX call. If t is omitted, the index type remains the same as the current index. Intervening calls which omit the t parameter do not change the most recent explicit type specification. The type remains in effect until changed by another STINDX call.

STINDX cannot change the type of an index that already exists on a file. You must ensure that the t parameter in a call to an existing index agrees with the type of the index in the file. Correct subindex type specification is your responsibility; no error message is issued for an incorrect index type specification.

Multilevel File Indexing

When a file is opened by an OPENMS call, the array selected as the index area is cleared. If the call references an existing file, the file index is located and read into the array. This initializes the master index.

You can create additional indexes (subindexes) by allocating additional index array areas, preparing the areas for use as described below, and calling the STINDX subroutine to indicate to the internal routines the location, length and type of the subindex array. This process can be chained as many times as required. (Each active subindex requires a separate index array area.) The mass storage routines use the subindex just as they use the master index; no distinction is made.

A separate array space must be declared for each subindex that will be in active use. Inactive subindexes can be stored in the random file as additional data records.

The subindex is read from and written to the file by the standard READMS and WRITMS calls, in the same manner as any other data record. Although the master index array area is cleared by OPENMS when the file is opened, STINDX does not clear the subindex array area; therefore, you should clear the array to zeroes before calling STINDX. If an existing file is being manipulated and the subindex already exists on the file, you must read the subindex from the file into the subindex array by a call to READMS before STINDX is called. The STINDX call then directs the mass storage routines to use this subindex as the current index. The first WRITMS to an existing file using a subindex must be preceded by a call to STINDX to inform the mass storage routine where to place the index control word entry before the write takes place.

If you wish to retain the subindex, you must write it to the file after the current index designation has been changed back to the master index, or to a higher level subindex, by a call to STINDX.

Mass Storage Input/Output Examples

Following are two programs that illustrate random file operations. Program MS1 creates a random file with number index and writes 10 records to the file. Program MS2 performs the following modifications to the file created by program MS1:

Modifies record 8 and rewrites it at end-of-data

Modifies record 6 and rewrites it in place

Replaces record 2 with a longer record

Adds two new records to the end of the file

```
      PROGRAM MS1
C
      DIMENSION INDEX(11), DATA(25)
      CALL OPENMS (3, INDEX, 11, 0)
      DO 50 NKEY=1, 10
          .
          .   (Generate record in array DATA)
          .
          CALL WRITMS (3, DATA, 25, NKEY)
50    CONTINUE
      END

      PROGRAM MS2
C
C     NOTE LARGER INDEX ARRAY TO ACCOMMODATE TWO NEW RECORDS.
C
      DIMENSION INDEX(13), DATA(25), MORDAT(40)
      CALL OPENMS (3, INDEX, 13, 0)
C
C     READ RECORD 8 FROM FILE TAPE3.
      CALL READMS (3, DATA, 25, 8)
      .
      .   (Modify array DATA)
      .
C     WRITE MODIFIED ARRAY AS RECORD 8 AT END-OF-INFORMATION.
      CALL WRITMS (3, DATA, 25, 8)
C
C     READ RECORD 6
      CALL READMS (3, DATA, 25, 6)
      .
      .   (Modify array DATA)
      .
C     REWRITE MODIFIED ARRAY IN PLACE AS RECORD 6
      CALL WRITMS (3, DATA, 25, 6)
C
C     READ RECORD 2 INTO LONGER ARRAY MORDAT
      CALL READMS (3, MORDAT, 25, 2)
      .
      .   (Add 15 new words to array MORDAT)
      .
C     SPECIFY IN-PLACE REWRITE OF RECORD 2.
C     HOWEVER, BECAUSE NEW RECORD IS LONGER THAN
C     OLD RECORD, THE NEW RECORD IS WRITTEN AT
C     END-OF-DATA.
      CALL WRITMS (3, MORDAT, 40, 2,-1)
      END
```

The following program creates a random file with a name index. The key names are RECORD1, RECORD2, RECORD3, and RECORD4.

```
      PROGRAM MS3
      DIMENSION INDEX(9), A(15, 4)
      CHARACTER*7 REC1, REC2
      DATA REC1/'RECORD1'/, REC2/'RECORD2'/
C
      CALL OPENMS (7, INDEX, 9, 1)
      .
      .   (Generate data in array A)
      .
C  WRITE 4 RECORDS TO FILE TAPE7.  THE KEY
C  NAMES ARE RECORD1, RECORD2, RECORD3, AND RECORD4.
      CALL WRITMS (7, A(1,1), 15, REC1)
      CALL WRITMS (7, A(1,2), 15, REC2)
      CALL WRITMS (7, A(1,3), 15, 'RECORD3')
      CALL WRITMS (7, A(1,4), 15, 'RECORD4')
C
C  CLOSE THE FILE
      CALL CLOSMS (7)
      END
```

Finally, the following example creates a subindexed file with a number index. Program MS4 creates four subindexes and writes nine data records for each subindex, for a total of 36 records. After each set of nine records has been written, the program writes the associated subindex to the file.

```
      PROGRAM MS4
      DIMENSION MASTER(5), SUBIX(10), REC(50)
C
      CALL OPENMS (2, MASTER, 5, 0)
      DO 10 MAJOR=1, 4
C
C  CLEAR SUBINDEX AREA
           DO 20 I=1, 10
           SUBIX(I) = 0
20         CONTINUE
C
C  MAKE SUBIX THE CURRENT INDEX
      CALL STINDX (2, SUBIX, 10)
C
C  GENERATE AND WRITE 9 RECORDS
      DO 30 MINOR=1, 9
           .
           .
           .
           CALL WRITMS (2, REC, 50, MINOR)
30    CONTINUE
C
C  CHANGE CURRENT INDEX BACK TO MASTER INDEX
      CALL STINDX (2, MASTER, 5)
C
C  WRITE THE SUBINDEX TO THE FILE.
      CALL WRITMS (2, SUBIX, 10, MAJ, 0, 1)
10    CONTINUE
C
C  READ RECORD 5 INDEXED UNDER THE 2ND SUBINDEX.
      CALL READMS (2, SUBIX, 10, 2)
      CALL STINDX (2, SUBIX, 10)
      CALL READMS (2, REC, 50, 5)
      .
      .   (Manipulate the selected record as desired)
      .
      END
```

Direct Access Files

Direct access file manipulations differ from conventional sequential file manipulations. In a sequential file, records are stored in the order in which they are written and can normally be read back only in the same order. A record can be retrieved from a sequential file only by sequentially reading records until the desired record is read. Sequential operations can be slow and inconvenient in applications where the logical order of writing and of retrieving records differs. In addition, a sequential read requires a continuous awareness of the current file position and the position of the required record. To remove these limitations, the FORTRAN input/output statements provide a direct access file capability.

In a direct access file, any record can be read, written, or rewritten directly, without concern for the position or structure of the file. This is possible because the file resides on a random access mass storage device that can be positioned to any portion of a file without the need for a sequential search. Thus, the entire concept of file position does not apply to a direct access file. The notion of rewinding a direct access file is, for instance, without meaning.

An important limitation of direct access files is that those files do not use a record key index. The disk addresses of the records are calculated from the product of the record number and record length. Thus, an arbitrarily large record number could cause a file to be excessively large. In general, the record ordinal should be used as the record number (first record numbered 1, second record numbered 2, and so forth.)

To create a direct access, file you must specify an OPEN statement with the ACCESS='DIRECT' parameter and include the RECL (record length) parameter. For example,

```
OPEN (2, FILE='DAFL', ACCESS='DIRECT', RECL=120)
```

opens an unformatted file DAFL for direct access. The file is associated with unit 2 and has a record length of 120 words.

You must specify the record length of a direct access file in the OPEN statement. All records on the file have the same length. The method of calculating the record length of a direct access file is described later in this section.

Direct Access Record Length Calculation

Records in a direct access file are identified by a record number. The record number is a nonzero positive decimal integer that is assigned when the record is written. Once a record is written with a record number, it can always be accessed by referencing the same number. The order of records on a direct access file is the order of their record numbers. Records can be written, rewritten, or read by specifying the record number in a READ or WRITE statement. Records can be read or written in any order; they need not be referenced in the order of their record numbers. The number of the record to be read or written is specified in a READ or WRITE statement with the REC= specifier.

If the length of the iolist in a direct access formatted WRITE statement is less than the record length of the direct access file, the unused portion of the record is blank filled. (You can select a different padding character through system commands.) A direct access WRITE statement must not attempt to write a record longer than the record length.

You can use a direct access file for formatted or unformatted input/output. However, you cannot use list directed or namelist input/output with direct access files.

An internal file cannot be used for direct access.

Example:

```
WRITE (2, '(3E10.4)', REC=6) A, B, C
WRITE (2, '(2I4, G20.10)', REC=1) I, J, X
```

Variables A, B, and C are written to record number 6, and variables I, J, and X are written to record number 1 of the direct access file associated with unit 2.

Example:

```
      OPEN (UNIT=2, FILE='DARG', ACCESS='DIRECT', FORM='FORMATTED', RECL=72)
      DO 14 I = 10, 2, -2
      READ (2, 99, REC=I, ERR=20) (A(J), J=1, 6)
99    FORMAT (6E12.6)
      .
      .
      .
14    CONTINUE
```

Records 10, 8, 6, 4, and 2 are read from the direct access file DARG.

The record length for a formatted direct access file is specified in characters. The record length for an unformatted direct access file is specified in words. If the iolist for an unformatted WRITE contains character data, you still specify the record length to be written in words. You can determine the record length by the following rules:

1. Count each noncharacter item as eight characters except for double precision and complex items, which count as 16 characters.
2. Calculate the total number of characters in all the character items.
3. Add the lengths calculated in steps 1 and 2 to determine the record length in characters; add 7, divide the result by 8, and truncate the fractional part to determine the number of words in the record.

Example:

```
CHARACTER A*7, B*9, C*10, D*20, E*15, F*12
INTEGER IA, IB, IC, ID(5)
OPEN (5, ACCESS='DIRECT', FORM='UNFORMATTED', RECL=17)
WRITE (5, REC=1) A, B, IA, C, IB, E, D, ID, F
```

The length of the output record is calculated as follows:

Length of noncharacter items:

length of IA = 8
length of IB = 8 characters
length of ID = 40 characters

Total length of noncharacter items = 56 characters.

Length of character items:

length of A = 7 characters
length of B = 9 characters
length of C = 10 characters
length of D = 20 characters
length of E = 15 characters
length of F = 12 characters

Total length of character items is 73 characters.

Record length in words = (56 + 73 + 7)/8 = 17 words.

Internal Input/Output

Internal input/output is performed on internal files. Internal files provide a means of reformatting and transferring data from one area of memory to another. Input and output on internal files are performed by formatted READ and WRITE statements and the ENCODE and DECODE statements. However, no input/output devices are involved. Internal files allow data to be reformatted without the necessity of physically writing it and rereading it under a different format specification. Internal files also allow numeric conversion to or from character data type. The two types of internal files are standard internal files and extended internal files. Standard internal files are preferred over extended internal files because the former are ANSI standard and are not dependent on the length of a computer word.

Standard Internal Files

A standard internal file can be any character variable, array, or substring. The variable, array, or substring is considered an internal file when it is referenced as an internal file in a READ or WRITE statement. If the file is a variable or substring, it consists of a single record whose length is the length of the variable or substring. If the file is an array, each array element constitutes a single record. For example, the declarative statement

```
CHARACTER*20 A(100)
```

defines a character array containing 100 20-character elements. This array can be referenced as an internal file named A, containing 100 records of 20 characters each.

Records of an internal file are defined by storing data into the records, either with an output statement or an assignment statement.

The following restrictions apply to internal files:

You cannot declare internal files in PROGRAM or OPEN statements.

You can use only formatted input/output; unformatted, list directed, namelist, mass storage, and buffer input/output are not valid for internal files.

You cannot use file positioning and file status statements cannot be used with internal files.

Internal WRITE

Data is written to standard internal files using a formatted WRITE statement in which the internal unit specifier u is a character variable, array, or substring name. The WRITE statement transmits data from the variables specified in iolist to consecutive locations starting with the leftmost character of the location specified by u; data is converted from internal to character format according to the format specification. The number of characters transmitted is determined by the record length.

The following examples illustrate internal files used for output.

Example:

```
INTEGER A, B, C, D
CHARACTER*4 AR(4)
.
.
.
A = 123
B = -27
C = 104
D = 1234
WRITE (AR, '(I4)') A, B, C, D
```

In Array A After the WRITE:

AR(1): 123 AR(2): -27 AR(3): 104 AR(4): 1234

The WRITE statement defines an internal file, AR, and writes four records to the file.

Example:

```
CHARACTER*8 FMT
DATA FMT /'( F8.2)'/
.
.
.
WRITE (FMT(2:3), '(I1)') N
READ (1, FMT) (A(I), I=1, N)
```

This example builds a format specification at execution time. The programmer wishes to read an N-element array according to format F8.2, and to specify the repeat count at execution time. The internal WRITE statement writes the variable N to the second and third positions of the string (F8.2). Thus, when READ is executed, the format specification will have the appropriate repeat count.

Internal READ

Data is read from a standard internal file using a formatted READ statement in which the internal unit identifier is a character variable, array, or substring. Data is transferred from consecutive locations starting at the first character position of u, converted under format specification, and stored in the variables specified in iolist.

Example:

```
CHARACTER*3 ZT(4), A, B, C
.
.
.
READ (ZT, '(A3)') A, B, C
```

Contents of ZT:

```
CAT|DOG|RUN|CAR
```

Read Into A, B, and C:

```
A:  CAT
B:  DOG
C:  RUN
```

The READ statement reads three records from internal file ZT and stores data into variables A, B, and C.

Example:

```
CHARACTER CN*12
.
.
.
READ (CN, '(4I3)') I, J, K, L
```

Contents of CN:

```
2ΔΔΔ56Δ4ΔΔΔ8
```

Read Into I, J, K, and L (internal integer format):

```
I:  2
J:  56
K:  4
L:  8
```

The READ statement reads the single record of internal file CN, converts the data to internal integer format, and stores the converted data into variables I, J, K, and L.

Extended Internal Files (CDC Extension)

NOTE

This feature is included for compatibility with other versions of FORTRAN; its use is not recommended. For guidelines, see appendix C.

An extended internal file can be any noncharacter variable, array, or array element. A record of an extended internal file is defined by writing the record. The record length is measured in characters. Since one word contains eight characters, the record length of an extended internal file is given by:

8*a

where a is the number of words in the record.

ENCODE

The ENCODE statement is the extended internal file output statement. This statement has the form:

ENCODE (c, fn, u) iolist

c An unsigned integer constant or variable having a value greater than zero; c specifies the number of characters to be transferred per record. The record length is calculated from c.

fn The label of a FORMAT statement, or a character expression whose value is a format specification; fn must not specify namelist or list directed formatting.

u Identifies the extended internal file in which the record is to be encoded; u is a noncharacter variable, array element, or array name.

iolist A list of noncharacter variables, arrays, or array elements to be transmitted to the extended internal file identified by u.

ENCODE is similar to an internal file formatted WRITE. Values are transferred to the receiving storage area from the variables specified in iolist under the specified format. The first record starts with the leftmost character of the location specified by u. The length in characters of each record is given by:

INT((c+7)/8)*8

where INT(a) is the largest integer less than or equal to a. If c is less than the record length, the remainder of the word is blank filled.

The internal file must be large enough to contain the total number of characters transmitted by the ENCODE statement. For example, if 56 characters are generated by the ENCODE statement, the array starting at location v must be at least 56 characters (seven words) in length. If A is the receiving array the declaration BOOLEAN A(7) is sufficient. If 27 characters are generated, the declaration BOOLEAN A(4) is sufficient.

If the list and the format specification transmit more than the number of characters specified per record, an execution error message is printed. If the number of characters transmitted is less than the record length, remaining characters in the record are blank filled.

You should not encode or decode an area in memory upon itself, as the results are unpredictable.

DECODE

The DECODE statement is the extended internal file input statement. This statement has the form:

DECODE (c, fn, u) iolist

c, fn, and u are as described for ENCODE.

iolist A list of noncharacter variables, arrays, or array elements to receive data from the extended internal file identified by u.

DECODE performs a memory-to-memory transfer of data similar to an internal file formatted READ. Starting at location u, ASCII characters in memory are converted according to the specified format and stored in the variables specified in iolist.

DECODE processing of an illegal character for a given conversion specification produces a fatal error.

DECODE can be used to pack the partial contents of two words into one. Assume that two variables, LOC1 and LOC2, contain the following character values:

LOC1 SSSSΔΔΔΔ

LOC2 ΔΔΔΔDDDD

The following statements store the string 'SSSSDDDD' into the variable NAME:

```
      BOOLEAN LOC1, LOC2, TEMP, NAME
      .
      .
      .
      DECODE (8, 1, LOC2) TEMP
1     FORMAT (5X, A4)
      ENCODE (8, 2, NAME) LOC1, TEMP
2     FORMAT (2A4)
```

The DECODE statement places the last four display code characters of LOC2 into the first four characters of TEMP. The ENCODE statement packs the first four characters of LOC1 and TEMP into NAME.

File Interface Input/Output (CDC Extension)

File interface subprograms provides a random access file capability. The subprograms operate on files having indexed organization. The first part of this section describes the structure and characteristics of indexed sequential files, and presents background information required to use the file interface calls. This section then describes the subprogram calls, including formats and detailed parameter descriptions. Finally, this section describes the attributes of indexed sequential files and presents information on maintaining those files.

Indexed Sequential File Structure

An indexed sequential file is a file organization in which the file interface to FORTRAN maintains records in sorted order using a primary key. Indexed sequential files have three basic elements: keys, records, and blocks.

Records and Keys

The basic element of an indexed sequential file is a record. Records contain the data for the file. The following example shows a group of records.

51324 Miller	Robert	Driver
42976 Stevens	Carol	Manager
39048 Jetson	Harry	Asst Manager
39248 Miller	Catherine	Secretary
82176 Beirmeyer	William	Driver
75090 Arnold	Terry	Computer Operator
49257 Lane	Gladys	Accountant
38602 Johnstone	Mark	Computer Operator
13905 McGuire	Stewart	Clerk

Each record in an indexed sequential file must have a unique primary key. A primary key is a group of contiguous bytes (a byte is 8 bits). The file interface uses the primary key, which must be set before an indexed sequential file can be created, to locate records in the file. All primary key values must be different; that is, no duplicate primary key values are allowed. Records in an indexed sequential file are stored by ascending primary key value.

The following example shows the same group of records now sorted by primary key. In this example the primary key chosen was the employee number because each employee has a unique number. The primary key could not have been defined to be the employee's last name because two employees have the same last name.

primary key		
13905 McGuire	Stewart	Clerk
38602 Johnstone	Mark	Computer Operator
39048 Jetson	Harry	Asst Manager
39248 Miller	Catherine	Secretary
42976 Stevens	Carol	Manager
49257 Lane	Gladys	Accountant
51324 Miller	Robert	Driver
75090 Arnold	Terry	Computer Operator
82176 Beirmeyer	William	Driver

Record Types

Three record types are supported: variable (V), fixed (F), and undefined (U). If you choose the fixed records, every record in your file has the same length, which is determined when the file is opened for the first time. If you choose variable or undefined record type, the length of each record is determined when the record is written to the file.

Key Types

The primary key can be a series of characters or a signed binary integer, depending on which key type you select. Three key types are allowed: uncollated character (default), integer, and collated character.

Blocks

An indexed sequential file consists of an internal file label, data blocks that contain records and internal pointers to those records, and an index of primary key values grouped into index blocks.

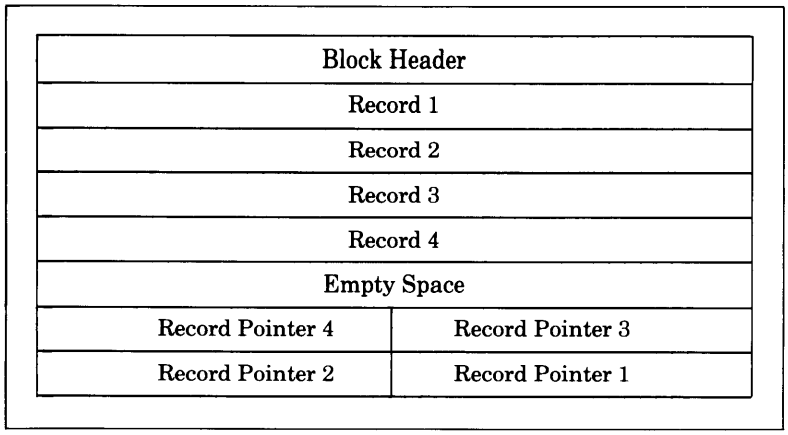
Internal File Label

The first block in all indexed sequential files is set aside for use as an internal file label. You cannot access or directly change information contained in this label.

Data Blocks

Records in an indexed sequential file are grouped into data blocks. All data blocks in the file are the same size. Each block contains a header, data records, padding, and record pointers.

The following example shows the structure of a data block that contains four records. The block header is at the beginning of the block and is immediately followed by four data records. The four record pointers are at the end of the block. Empty space (padding) between the last data record and the record pointers allow for inserting records at a later time.



Data Records

The data records are stored following the block header in ascending order of primary key values. When records are added to the file, existing records are relocated in the block so that records are always in physical and logical order for sequential access.

Record Pointers

Record pointers are stored at the end of the data block beginning with the last byte. This means that the record pointer for the first record in the data block is stored in the last few byte, the record pointer for the second record is stored in the next to the last few byte, and so forth. (See the preceding example.) If all records in a block are the same length, only one record pointer is needed; otherwise, one pointer per record is needed.

Data Block Split

If the data block cannot accommodate another record, a data block split occurs. Records in the data block up to the new record remain in the existing block. All records in the existing block which come after the new record are moved to a newly created block. The new record is put into either the new block or the existing block based on the amount of empty space in the blocks and the size of the new record. If the new record does not fit in either block, a second new block is created and the new record is put into this block.

The following example shows an example of a data block split. Part I shows data block A before record 3 is inserted. To keep the indexed sequential file in order, record 3 must be inserted between record 2 and record 4; therefore, a data block split occurs. Part II shows the result of the block split. Record 1 and record 2 remain in data block A, while record 4, and record 5 are moved to data block B. Record 3 is inserted in data block A because data block A has more empty space than data block B.

Part I:

Block Header	
Record 1	
Record 2	
Record 4	
Record 5	
Empty Space	
Record Pointer 5	Record Pointer 4
Record Pointer 2	Record Pointer 1

Data Block A

Record 3

Record 3 needs to be inserted into data block A. Therefore, a data block split occurs.

Part II:

Block Header	
Record 1	
Record 2	
Record 3	
Empty Space	
Record Pointer 3	
Record Pointer 2	Record Pointer 1

Data Block A

Block Header	
Record 4	
Record 5	
Empty Space	
Record Pointer 5	Record Pointer 4

Data Block B

Index Blocks

An index that maps the primary key of a record to the data block in which the record resides is created and maintained internally. Entries for the index are grouped into index blocks. Index blocks are the same size as data blocks.

The structure of an index block is shown below. The six index records reference the first record in each of six lower level index blocks or data blocks.

Block Header
Index Record 1
Index Record 2
Index Record 3
Index Record 4
Index Record 5
Index Record 6
Empty Space
Record Pointer

All indexed sequential files, except those consisting of a single data block, have a primary (level 0) index block. The primary index block leads to any existing lower level index blocks. A maximum of 15 levels of index blocks (numbered 0 to 14) is allowed; however, performance is usually best when no more than two index levels exist.

Index Block Header

The index block header is followed by index records, empty space (padding), and a record pointer. Only one record pointer per block is needed because the entries are always the same length.

Index Records

An index record contains the value of the primary key for the first record in the block and the physical address of the block. Index records are stored in ascending sequence of primary key values. An index record identifies all records with primary key values between its key and the key identified by the next index record. The index record references a block that is one level down in the hierarchy (up to 15 index levels are allowed). The next level can consist of either data blocks or index blocks.

Index Block Padding

Padding in index blocks is particularly significant because index blocks must contain a index record for each data block in the file. If no index padding is specified at file creation time, the addition of new data blocks can result in index block splits. The addition of new index blocks, in turn, can force the creation of a new index level which may increase the amount of time needed to access a record in the file. However, to the file, no index blocks exist. When the first data block is full and a second data block is needed, ned with initial padding that allows for more index records, the number of index levels will probably remain constant.

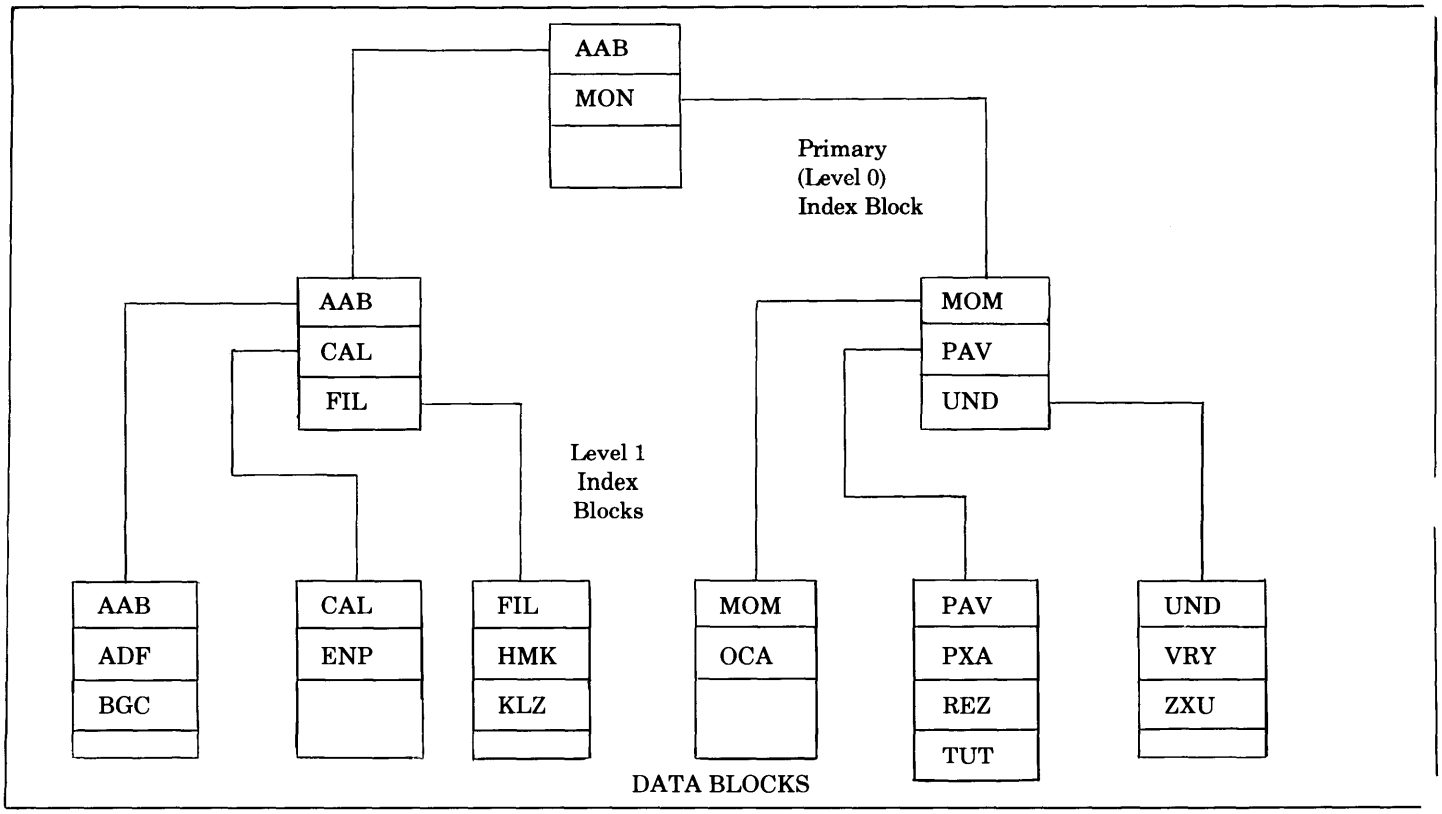
Hierarchy of Index and Data Blocks

When the first data record is written the file interface creates a primary (level 0) index block. An index record is stored in the index block as each new data block is created.

When the primary index block cannot hold any more index records, the file interface creates a second index block. The new index block and the existing index block become level 1 index blocks and contain index records for the data blocks. At the same time, a new primary index block is created to hold index records referencing the level 1 index blocks. Additional level 1 index blocks are added until the primary index block is filled and another level of indexing is needed.

The time required to retrieve a record randomly from the file is related directly to the number of index levels and the size of the blocks. Each index block accessed and searched adds to retrieval time. As long as the number of index levels remains constant, the time to retrieve a record randomly remains the same.

The following example illustrates the hierarchy of index blocks and data blocks in an indexed sequential file with two levels of index blocks. The level 0 (primary) index block contains two index records that point to the two level 1 (secondary) index blocks. In the example, each level 1 index block contains three index records that point to three data blocks.



Indexed Sequential File Processing Concepts

The file interface subprograms enable you to create new files and to process existing ones. Before using the subprograms, you should be familiar with the concepts described below. The formats of the file processing calls presented in this discussion are listed later in this chapter under File Processing Calls. An alphabetical list of the file attributes and parameters mentioned in this discussion is presented later in this section under File Attributes and Parameters.

General Concepts

The following paragraphs present concepts that apply to both creating a new indexed sequential file and processing an existing indexed sequential file.

File Attributes and Parameters

A file attribute is a characteristic of the file. File attributes describe the structure of the file and define processing limitations for the file.

File attributes can be separated into the following categories:

Preserved (structural)

Preserved attributes determine file structure; these values are kept for the lifetime of the file, although the values of some preserved attributes can be changed with the CHANGE_FILE_ATTRIBUTE (CHAFA) command.

Temporary

Temporary attributes determine how the job processes the file. These values are discarded when the file is detached.

Returned

Returned attributes are set by the operating system and can be fetched for information on the current state of the file or the last file processing operation.

The file interface to FORTRAN also has parameters that can be set by the file processing calls that includes them. The value of the parameter is stored in the file information table and remains valid until the file is closed or until the value is changed by another file processing call.

File Information Table

The file information table is an internal table that contains entries describing a file. A program must reserve one word which acts as a pointer to the table. This word appears in all file interface calls as the fit parameter. Users of Advanced Access Methods Version 2 (NOS and NOS/BE) should read appendix C before continuing.

The Working Storage Area

The working storage area is a user-defined memory location of contiguous bytes in the program where the file interface routines find or return one record for a write or read operation.

When you issue a CALL GET or CALL GETN, one record is copied from the file into a working storage area.

When you issue a CALL PUT or CALL PUTREP, the record is moved from a working storage area into the file.

Each record that is written to the file can originate in a different area of the program. Each record that is read from the file can be directed to a different area of the program.

A full record is returned for each CALL GET or CALL GETN unless the record is longer than the value of the WORKING_STORAGE_LENGTH (WSL) parameter. The number of characters in the record is always returned to you in RECORD_LENGTH (RL) parameter. Because the working storage area is defined as an integral number of bytes and the length of the record being read can be less than the working storage area length, you can use the value of RECORD_LENGTH to tell how many characters in the working storage area are part of the returned record.

Creating an Indexed Sequential File

You should submit a separate job or task to create the indexed sequential file. The general process to create an indexed sequential file is:

1. Set file attributes (SET_FILE_ATTRIBUTES and CALL FILEIS).
2. Open the file (CALL OPENM).
3. Write records to the file (CALL PUT).
4. Close the file (CALL CLOSEM).

CALL STOREF can be used to store values in temporary attributes and parameters any time after the CALL FILEIS statement as been executed.

IFETCH or CALL IFETCH can be used to retreive values of file attributes and parameters any time after the file is opened. The value of the ERROR_STATUS (ES) attribute can be fetched any time after the CALL FILEIS statement has been executed.

Setting and Fetching File Attributes and Parameters

All new indexed sequential files must have certain file attributes and parameters set so that the files can be processed. You can set file attributes and parameters in the following ways:

SET_FILE_ATTRIBUTE command (sets only attributes)

CALL FILEIS statement

CALL STOREF statement

Default values defined by the files interface

The SET_FILE_ATTRIBUTE command can override the CALL FILEIS statement, the CALL STOREF statement, and the default values.

If you have not set a file attribute that is needed for either opening the file or executing a file processing request, the file information table is checked and if a value is not found, the default value is used.

SET_FILE_ATTRIBUTE Command

The SET_FILE_ATTRIBUTE (SETFA) command sets the specified file attributes. The SETFA command can specify the initial attribute values or override the values of temporary file attributes. The SETFA command can be used in any job.

File attributes pertaining to only indexed sequential files that can be set with the SETFA command are as follows:

ACCESS_MODE
AVERAGE_RECORD_LENGTH
COLLATE_TABLE_NAME
DATA_PADDING
EMBEDDED_KEY
ERROR_EXIT_NAME
ESTIMATED_RECORD_COUNT
FILE_ORGANIZATION
FORCED_WRITE
INDEX_LEVEL
INDEX_PADDING
KEY_LENGTH
KEY_POSITION
KEY_TYPE
MAXIMUM_BLOCK_LENGTH
MAXIMUM_RECORD_LENGTH
MINIMUM_RECORD_LENGTH
OPEN_POSITION
RECORD_LIMIT
RECORD_TYPE
RECORDS_PER_BLOCK
STATUS

For a detailed discussion of the SET_FILE_ATTRIBUTE command, see the SCL System Interface manual.

Note that SCL abbreviations used in the SETFA command differ in some cases from the abbreviations used in a FORTRAN program.

CALL FILEIS Statement

You use the CALL FILEIS statement within your FORTRAN program to set file attributes.

All the attributes and parameters required for file processing can be set by the CALL FILEIS statement. Values for the attributes and parameters can be program locations (variable names), positive integers, and symbolic options in character format.

Many attributes are assigned a default value if no value is specified. You should be aware of each default value and the effect it can have on your program. (See the list of file attributes in this chapter under File Attributes and Parameters for default values.)

Values of attributes in the file information table are checked for validity and consistency at execution time. If the same value is referenced more than once, the last value specified is used.

If an unrecognizable keyword is detected in the CALL FILEIS statement, a trivial error occurs and processing continues. If a specified value is out of range for the keyword, a fatal error occurs and processing stops.

CALL STOREF Statement

The CALL STOREF statement sets or changes the value of a file attribute or parameter during program execution.

You can issue a CALL STOREF statement before or after opening the file. However, the statement must be after execution of the CALL FILEIS statement.

The following attributes cannot be set by the CALL STOREF statement after the file has been opened:

AVERAGE_RECORD_LENGTH
EMBEDDED_KEY
FILE_ORGANIZATION
KEY_LENGTH
KEY_POSITION
KEY_TYPE
MAXIMUM_RECORD_LENGTH
MINIMUM_RECORD_LENGTH
RECORD_TYPE

You can set only one attribute or parameter with each CALL STOREF statement. The value can be an integer, integer variable, or a symbolic option in character format. Integer values are retained as right-justified integers in the file information table. Symbolic values become bit strings and, depending on the bit string length, are fetched as a single bit or an integer.

Fetching File Attributes and Parameters

The values of file attributes and parameters can be examined whenever the file is opened using either the IFETCH function (IFETCH) or the IFETCH subroutine (CALL IFETCH). Your program can fetch any of the attributes or parameters listed in this chapter under File Attributes and Parameters. (The ERROR_STATUS attribute can be fetched any time after CALL FILEIS has been executed.)

For both the IFETCH function and the IFETCH subroutine, only the first eight characters are returned when fetching COLLATE_TABLE_NAME, LOCAL_FILE_NAME, or ERROR_EXIT_NAME. The characters are in left-justified FORTRAN boolean format. Fetching addresses, such as WORKING_STORAGE_AREA (WSA), is not recommended because the program cannot do anything with the returned value.

Recommendations for Setting File Attributes and Parameters

Before opening the file at creation time, you must set file attributes defining the structure of the file and processing limitations for the file. The file attributes are stored with the file in a label and are referenced automatically whenever the file is processed.

The following table lists the file attributes that the file interface needs to open a new file. The second column gives the default value for the attribute.

Attribute	Default
AVERAGE_RECORD_LENGTH (ARL)	Calculated for you
DATA_PADDING (DP)	No (zero) padding
EMBEDDED_KEY (EMK)	Kays are embedded
ESTIMATED_RECORD_COUNT (ERC)	100000 records or RECORD_LIMIT (if specified)
FILE_ORGANIZATION (FO)	You must set to indexed sequetial
INDEX_PADDING (IP)	No (zero) padding
KEY_LENGTH (KL)	You must specify; there is no default
KEY_TYPE (KT)	Uncollated keys
MAXIMUM_BLOCK_LENGTH (MBL)	Calculated for you
MINIMUM_RECORD_LENGTH (MNR)	Depends on RECORD_TYPE and EMBEDDED_KEY
MAXIMUM_RECORD_LENGTH (MRL)	You must specify; there is no default
INDEX_LEVEL (NL)	2 index Levels
RECORDS_PER_BLOCK (RPB)	2 records per block
KEY_POSITION (RKP)	0 position (first bit of first byte)
RECORD_TYPE (RT)	Undefined type records

Setting Required File Attributes

You must set the KEY_LENGTH (KL) and MAXIMUM_RECORD_LENGTH (MRL) attributes; these attributes do not have default values. If you do not set these two attributes, a fatal error occurs when the first CALL OPENM for the file is executed.

You must also set the LOCAL_FILE_NAME (LFN) attribute. If the LOCAL_FILE NAME attribute has not been specified, a fatal error occurs when you try to open the new file. After the file is created the LOCAL_FILE_NAME (LFN) attribute can be changed any time the file is closed.

If you choose collated keys, you must set the COLLATE_TABLE_NAME attribute (CTN). COLLATE_TABLE_NAME specifies the name of the collation table. You must provide your own collation table; there is no system-defined default table. See appendix H for information on creating a collation table.

Defining File Structure

The structure of an indexed sequential file is defined when the file is created. This structure cannot be changed for the life of the file. File structure is defined by the following file attributes:

AVERAGE_RECORD_LENGTH (ARL)
DATA_PADDING (DP)
ESTIMATED_RECORD_COUNT (ERC)
INDEX_LEVEL (NL)
INDEX_PADDING (IP)
KEY_LENGTH (KL)
MAXIMUM_BLOCK_LENGTH (MBL)
RECORDS_PER_BLOCK (RPB)

You should set as many of these attributes as possible; however, it is recommended that you allow the operating system to calculate values for the INDEX_LEVEL, MAXIMUM_BLOCK_LENGTH, and RECORDS_PER_BLOCK attributes.

Defining Record Type and Size

You must establish the type and size of records before the file is first opened on a creation run. Either set the RECORD_TYPE (RT) attribute to one of the three record types or accept the default of undefined type records.

The MINIMUM_RECORD_LENGTH (MNR) and MAXIMUM_RECORD_LENGTH (MRL) attributes set the minimum and maximum number of bytes in any record in the file. The value of MINIMUM_RECORD_LENGTH must be between zero and value of MAXIMUM_RECORD_LENGTH, inclusive.

If the primary key is embedded in the record, the minimum record length must include the full primary key. For the life of the file, any record that is larger or smaller than the maximum and minimum number of bytes defined for the file is rejected and a trivial error occurs.

Defining the Primary Key

The primary key is a set of contiguous bytes (a number or string of characters). For an indexed sequential file, each data record must have a unique primary key value. Records are stored in ascending order by primary key value.

You must define the location and length of the primary key at file creation time by setting the KEY_LENGTH (KL), KEY_TYPE (KT), EMBEDDED_KEY (EMK), and KEY_POSITION (RKP) attributes.

Setting KEY_TYPE

At file creation time, you must set KEY_TYPE (KT) for the primary key. Primary keys can be uncollated (U), integer (I), or collated (S).

Uncollated keys can be from 1 to 255 characters long (each character is one byte) and are compared using the ASCII collating sequence. Uncollated is the default for KEY_TYPE.

Integer keys can be from 1 to 8 bytes long (each byte is 8 bits) and are compared as signed integers.

Collated keys can be from 1 to 255 characters long (each character is one byte) and are compared using a collation table. If you specify this key type, you must supply an explicit collation table; there is no system-supplied default collation table.

You cannot change the value of the KEY_TYPE (KT) attribute after you have set it at file creation time.

Setting EMBEDDED_KEY and KEY_POSITION

At file creation time, the value for the EMBEDDED_KEY (EMK) attribute is determined. The value of this attribute specifies whether the primary keys are part of the data record (embedded) or separate (nonembedded).

The key is embedded if EMBEDDED_KEY (EMK) is set to YES and nonembedded if EMBEDDED_KEY is set to NO. The default is a file with embedded keys.

For files with embedded keys, you must also set the KEY_POSITION (RKP) attribute or accept the default of zero. The KEY_POSITION attribute is used to locate the first byte of the primary key.

Defining Block Size

Block size is set in one of two ways:

You specify the block length directly using the MAXIMUM_BLOCK_LENGTH attribute (MBL).

You accept the default block size calculated by the operating system.

The preferred method is for you to allow the operating system to calculate block size.

You specify data block size directly by defining MAXIMUM_BLOCK_LENGTH (MBL), which is the maximum number of bytes in the data block. If you specify MAXIMUM_BLOCK_LENGTH, the value is first increased, if necessary, so a block can hold at least one maximum-length record. Then, the value is rounded up to a power of 2 between 2048 and 65536 bytes, inclusive.

Alternatively, you can accept the default block size. The operating system calculates the MAXIMUM_BLOCK_LENGTH (MBL) as follows:

1. The value of the AVERAGE_RECORD_LENGTH (ARL) attribute is used as the average length of the records in the file. If AVERAGE_RECORD_LENGTH has not been set, the arithmetic mean of the MAXIMUM_RECORD_LENGTH (MRL) and MINIMUM_RECORD_LENGTH (MNR) attributes is used as the average record length. If you do not specify a value for the MINIMUM_RECORD_LENGTH attribute, zero is used.

   When the indexed sequential file contains variable length records, you should determine the value of the AVERAGE_RECORD_LENGTH (ARL) attribute as follows:

   If most records in the file are of a specific length, set AVERAGE_RECORD_LENGTH to that length.

   If record lengths are well distributed, the value of AVERAGE_RECORD_LENGTH should be set to the median record length; that is, half the records are smaller and half are larger than the value of AVERAGE_RECORD_LENGTH.

   If keys are nonembedded, AVERAGE_RECORD_LENGTH should be determined without regard to KEY_LENGTH (KL).

2. The value of the RECORDS_PER_BLOCK (RPB) attribute is used as an estimate of the number of records a data block should contain. If RECORDS_PER_BLOCK is not specified, the estimate of 2 records per block is used. This attribute has only a small effect on the calculation. RECORDS_PER_BLOCK is used only for the calculation; it is not a limit to the number of records a block can contain.

3. The value of the ESTIMATED_RECORD_COUNT (ERC) attribute is used as the estimated maximum number of records in the file. If ESTIMATED_RECORD_COUNT is not specified, the value of the RECORD_LIMIT (RL) attribute is used. If RECORD_LIMIT is not specified, the estimate of 100000 records is used.

4. The value of the INDEX_LEVEL (NL) attribute is used as the maximum number of index levels. If INDEX_LEVEL is not defined, 2 is used.

5. The operating system looks for the smallest block size that will:

   Contain RECORDS_PER_BLOCK average length records,

   and

   Contain enough index records so that when the file has grown to the estimated maximum number of data records (ESTIMATED_RECORD_COUNT), the number of index levels will be within the maximum (INDEX_LEVEL).

If you decide to let the operating system calculate MAXIMUM_BLOCK_LENGTH, you should set all or most of the attributes that are used to to calculate MAXIMUM_BLOCK_LENGTH.

A large block size is efficient in that it minimizes the number of index levels. However, in extreme cases, a large block size can reduce efficiency because a bigger block will have to be read to memory each time a record is randomly accessed. For accessing records sequentially, a bigger data block is always more efficient because fewer blocks will be read to memory.

Data and Index Block Padding

Data block padding is the unused area in a data block at file creation time. As a data block is created, data records and record pointers are stored in the block until the specified percentage of the block remains empty; then a new block is started. The new block is also filled until the specified percentage of the block remains empty. New blocks are created and filled following this process until all records have been written to the file. Data block padding loses its effectiveness when the file is closed for the first time and the reserved space is used as needed.

The degree of block fullness is not important if the creation run writes to the file only a small fraction of the number of records the file will eventually contain. However, data block padding is important if the majority of records that the file will

contain are written to the file during the creation run. For example, suppose the creation run writes one million records to the file with another 100000 records to be added eventually. If the creation run leaves blocks 10% to 15% empty and the additional records are added at random throughout the file, it is probable that the records can be inserted without causing a lot of data block splits. The cost is having the file occupy 10% to 15% more space before the additional records are added.

Data block padding should be used with care because the padding is allocated in every data block in the data file.

Data block padding should only be used if the records being inserted during the creation of the file have been already sorted by primary key. In this case, data block padding helps avoid data block splits when the file is updated.

You specify data block padding by setting the DATA_PADDING (DP) attribute to a value between 0 and 99, inclusive.

Index blocks are the same size as data blocks. Therefore, index block size is automatically defined when data block size is defined. The percentage of index block padding, however, need not be the same as for data block padding.

A small percentage of index block padding is usually recommended if a number of updates to the file is expected. If you specify index block padding at file creation time, index entries can be added without creating additional index levels.

You specify index block padding by setting the INDEX_PADDING (IP) attribute to a value between 0 and 99, inclusive.

Opening a File

You must issue an open request before issuing other file processing requests. Open processing includes the following events in the order listed:

1. File attributes specified in the open request are set.
2. Values of parameters specified in the SET_FILE_ATTRIBUTE (SETFA) command are placed in the file attributes. SETFA command values can override values previously set by the CALL FILEIS statement.
3. Minimum attributes and parameters required by the indexed sequential file organization are checked.
4. File attributes are checked for consistency in logic.
5. The file is flagged as being open.
6. For a new file, the internal file attribute table is constructed.

If a format error, omission, or logical inconsistency is detected in the file attributes or parameters, a condition code is put in the ERROR_STATUS (ES) attribute. If the MESSAGE_CONTROL (DFC) attribute has been set properly, an appropriate diagnostic is issued

Errors frequently occur when all file attributes required for the indexed sequential file organization have not been defined correctly, or when one specified file attribute precludes another specified file attribute.

An error detected during open processing prevents the file from being opened. You should check the value of the ERROR_STATUS (ES) attribute after an open request (as described later in this chapter under Error Communication).

Writing Records

A write operation causes a record to be moved from a working storage area to the file being processed. The value of the primary key determines the logical position a record occupies in the file.

On a creation run you should try to have the records already sorted in ascending order by primary key value. When presorted records are written to a new file, the result is a more efficient run in terms of file size and creation time. On subsequent runs, records to be written can be in random order.

Closing a File

When file processing is complete, you must close the file. When a file is closed, the following sequence of events occurs:

1. Blocks are written to mass storage from memory.
2. The internal file attribute tables are updated.
3. The file is flagged as being closed.

The close operation allows the file to be reopened later.

Processing an Existing File

After an indexed sequential file has been created, it can be processed for reading and/or updating. You process all existing indexed sequential files in the same manner:

1. Set file attributes (SET_FILE_ATTRIBUTE and CALL FILEIS).
2. Open the file (CALL OPENM).
3. Position the file to a specific record, if desired (CALL GET, CALL GETN, CALL REWND, CALL SKIP, or CALL STARTM).
4. Access records in the file:

   Read records (CALL GET or CALL GETN)
   Write records (CALL PUT)
   Replace records (CALL REPLC or CALL PUTREP)
   Delete records (CALL DLTE)

5. Close the file (CALL CLOSEM).

CALL FLUSHM can be used any time after the file is opened to write all blocks to mass storage. CALL FLUSHM does not change the position of the file.

CALL STOREF can be used to store values in temporary attributes and parameters any time after the CALL FILEIS statement as been executed.

IFETCH or CALL IFETCH can be used any time after the file is opened to retreive values of file attributes and parameters. The ERROR_STATUS (ES) attribute can be fetched any time after the CALL FILEIS statement has been executed.

Setting File Attributes and Parameters

When processing an existing indexed sequential file, the file interface routines reference the file information table set up when the file was created. Many of the file attributes set at creation time cannot be changed after the file is opened at creation time. These attributes are called preserved (or structural) attributes.

Temporary file attributes and parameters can be set when processing an existing file because their values are discarded when the file is detached from the job or task.

Opening a File

You must issue an open request before issuing any other file processing requests. Open processing includes the following events in the order listed:

1. File attributes specified in the open request are set. For an existing file, preserved attributes specified in the open request are verified.
2. Values of parameters specified in the SET_FILE_ATTRIBUTE (SETFA) command are placed in the file attributes. SETFA command values can override values previously set by the CALL FILEIS statement.
3. For an existing file, file attributes are extracted from the internal file attribute tables. Some of the values specified for the open can override previous file attribute values.
4. Minimum parameters required by the indexed sequential file organization are checked.
5. File attributes are checked for consistency in logic.
6. The file is flagged as being open.

If a format error, omission, orlogical inconsistency is detected in the file attributes, an error status code is put in the ERROR_STATUS attribute. If the MESSAGE_CONTROL (DFC) attribute has been set properly, an appropriate diagnostic is issued.

Errors frequently occur when not all file attributes required for the indexed sequential file organization have been defined correctly, or when one specified file attribute precludes another specified file attribute.

An error detected during open processing prevents the file from being opened. You should check the value of the ERROR_STATUS (ES) parameter. (See appendix B for information on error processing.)

Positioning an Indexed Sequential File

You can position a file at a specific record, skip records in a file, rewind a file, or position a file by major key.

Positioning to a Specific Record

An indexed sequential file can be positioned to a specific record using the CALL GET and CALL STARTM statements. In both cases, a CALL GETN can then be issued to read the file sequentially from that position.

The CALL GET statement transfers to the working storage area the record whose primary key value matches the primary key value specified in the statement. The file is positioned at the end of the record that is returned.

The CALL STARTM statement positions the file at the beginning of the record with the primary key value that satisfies the specified primary key relationship. The call can search for the first record with the primary key equal to, greater than, or greater than or equal the specified primary key value. A record is not returned to the working storage area.

Skipping Records

The CALL SKIP statement skips either forward or backward the specified number of records. The file is positioned at the beginning of the record. A record is not returned to the working storage area.

Rewinding the File

The CALL REWND statement postions the file at the start of the first record in the file (beginning-of-information). A record is not returned to the working storage area.

Positioning by Major Key

A file also can be positioned by a major key, which is a high-order portion of a primary key in an indexed sequential file. To position a file by major key, you must define the length and value of the major key to be matched against the same number of bytes at the beginning of each key in the file. The file interface routine retrieves the first record in the file with primary key matching the major key.

An indexed sequential file also can be positioned using a major key. A major key is a high-order portion of a primary key in an indexed sequential file (that is, it is a key that contains fewer bytes than the primary key of a record in the file). The major key is matched with an equal number of bytes at the beginning of a record key.

The GETN and the STARTM routines can be used to position a file by major key. Both the calls contain the MKL (MAJOR_KEY_LENGTH) parameter. When specified, the routine searches the first MKL bytes in each primary key that matches the portion of the key stored in the KEY_ADDRESS parameter.

For example, suppose the following names are primary keys for an indexed sequential file:

```
.
.
.
Meade, Steven
Meuller, Susan
Miller, Thomas
Miner, Cathy
Morgan, Jane
.
.
.
```

If the major key is defined as the letters MI, and the MAJOR_KEY_LENGTH is set to 2, execution of either the CALL GET or CALL STARTM statements positions the file at the record whose primary key is Miller, Thomas.

Accessing Records in an Existing File

After an indexed sequential file has been created, there are several different reasons to access records in the file. Records are accessed to perform any of the following operations:

Read records
Write new records
Replace records
Delete records

Reading Records

The structure of the indexed sequential file allows you to access records in the file in either of two ways: sequentially by the logical position of the record or randomly by primary key value.

If you want to read all the records in the indexed sequential file, you can use the CALL GETN statement to start with the first record and retrieve all the other records sequentially.

If you want to retrieve a single record from the indexed sequential file, you can use the CALL GET statement. In the statemnt, you specify a primary key value, and the record with a matching primary key is retrieved, if it exists.

You can also read access a contiguous group of records residing anywhere in the file. This is accomplished by first performing a random reading (CALL GET) the first record in the contiguous group and then sequentially retrieving (CALL GETN) the rest of the records.

Writing Records

New records are added to an existing file using the CALL PUT statement. Execution of the CALL PUT statement causes a record to be moved from a working storage area to the file being processed. Each new record is placed in the file according to the value of its primary key.

Records to be added do not need to be in any special order; however, execution is faster if the records already are sorted by primary key value.

If new record has the same primary key value as a record already in the file, a trivial error occurs and processing continues.

Replacing Records

Records in an indexed sequential file can be replaced using either the CALL REPLC or the CALL PUTREP statements. The new record can be larger or smaller than the record being replaced; however, the length of the new record must be within the maximum and minimum record lengths defined for the file. In both calls, you must specify the primary key in order to identify the affected record.

For the CALL REPLC statement, the file is searched for the record whose primary key matches the primary key specified in the call. If a matching primary key is found, the old record is replaced with the new record. If a record with a matching key is not found, a trivial error occurs and processing continues.

For the CALL PUTREP statement, the file is first searched for a primary key matching the primary key specified in the call. If a matching primary key is found, the old record is replaced with the new record. If a record with a matching primary key is not found, the new record is added to the file.

Deleting Records

Records in an indexed sequential file can be removed from the file using the CALL DLTE statement. In the call, you must specify the primary key in order to identify the affected record.

When the statement is executed, the data record is either flagged as logically deleted, or is physically deleted from the data block. If the first record of a data block is deleted, index blocks are updated as applicable.

Flushing a File

All blocks with changes can be written to mass storage using the CALL FLUSHM statement. After the statement has been executed, mass storage contains the current state of the file. The file remains open and is not repositioned by the request. Blocks in memory are not disturbed.

Closing a File

When file processing is complete, you must close the file. In a close operation, the following events are performed:

1. Blocks are written to mass storage from memory.
2. The internal file attribute tables are updated.
3. The file is flagged as being closed.

The close operation allows the file to be reopened later.

File Interface Calls

The FORTRAN file interface calls are described below in alphabetical order. In all file interface calls, the parameters are position dependent; therefore, parameters to the left of a desired parameter must be specified. The only required parameter is the fit parameter. If a parameter is not applicable, a zero must be specified to mark a needed position.

Each of the file interface calls is shown as a group of calls. To use the calls most effectively, you should include the group of calls in your program.

All the calls described in this section can be shortened to

CALL <request> (fit)

provided that the parameters in the call either have been set to the desired values by previous calls or default to acceptable values.

CALL CLOSEM

The last statement in a program that references an indexed sequential file must be a close request. This ensures that all updated blocks are written to mass storage.

The format of the close request is:

CALL CLOSEM (fit, *cf***)**
CALL IFETCH (fit, 'ES', status)

fit Variable name for the file information table.

cf Close flag handling:

'R' Rewind (default)

'N' No action

'U' File is detached after being closed

status Variable containing the value of error status; zero after successful operation

When the close request is executed, any data or index blocks in memory are written to the mass storage file. The internal tables (file label) that are part of the file are also updated. File statistics are written to the $ERRORS file, if requested.

You should include the CALL IFETCH statement to check whether an error occurred during the close operation.

CALL DLTE

A record in an indexed sequential file can be removed from the file through execution of a delete request. The file must be open for input/output to delete a record.

The format of the delete request is:

CALL DLTE (fit, *ka, kp, 0, ex*)
CALL IFETCH (fit, 'ES', status)

fit Variable name for the file information table.

ka Key for record to be deleted.

kp For FORTRAN 5 compatibility (see appendix E). New applications should set this parameter to zero.

ex Error exit routine; defaults to zero.

status Variable containing the value of error status; zero if operation was successful.

When deleting a series of logically contiguous fixed-length records, you can save execution time by beginning with the record having the highest primary key value. Deletion of the last record in a data block is performed quickly because the record count just needs to be reduced by one. Deletion of the first record in a data block, however, can move all remaining records in the data block. By deleting the record with the highest key value first, you can avoid relocation of records to be subsequently deleted.

When the delete request is executed, the data record is either flagged as logically deleted or physically deleted from the data block. When the first record in a data block is deleted, index blocks are updated as applicable.

If a data block or index block contains no records as a result of the delete request, it is linked into a chain of empty blocks. These blocks are reused when new blocks are required for file expansion.

The position of the file is not affected by the execution of a delete request.

If the file does not contain a record whose primary key matches the given key, a trivial error occurs.

You should include a CALL IFETCH statement to check whether an error occurred during the delete operation.

CALL FILEIS

The FILEIS call sets file attributes.

The format of the CALL FILEIS statement is:

CALL FILEIS (fit,keyword,value,...,*keyword,value*)

fit
Variable name for the file information table.

keyword

A file attribute or parameter (must be followed by an allowable value for the attribute). The name must be a character expression (for example, 'KL').

value

Value of the keyword (must be preceded by the name of an attribute or parameter). The value must be either a character expression (symbolic options), an integer representation (numeric options), or a subroutine name or variable name.

The first parameter in the CALL FILEIS statement is the name of the file information table where attribute and parameter values are stored. (As a FORTRAN programmer, you must allocate one word of memory for the name of the file information table. This is easily done using the INTEGER statement.) This variable name identifies the file information table and is the first parameter in every file interface call. When the CALL FILEIS statement is executed, the file information table is established and associated with the specified variable name.

The remaining parameters in the call are attribute and parameter abbreviations with their associated values. The pairs can appear in any order. When the CALL FILEIS statement is executed, the attribute and parameter values are stored in the file information table.

All the attributes and parameters required for file processing can be set by the CALL FILEIS statement. Values for the attributes and parameters can be program locations (variable names), positive integers, and symbolic options in character format.

Many attributes are assigned a default value if no value is specified. You should be aware of each default value and the effect it can have on your program. (See the list of file attributes under File Attributes and Parameters for default values.)

Values of attributes in the file information table are checked for validity and consistency at execution time. If the same value is referenced more than once, the last value specified is used.

If an unrecognizable keyword is detected in the CALL FILEIS statement, a trivial error occurs and processing continues. If a specified value is out of range for the keyword, a fatal error occurs and processing stops.

CALL FLUSHM

A flush request writes all blocks with changes to mass storage.

The format of the flush request is:

CALL STOREF (fit, 'EX', ex)
CALL FLUSHM (fit)
CALL IFETCH (fit, 'ES', status)

fit Variable name for the file information table.

ex Error exit routine; defaults to zero.

status Variable containing the value of error status; zero for successful operation.

After the flush request has been executed, mass storage will contain the current state of the file. The file remains open and is not repositioned by the request. Blocks in memory are not disturbed.

You should include a STOREF call to store the appropriate value in the the ERROR_EXIT_PROCEDURE (EX) attribute if you have such a procedure. The IFETCH call should also be included to check whether an error occurred during the flush operation.

CALL GET

Records in an indexed sequential file can be read randomly by primary key values. For a random read, the file must be open for either input or input/output.

The format of the random read request is:

CALL STOREF (fit, 'WSL', wsl)
CALL GET (fit, *wsa, ka, kp, mkl, 0, ex*)
CALL IFETCH (fit, 'ES', status)
CALL IFETCH (fit, 'RL', reclg)
CALL IFETCH (fit, 'FP', filpos)

fit	Variable name for the file information table.
ex	Error exit routine; defaults to zero.
filpos	Variable containing the position of the file after the read.
ka	Key for record to be read.
kp	For FORTRAN 5 compatibility (see appendix E). New applications should set this parameter to zero.
mkl	Length in bytes of the major key; defaults to zero. It is reset to zero after each GET.
reclg	Variable containing the length in bytes of the record that was read.
status	Variable containing the value of error status; zero for a successful operation.
wsa	Working storage area.
wsl	Working storage length (in bytes).

Before attempting a random read, the working storage area must be set either in the CALL GET statement using the wsa parameter or in a previous file interface call.

The WSL (WORKING_STORAGE_LENGTH) parameter is necessary for GET calls. If you have not set WSL previously and do not set it in the CALL GET statement, or have never referenced it in an IFETCH call, then the value of the MAXIMUM_RECORD_LENGTH (MRL) attribute is used.

You must also set the KA (KEY_ADDRESS) parameter and store the value of the primary key to be read into the location. Execution of the read request returns the record with the primary key matching the specified primary key value to the working storage area.

At the completion of the read request, the RECORD_LENGTH (RL) attribute is set to the actual length of the record.

You can also randomly read records by major key value. A major key is a key that contains fewer bytes than the primary key of a record in the file, and is to be matched with an equal number of bytes at the beginning of a record key. If a CALL GET statement does not include the MKL (MAJOR_KEY_LENGTH) parameter, zero is always assumed regardless of any previous value because the value of MKL is not saved.

After successful execution of the CALL GET statement, the file will be positioned at the end of the record that was returned.

An excess data error occurs if the record is longer than the value set in the WSL (WORKING_STORAGE_LENGTH) parameter. (A record longer than the length of the working storage area is prevented from overwriting a portion of the calling program or other preserved information.) Control is passed to the error exit routine, if specified, after transferring to the working storage area the number of bytes specified by the WSL parameter.

Control is passed to your end-of-data routine, if specified, by a GET request that detects end-of-information.

If the file has nonembedded keys (EMK is set to NO) and the KA (KEY_ADDRESS) parameter has been set, the primary key of the record retrieved is returned to the location indicated by the KA parameter.

If no record in the file has a matching primary key value, a trivial error occurs. The file is positioned at the point where the record should exist.

You should include the STOREF call to ensure that the appropriate value is stored in the WSL parameter. The three IFETCH calls should be included to check whether an error occurred during the read operation, and to check the size of the record that was read and the position of the file after the read operation.

CALL GETN

Records in an indexed sequential file can be read sequentially in ascending order according to the primary key values. The file must be opened for either input or input/output.

The format of the sequential read request is:

CALL STOREF (fit, 'WSL', wsl)
CALL GETN (fit, *wsa, ka, ex*)
CALL IFETCH (fit, 'ES', status)
CALL IFETCH (fit, 'RL', reclg)
CALL IFETCH (fit, 'FP', filpos)

fit	Variable name for the file information table.
ex	Error exit routine; default is zero.
filpos	Variable containing the position of the file after the read.
ka	Location where the key of the record that was read is returned.
reclg	Variable containing the length in bytes of the record that was read.
status	Variable containing the value of error status; zero for a successful operation.
wsa	Working storage area.
wsl	Working storage length (in bytes).

You must set the WSA (WORKING_STORAGE_AREA) parameter before the sequential read request can be processed. When the request is executed, the next record in primary key sequence is returned to the specified working storage area, and the file is positioned at the beginning of the next record in primary key sequence.

The WSL (WORKING_STORAGE_LENGTH) parameter is necessary for GETN calls. If you have not set WSL previously and do not set it in the CALL GETN statement, or have never referenced it in an IFETCH call, then the value of the MAXIMUM_RECORD_LENGTH (MRL) attribute is used. If the value of the wsl parameter is less than the length of the record, WSL characters are returned and a trivial error occurs.

If the KA (KEY_ADDRESS) parameter is set to zero in the CALL GETN statement, the key will not be returned to address KA as a separate character string. If the KA parameter is not specified in the statement, the current value in the KA parameter is used.

At the completion of the read request, the RECORD_LENGTH (RL) attribute is set to the actual length of the record If the sequential read request was unsuccessful, the RECORD_LENGTH (RL) attribute is not defined.

If the file is already positioned at the end of the last record and a CALL GETN statement is issued, the end-of-information status is returned and the end-of-data exit, if specified, is taken.

Execution of the CALL GETN statement repositions the file.

You should include the STOREF call to make sure the appropriate value is stored in the wsl parameter. The three IFETCH calls should be included to check that no error occurred during the read operation, and to check the size of the record that was read and the position of the file after the read operation.

CALL IFETCH

The format of the CALL IFETCH statement is:

CALL IFETCH (fit, keyword, variable)

fit

Variable name for the file information table.

keyword

Name of the attribute or parameter whose value is to be fetched. The name must be a character expression (for example, 'MRL').

variable

Value of keyword is returned in integer variable.

The following attributes can be fetched through IFETCH at any time:

ECT NL KP

The following attributes cannot be fetched unless the file is open:

DX	ES	EX	FNF	FP	KA
KP	LFN	LOP	MKL	OC	ON
REL	RKW	RL	WSA	WSL	$FI

All other attributes may be fetched provided either of the following is true:

The file is open, or

The attribute has been set by a FILEIS or STOREF call.

Note that when fetching COLLATE_TABLE_NAME, LOCAL_FILE_NAME, or ERROR_EXIT_NAME, only the first eight characters are returned. The characters are in left-justified FORTRAN boolean format.

Fetching addresses, such as WORKING_STORAGE_AREA (WSA), is not recommended because the program cannot do anything with the returned value.

CALL OPENM

You must issue an open request for an indexed sequential file before the file can be processed. Open processing gathers attributes specified by the SET_FILE_ATTRIBUTE (SETFA) command into the file label, supplies default values for file attributes you have not set, and checks file attributes for logical consistency and required attributes.

The format of the open request is:

CALL OPENM (fit, *pd, of*)
CALL IFETCH (fit, 'ES', status)

fit Variable name of the file information table (required).

pd Type of processing:

'INPUT'

File is open for read only (default).

'OUTPUT'

'I-O' or 'IO'

File is open for read and write.

'NEW'

A new file is being created; sets the ACCESS_MODE (PD) attribute to OUTPUT and the ON attribute to NEW.

of File positioning at open time (open flag):

'R' Open leaves the file positioned so that execution of a CALL GETN statement will return the first logical record of the file (default).

'E' Open leaves the file positioned so that execution of a CALL GETN statement will return end-of-information (EOI) as the file position (FP = EOI).

status Error status; zero after a successful operation.

During open processing, the file attributes and parameters are checked for errors.

If you specify NEW as the second parameter, the ACCESS_MODE (PD) attribute is set to OUTPUT and the OLD/NEW (ON) attribute to NEW. If you set the PD and ON attributes before opening the file, you need specify only the fit parameter.

You must open an existing indexed sequential file before any data records can be accessed. Values of preserved attributes are processed from internal file attribute tables during open processing. Any SET_FILE_ATTRIBUTE commands are processed the consistency of file attributes is checked.

If the file is permanent, the value set for ACCESS_MODE (PD) must be a subset of the permissions specified in the ATTACH_FILE command.

The first time an indexed sequential file is opened after the creation run, you must change the OLD/NEW (ON) attribute from NEW to OLD. You can do this by issuing a CALL FILEIS statement or a CALL STOREF statement before the file is opened. If you do not set the ON attribute by one of these statements, the attribute is set to OLD by specifying INPUT, OUTPUT, or I-O for the PD parameter in the open request.

A trivial error occurs if you attempt execution of a file processing statement and the ACCESS_MODE (PD) attribute is not set to an appropriate value for that statement. For an existing file, a fatal error occurs if the OLD/NEW (ON) attribute is not set to OLD.

Any error detected during open processing sets the ERROR_STATUS (ES) attribute. If the ERROR_EXIT_PROCEDURE (EX) attribute contains an address of your error routine, the control transfers to the routine. If the error routine corrects the condition that caused the error and executes another open request, file processing can continue. If the error routine does not correct the error and reissues the open request, the file is not opened and further file access is not allowed.

The OPEN/CLOSE (OC) attribute can be checked for information on whether or not the file was opened.

The IFETCH call should be included to check whether an error occurred during the open operation.

CALL PUT

Records are written to an indexed sequential file when a put request is executed. The file must be opened for output or input/output.

The format of the write request is:

CALL PUT (fit, *wsa, rl, ka, kp, 0, ex*)
CALL IFETCH (fit, 'ES', status)

fit	Variable name for the file information table.
ex	Error exit routine.
ka	Primary key of the new record (ignored for files with embedded keys).
kp	For FORTRAN 5 compatibility (see appendix C). New applications should set this parameter to zero.
rl	Length in bytes of record being written.
status	Variable containing the value of error status; zero after a successful operation.
wsa	Working storage area.

Prior to isssuing the put request, you must establish the record to be written in a working storage area, which is specified by the WSA (WORKING_STORAGE_AREA) parameter. If the primary key is nonembedded, you must also establish the location of the key value for the record to be written. If the primary key is embedded, the key location is determined by the KEY_POSITION (RKP) attribute.

When you put a group of contiguous records in the file, execution is usually faster if the records are sorted in ascending primary key order before you call the file interface routines. Also, the resulting file will probably be smaller.

If you present the file interface routines with unsorted records, the result is inefficient file structure with more data blocks than necessary because of numerous data block splits. Also, more time is required to create the file. (See Indexed Sequential File Structure, earlier in this section.)

If the file has nonembedded keys, the KA (KEY_ADDRESS) parameter specifies the starting address (a variable name) of the primary key. The KA parameter is ignored if the file has embedded keys.

Any errors detected during execution of the CALL PUT statement cause control to transfer to the error routine if one is specified.

You should include the IFETCH call to check whether an error occurred during the put operation.

CALL PUTREP

You can issue a put/replace request to cause a record to replace an existing record or to add a new record to the file. The file must be open for input/output.

The format of the put/replace request is:

CALL PUTREP (fit, *wsa, rl, ka, kp, 0, ex*)
CALL IFETCH (fit, 'ES', status)

fit	Variable name for the file information table.
ex	Error exit routine; default is zero.
ka	Key of record to be replaced.
kp	For FORTRAN 5 compatibility only (see appendix C). New applications should set this parameter to zero.
rl	Length in bytes of the record being replaced.
status	Variable containing the value of error status; zero after a successful operation.
wsa	Working storage area.

A put request is executed if the primary key of the new record does not match any existing primary key. A replace request is executed if a matching primary key is found in the file.

For U and V type records, the new record can be smaller or larger than the existing record; however, the new record must be within the MINIMUM_RECORD_LENGTH (MNR) and MAXIMUM_RECORD_LENGTH (MRL) defined for the file.

For F type records, the length of the new record must be equal to the MAXIMUM_RECORD_LENGTH (MRL) defined for the file.

The position of the file is not affected by the execution of a put/replace request.

You should include the IFETCH call to check whether an error occurred during the put/replace operation.

CALL REPLC

You can issue a replace request to cause a new record to replace an existing record in an indexed sequential file. The new record must have the same primary key value as the record being replaced. The file must be open for input/output.

The format of the replace request is:

CALL REPLC (fit, *wsa, rl, ka, kp, 0, ex*)
CALL IFETCH (fit, 'ES', status)

fit	Variable name for the file information table.
ex	Error exit routine; default is zero.
ka	Key of record to be replaced.
kp	For FORTRAN 5 compatibility only. (See appendix E.) New applications should set this parameter to zero.
rl	Length in bytes of new record.
status	Variable containing the value of error status; zero after a successful operation.
wsa	Working storage area.

The primary key of the new record must match an existing primary key. A trivial error occurs if the file does not contain a record with primary key matching the primary key of the new record.

For U and V type records, the new record can be smaller or larger than the existing record; however, the new record must be within the MINIMUM_RECORD_LENGTH (MNR) and the MAXIMUM_RECORD_LENGTH (MRL) defined for the file.

For F type records, the length of the new record must be equal to the MAXIMUM_RECORD_LENGTH (MRL) defined for the file.

The position of the file is not affected by the execution of a replace request.

You should include the IFETCH call to check whether an error occurred during the replace operation.

CALL REWND

The rewind request positions the file to beginning-of-information, which is the start of the record with the lowest primary key value.

The format of the rewind request is:

CALL STOREF (fit, 'EX', ex)
CALL REWND (fit)
CALL IFETCH (fit, 'ES', status)

fit Variable name for the file information table.

ex Error exit routine; default is zero.

status Variable containing the value of error status; zero after a successful operation.

The file must be open when you issue the rewind request. Rewinding a file is more efficient than extensive backward skipping of records.

You should include the IFETCH call to check whether an error occurred during the rewind operation.

CALL SKIP

A skip request repositions an indexed sequential file either forward or backward a specified number of records.

The format of the skip request is:

CALL STOREF (fit, 'EX', ex)
CALL SKIP (fit, *count***)**
CALL IFETCH (fit, 'ES', status)
CALL IFETCH (fit, 'SKP', skpct)
CALL IFETCH (fit, 'FP', filpos)

fit File information table.

count Number of records to be skipped. A positive count results in a skip forward, toward end-of-information (EOI), and a negative count results in a skip backward, toward beginning-of-information (BOI). The default is zero.

ex Error exit routine; default is zero.

filpos Variable containing the position of the file after skip.

skpct Variable containing the value of the residual skip count. The value is zero if the skip was completed; otherwise the value is a positive integer indicating the number of skips that were not performed.

status Variable containing the value of error status; zero after a successful operation.

A forward skip that cannot be completed results in end-of-information (EOI) status. A backward skip that cannot be completed results in beginning-of-information (BOI) status. If either BOI or EOI occurs, and you have set the ERROR_EXIT_PROCEDURE (EX) attribute to a subroutine name, the subroutine is executed.

A completed skip request positions the file at the beginning of a record, or at the end of the last record, with end-of-record (EOR) status returned to the FILE_POSITION (FP) attribute.

Information on the position of the file is returned to the FILE_POSITION (FP) attribute. The FILE POSITION is set to 1 for BOI, 16 for EOR, and 64 for EOI.

If the skip is not completed, the residual skip count is stored in the RESIDUAL_SKIP_COUNT (SKP) attribute. If the skip is completed, the RESIDUAL_SKIP_COUNT (SKP) attribute is set to zero.

The skip request should be used only for skipping a small number of records because each intervening record is read and counted, which increases execution time. A random read request takes less time than a lengthy skip request.

You should include the STOREF parameter to make sure that the appropriate value is stored in the ERROR_EXIT_PROCEDURE (EX) attribute. You should also include the three IFETCH calls to check whether an error occurred during the skip operation, and to check if the skip completed and the position of the file after the skip operation.

CALL STARTM

The start request positions the file to the beginning of the record with the primary key value that satisfies the specified primary key relation. The record is not returned to the working storage area. The file must be open for input or input/output.

The format of the start request is:

CALL STOREF (fit, 'REL', rel)
CALL STARTM (fit, *ka, kp, mkl, ex***)**
CALL IFETCH (fit, 'ES', status)
CALL IFETCH (fit, 'FP', filpos)

fit Variable name for the file information table.

ex Error exit routine; default is zero.

filpos Variable containing the position of the file after start operation.

ka Primary key used for positioning the file.

kp For FORTRAN 5 compatibility only (see appendix E). New applications should set this parameter to zero.

mkl Length in bytes of the major key. It is reset to zero after each START.

rel Relation of the primary key to the key specified in the start request:

'EQ' The primary key of the record is equal to the primary key specified by the ka parameter (default).

'GE' The primary key of the record is greater than or equal to the key specified by the ka parameter.

'GT' The primary key of the record is greater than the key specified by the ka parameter.

status Variable containing the value of error status; zero after successful operation.

If you want a relation other than EQ, you must set the REL (KEY_RELATION) parameter to establish the relationship between the primary key of the record at which the file is to be positioned and the primary key specified by the KA (KEY_ADDRESS) parameter.

If you set the REL parameter to EQ and an equal primary key does not exist in the file, a trivial error occurs and the file is positioned at the beginning of the record with the next highest key.

If you set the REL parmeter to GE or GT and no primary key in the file meets the condition, the end-of-data exit (DX) routine is executed, if specified.

You can also position the file using a major key value. A major key is a key that contains fewer bytes than the primary key of a record in the file, and is to be matched with an equal number of bytes at the beginning of the key of a record. If a CALL STARTM statement does not include the MKL (MAJOR_KEY_LENGTH) parameter, zero is always assumed regardless of any previous value because MKL is not saved in the file information table.

You should include the STOREF call to make sure that the REL parameter has the appropriate key relation value. You should also include the IFETCH calls to check whether any errors occurred during the operation and the position of the file after the operation.

CALL STOREF

The STOREF call sets or changes an attribute or parameter value during program execution.

The format of the CALL STOREF statement is:

CALL STOREF (fit, keyword, value)

fit

Variable name for the file information table.

keyword

Name of attribute or parameter whose value is to be stored. The attribute name must be a character expression.

value

Value to be stored in specified keyword. The value must be either a character expression (symbolic options), an integer representation (numeric options), or a subroutine name or variable name.

You can issue a CALL STOREF statement before or after opening the file. However, the statement must be after execution of the CALL FILEIS statement.

The following attributes cannot be set by the CALL STOREF statement after the file has been opened:

AVERAGE_RECORD_LENGTH
EMBEDDED_KEY
FILE_ORGANIZATION
KEY_LENGTH
KEY_POSITION
KEY_TYPE
MAXIMUM_RECORD_LENGTH
MINIMUM_RECORD_LENGTH
RECORD_TYPE

You can set only one attribute or parameter with each CALL STOREF statement. The abbreviation for the attribute or parameter and the value to be stored. The value can be an integer, integer variable, or a symbolic option in character format. Integer values are retained as right-justified integers in the file information table. Symbolic values become bit strings and, depending on the bit string length, are fetched as a single bit or an integer.

IFETCH

The format of the IFETCH function is:

IFETCH (fit, keyword)

fit

Variable name for the file information table.

keyword

Name of the attribute or parameter whose value is to be fetched. The attribute name must be a character expression.

IFETCH is an integer function that returns the current value of a single file attribute. A one-bit field is returned in the sign bit. If the value of the function is negative, the bit is set to one; if the value of the function is positive, the bit is set to zero. Note that when fetching COLLATE_TABLE_NAME, LOCAL_FILE_NAME, or ERROR_EXIT_NAME, only the first eight characters are returned. The characters are in left-justified FORTRAN boolean format.

Fetching addresses, such as WORKING_STORAGE_AREA (WSA), is not recommended because the program cannot do anything with the returned value.

FORTRAN Programming Examples

The remainding sections show how to create and update an indexed sequential file using FORTRAN.

Indexed Sequential File Creation Example

Figure 6-1 shows a FORTRAN program to create an indexed sequential file. The data used to create the file is shown in figure 6-2.

```
      Program FTNISCT
C  ************************************************
C  *  This program creates an indexed sequential     *
C  *  file (ISFILE) from a sequential file (DATAFIL). *
C  ************************************************
C  Declare variables.
      Integer isfit, reclen, stat
      Common key, iswsa
      Character * 65 iswsa
      Character * 20 key
C  Set file attributes before opening ISFILE.
      Call Fileis (isfit,'lfn','ISFILE',
     +            'mrl',65,'arl',65,'rt','f',
     +            'kl',20,'kt','u','rkp',0,'emk','yes',
     +            'ip',10,'dp',15,'erc',30,
     +            'dfc',3)
C  Open DATAFIL and ISFILE.  Check for error on ISFILE open.
      Open (2,file='DATAFIL')
      Call Openm (isfit,'NEW')
      stat = Ifetch (isfit,'es')
      If (stat.ne.0) go to 70
C  Read each record from DATAFIL into the working storage area (iswsa),
C  and then put record into ISFILE.  After put, check for error.  If no
C  error occured, print the record.
   10 Continue
         Read (2,'(A65)',End=30) iswsa
         Call Put (isfit,iswsa,reclen,key,0,0,0)
         Call Ifetch (isfit,'ES',stat)
         If (stat.ne.0) go to 70
         Print '(1X,A65)', iswsa
         Go to 10
C  When all records in DATAFIL have been read, close ISFILE and
C  check whether error occurred during CLOSE.
   30 Continue
      Call Closem (isfit)
      Call Ifetch (isfit,'ES',stat)
      If (stat.ne.0) Print 900, stat
      Stop
C  If error occurs during OPEN or PUT, control transfers to this point
C  in program.  The error number is printed and the file is closed.
   70 Continue
      Print 900, stat
      Call Closem (isfit)
      Stop
  900 Format (' ES = ',I6)
      End
```

Figure 6-1. Creation Program

```
Algeria          19709000    919591 Algiers      Africa
America         225195000   3615105 Washington   NAmerica
Australia        14796000   2967895 Canberra     Australia
Austria          74760000     32374 Vienna       Europe
Belgium           9875000     11781 Brussels     Europe
Canada           20050000   3851791 Ottawa       NAmerica
Denmark           5157000     16629 Copenhagen   Europe
England          55717000     94226 London       Europe
France           53844000    211207 Paris        Europe
India           700734000   1269340 New Delhi    Asia
Ireland           3349000     27136 Dublin       Europe
Ivory Coast       8513000    124503 Abidjan      Africa
Japan           118783000    143750 Yokohama     Asia
Mexico           70143000    761601 Mexico       SAmerica
Sweden            8335000    173731 Stockholm    Europe
Switzerland      63000000     15941 Bern         Europe
Tanzania         18744000    364898 Zanzibar     Africa
Turkey           47284000    301381 Ankara       Asia
United States   225195000   3615105 Washington   NAmerica
USSR            269302000   8649498 Moscow       Asia
Venezuela        15771000    352143 Caracas      SAmerica
West Germany     60948000     95976 Bonn         Europe
```

Figure 6-2. Data for Creation Program

In the program, after the files have been opened, records are read from the data file (DATAFIL) into a working storage area (ISWSA). The records are read from the working storage area (ISWSA) into the indexed sequential file (ISFILE) and are stored logically in ascending order by primary key values.

After all the records have been put in the indexed sequential file, the file is closed.

Output from the program is shown in figure 6-3. Note the INFORMATIVE messages that are output by the file interface routines.

```
Algeria              19709000    919591 Algiers      Africa
America             225195000   3615105 Washington   NAmerica
Australia            14796000   2967895 Canberra     Australia
Austria              74760000     32374 Vienna       Europe
Belgium               9875000     11781 Brussels     Europe
Canada               20050000   3851791 Ottawa       NAmerica
Denmark               5157000     16629 Copenhagen   Europe
England              55717000     94226 London       Europe
France               53844000    211207 Paris        Europe
India               700734000   1269340 New Delhi    Asia
Ireland               3349000     27136 Dublin       Europe
Ivory Coast           8513000    124503 Abidjan      Africa
Japan               118783000    143750 Yokohama     Asia
Mexico               70143000    761601 Mexico       SAmerica
Sweden               8335000     173731 Stockholm    Europe
Switzerland          63000000     15941 Bern         Europe
Tanzania             18744000    364898 Zanzibar     Africa
Turkey               47284000    301381 Ankara       Asia
United States       225195000   3615105 Washington   NAmerica
USSR                269302000   8649498 Moscow       Asia
Venezuela            15771000    352143 Caracas      SAmerica
West Germany         60948000     95976 Bonn         Europe
-- File ISFILE : 0 DELETE_KEYs done since last open.
-- File ISFILE : 0 GET_KEYs done since last open.
-- File ISFILE : 0 GET_NEXT_KEYs done since last open.
-- File ISFILE : 22 PUT_KEYs (and PUTREPs->put) since last open.
-- File ISFILE : 0 PUTREPs done since last open.
-- File ISFILE : 0 REPLACE_KEYs (and PUTREPs->replace) since last open.
```

Figure 6-3. Output from Creation Program

Indexed Sequential File Update Example

Figure 6-4 shows a program used to update ISFILE. Figure 6-5 shows the data used to update the file.

The update program uses the first three characters of each update record to determine whether the specified record is to be deleted, put, or replaced. The CALL PUTREP statement is used instead of the CALL PUT and CALL REPLACE statement. The CALL PUTREP statements puts a record in the file if a matching primary key is not found in the file, and replaces a record in the file if a matching primary key is found in the file.

The subroutine ErrXit is called whenever an error occurs. If an error occurs, either trivial, fatal, or catastrophic, ErrXit prints the error number and closes the file.

Figure 6-6 shows the program output. Note the INFORMATIVE messages that are output by the file interface routines.

```
      Program FTNISUD
C  ***************************************************************
C  *  This program updates an existing indexed sequential file   *
C  *  (ISFILE).  The update information is in file UPDTFIL       *
C  ***************************************************************
C  Declare variables.
      Integer isfit, reclen, filpos, stat
      Common iswsa, key, fit
      Character iswsa * 65, key * 20, fit * 1, code * 3
      External ErrXit
C  Set file attributes.
      Call Fileis (isfit, 'lfn', 'ISFILE', 'dfc', 3)
C  Open UPDTFIL and ISFILE.  Check for error on open of ISFILE.
      Open (2, file='UPDTFIL')
      Call Openm (isfit, 'I-O')
      stat = Ifetch (isfit, 'es')
      If (stat.ne.0) go to 90
      reclen = Ifetch (isfit, 'MRL')
C  Read UPDTFIL and based on the first field (code) in each
C  record of UPDTFIL, either delete, or put/replace a record.
   10 Continue
        Read (2, '(A3, A65)', End=30) code, iswsa
        key = iswsa (1:20)
        If (code.eq.'DLT') Then
          Call Dlte (isfit, key, 0, 0, ErrXit)
        Else If ((code.eq.'PUT').or.(code.eq.'RPL')) Then
          Call Putrep (isfit, iswsa, reclen, key, 0, 0, ErrXit)
        Else
          Print '('' Code '', A3,'' is invalid'')', code
        End If
        Go to 10
C  After all records in UPDTFIL have been read, print updated file.
C  Note the use of Ifetch and the stat variable in this part of the program.
   30 Continue
      Call Rewnd (isfit)
      stat = Ifetch (isfit, 'ES')
      If (stat.ne.0) Go to 90
      Call Storef (isfit, 'WSL', 65)
      Call Storef (isfit, 'EX', ErrXit)
   50 Continue
        Call Getn (isfit, iswsa)
        stat = Ifetch (isfit, 'ES')
        If (stat.ne.0) Go to 90
        filpos = Ifetch (isfit, 'FP')
        If (filpos.eq.64) go to 70
        reclen = Ifetch (isfit, 'rl')
        Print 900, iswsa, reclen, filpos
        Go to 50
```

(Continued on next page)

Figure 6-4. Update Program

(Continued from previous page)

```
C  Close the file and stop.
   70 Continue
      Call Closem (isfit)
      Call Ifetch (isfit, 'ES', stat)
      If (stat.ne.0) Print '(1X, I6)', stat
      Stop
   90 Continue
      Print '(" Error -- ES = ", I6)', stat
      Call Closem (isfit)
      Stop
  900 Format (1X,' Record = ', A65, 2X, ' RL = ', I3, 2X, ' FP = ', I1)
      End
C  ************************************************************
C  * The program will transfer control to ErrXit if an error *
C  * occurs during execution of the file processing calls.    *
C  ************************************************************
      Subroutine ErrXit
      Common fit
      Character * 1, fit
      Integer stat
      stat = Ifetch (isfit, 'ES')
      Print '('' Executing ErrXit.  ES = '', I6)', stat
      Call Closem (isfit)
      Stop 'ERROR OCCURRED'
      End
```

Figure 6-4. Update Program

```
RPLCanada          24336000   3851791 Montreal     NAmerica
PUTChina         1053788000   3705390 Beijing      Asia
DLTAmerica
PUTSpain           38686000    194897 Madrid       Europe
PUTItaly           57513000    116303 Rome         Europe
RPLJapan           11878300    143750 Tokyo        Asia
```

Figure 6-5. Update File

```
Record = Algeria          19709000    919591 Algiers      Africa     RL =  65  FP = 16
Record = Australia        14796000   2967895 Canberra     Australia  RL =  65  FP = 16
Record = Austria          74760000     32374 Vienna       Europe     RL =  65  FP = 16
Record = Belgium           9875000     11781 Brussels     Europe     RL =  65  FP = 16
Record = Canada           24336000   3851791 Montreal     NAmerica   RL =  65  FP = 16
Record = China          1053788000   3705390 Beijing      Asia       RL =  65  FP = 16
Record = Denmark           5157000     16629 Copenhagen   Europe     RL =  65  FP = 16
Record = England          55717000     94226 London       Europe     RL =  65  FP = 16
Record = France           53844000    211207 Paris        Europe     RL =  65  FP = 16
Record = India           700734000   1269340 New Delhi    Asia       RL =  65  FP = 16
Record = Ireland           3349000     27136 Dublin       Europe     RL =  65  FP = 16
Record = Italy            57513000    116303 Rome         Europe     RL =  65  FP = 16
Record = Ivory Coast       8513000    124503 Abidjan      Africa     RL =  65  FP = 16
Record = Japan            11878300    143750 Tokyo        Asia       RL =  65  FP = 16
Record = Mexico           70143000    761601 Mexico       SAmerica   RL =  65  FP = 16
Record = Spain            38686000    194897 Madrid       Europe     RL =  65  FP = 16
Record = Sweden            8335000    173731 Stockholm    Europe     RL =  65  FP = 16
Record = Switzerland      63000000     15941 Bern         Europe     RL =  65  FP = 16
Record = Tanzania         18744000    364898 Zanzibar     Africa     RL =  65  FP = 16
Record = Turkey           47284000    301381 Ankara       Asia       RL =  65  FP = 16
Record = USSR            269302000   8649498 Moscow       Asia       RL =  65  FP = 16
Record = United States   225195000   3615105 Washington   NAmerica   RL =  65  FP = 16
Record = Venezuela        15771000    352143 Caracas      SAmerica   RL =  65  FP = 16
Record = West Germany     60948000     95976 Bonn         Europe     RL =  65  FP = 16
--  File ISFILE : AMP$GET_NEXT_KEY has reached a file boundary : EOI.
--  File ISFILE :  1 DELETE_KEYs done since last open.
--  File ISFILE :  0 GET_KEYs done since last open.
--  File ISFILE :  24 GET_NEXT_KEYs done since last open.
--  File ISFILE :  3 PUT_KEYs (and PUTREPs->put) since last open.
--  File ISFILE :  5 PUTREPs done since last open.
--  File ISFILE :  2 REPLACE_KEYs (and PUTREPs->replace) since last open.
```

Figure 6-6. Output from Update Program

File Attributes and Parameters

The file interface subprograms communicate with the internal input/output routines through parameters specified in the CALL statements. Most of these parameters cause a value to be set for a file attribute.

A file attribute is a characteristic of a file. File attributes describe the structure of the file and define processing limitations for the file. The file attribute values are stored in an internal file information table, and remain valid until the file is closed.

This subsection presents a list of the values for file attributes and subprogram call parameters pertaining to indexed sequential files. Subprogram parameters that do not correspond to a file attribute are indicated in the parameter descriptions.

File attributes can be separated into the following categories:

Preserved (structural)

Preserved attributes determine file structure; these values are kept for the lifetime of the file, although the values of some preserved attributes can be changed with the CHANGE_FILE_ATTRIBUTE (CHAFA) command. (See the SCL Quick Reference manual for further information on the CHAFA command.)

Temporary

Temporary attributes determine how the job processes the file. These values are discarded when the file is detached.

Returned

Returned attributes are set internally and can be fetched for information on the current state of the file or the last file processing operation.

Certain attributes can be set by the file interface calls. Those attributes are specified as parameters in the calls. The value is stored in the file information table and remains valid until the file is closed.

Most of the attribute names have three forms that can be used in the file interface calls: an SCL-style name, an SCL-style abbreviation, and a FORTRAN abbreviation. The SCL-style names and abbreviations are identical to those used for the SETFA and CHAFA commands, except that they are preceded by a dollar sign ($). The FORTRAN abbreviations are identical to those used in NOS FORTRAN 5. The SCL-style names, SCL-style abbreviations, and FORTRAN abbreviations are equivalent; in the file interface calls, you can use any of the three forms.

In some cases, the attribute values also have three forms, corresponding to the three forms of the attribute name. The three forms of the attribute values are equivalent; however, the SCL-style values can be used only with SCL-style attribute names, and the FORTRAN values can be used only with FORTRAN attribute names. For example, the following FILEIS calls are equivalent. The first call uses SCL-style names for the attribute name and value, the second call uses SCL abbreviations, and the third call uses FORTRAN abbreviations:

```
CALL FILEIS (AFILE, '$ACCESS_MODE', 'READ')

CALL FILEIS (AFILE, '$AM', 'R')

CALL FILEIS (AFILE, 'PD', 'INPUT')
```

The following FILEIS call is incorrect becauses it mixes a FORTRAN abbreviation with an SCL parameter value:

```
CALL FILEIS (AFILE, 'PD', 'READ')
```

In the following file interface call descriptions, the three forms of the attribute names are shown in parentheses. The SCL name is shown first, followed by the SCL abbreviation, followed by the FORTRAN abbreviation. In the descriptions of the attribute values, the name type (FORTRAN or SCL) for which the values are valid is shown in parentheses. If no name type is shown after a parameter value, the same value is used for both FORTRAN and SCL attribute names.

Note that the OLD/NEW, OPEN/CLOSE, FATAL/NONFATAL, LEVELS_OF_INDEXING, and RECORD_LENGTH attributes do not have SCL names, as they are provided only for NOS FORTRAN 5 compatibility.

Table 6-4 summarizes the file attributes and subprogram call parameters, shows the names and abbreviations used in the file interface call formats, and indicates the life of the attribute or parameter.

Table 6-4. File Attribute and Parameters for File Interface Calls

Attribute or Parameter Name	SCL-style Name	SCL-style Abbreviation	FORTRAN Abbreviation
ACCESS_MODE	$ACCESS_MODE	$AM	PD
AVERAGE_RECORD_LENGTH	$AVERAGE_RECORD_LENGTH	$ARL	ARL
COLLATE_TABLE_NAME	$COLLATE_TABLE_NAME	$CTN	CTN
COLLATE_TABLE	$COLLATE_TABLE	$CT	DCT
DATA_PADDING	$DATA_PADDING	$DP	DP
EMBEDDED_KEY	$EMBEDDED_KEY	$EK	EMK
ERROR_COUNT	$ERROR_COUNT	$EC	ECT
ERROR_EXIT_NAME	$ERROR_EXIT_NAME	$EEN	EXN
ERROR_LIMIT	$ERROR_LIMIT	$EL	ERL
ERROR_STATUS	$ERROR_STATUS	$ES	ES
ESTIMATED_RECORD_COUNT	$ESTIMATED_RECORD_COUNT	$ERC	ERC
FATAL/NONFATAL	-	-	FNF
FILE_IDENTIFIER	$FILE_IDENTIFIER	$FI	-
FILE_ORGANIZATION	$FILE_ORGANIZATION	$FO	FO
FILE_POSITION	$FILE_POSITION	$FP	FP
FORCED_WRITE	$FORCED_WRITE	$FW	FWI
INDEX_LEVEL	$INDEX_LEVEL	$IL	NL
LEVELS_OF_INDEXING	-	-	NL
LOCAL_FILE_NAME	$LOCAL_FILE_NAME	$LFN	LFN
INDEX_PADDING	$INDEX_PADDING	$IP	IP
KEY_ADDRESS	$KEY_ADDRESS	$KA	KA
KEY_LENGTH	$KEY_LENGTH	$KL	KL
KEY_TYPE	$KEY_TYPE	$KT	KT
KEY_POSITION	$KEY_POSITION	$KP	KP
KEY_RELATION	$KEY_RELATION	$KR	KR
LAST_OPERATION	$LAST_OPERATION	$LO	LOP
MAJOR_KEY_LENGTH	$MAJOR_KEY_LENGTH	$MKL	MKL
MAXIMUM_BLOCK_LENGTH	$MAXIMUM_BLOCK_LENGTH	$MAXBL	MBL
MXIMUM_RECORD_LENGTH	$MAXIMUM_RECORD_LENGTH	$MAXRL	MRL
MESSAGE_CONTROL	$MESSAGE_CONTROL	$MC	DFC
MINIMUM_RECORD_LENGTH	$MINIMUM_RECORD_LENGTH	$MINRL	MNR
OLD/NEW	-	-	ON
OPEN/CLOSE	-	-	OC
OPEN_POSITION	$OPEN_POSITION	$OP	OF
RECORD_LENGTH	-	-	RL
RECORD_LIMIT	$RECORD_LIMIT	$RL	FLM
RECORD_TYPE	$RECORD_TYPE	$RT	RT
RECORDS_PER_BLOCK	$RECORDS_PER_BLOCK	$RPM	RB
RESIDUAL_SKIP_COUNT	-	-	RA
SKIP_COUNT	$SKIP_COUNT	$SC	SKP
WORKING_STORAGE_AREA	$WORKING_STORAGE_AREA	$WSA	WSA
WORKING_STORAGE_LENGTH	$WORKING_STORAGE_LENGTH	$WSL	WSL

ACCESS_MODE ($ACCESS_MODE, $AM, PD)

The ACCESS_MODE attribute is the set of access permissions. (Temporary attribute)

The FORTRAN values for ACCESS_MODE are:

'INPUT'

Records can be read.

'OUTPUT'

Records can be shortened, appended, and modified.

'I-O' or 'IO'

Records can be read, shortened, appended, and modified.

'NEW'

Records can be read, shortened, appended, and modified.

Omission causes 'INPUT' to be used.

The SCL values for ACCESS_MODE are:

'READ'

GET, GETN, GETNR, REWIND, or START may be called.

'APPEND'

PUT may be called.

'SHORTEN'

DLTE may be called.

'MODIFY'

File statistics will be kept.

You can specify two or more values by enclosing the values in a single pair of apostrophes and separating the values with a comma.

If you do not specify a value for the ACCESS_MODE parameter, the value defaults to READ.

Calling REPLC requires both APPEND and SHORTEN permission.

AVERAGE_RECORD_LENGTH ($AVERAGE_RECORD_LENGTH, $ARL, ARL)

The AVERAGE_RECORD_LENGTH attribute estimates the average length, in bytes, of all records in the file. Used only at open-new time as a possible guide to choosing a block size. (Preserved attribute)

AVERAGE_RECORD_LENGTH can be set to a value from zero through MAXIMUM_RECORD_LENGTH.

When the file contains variable length records, you should determine the value of AVERAGE_RECORD_LENGTH as follows:

If most records in the file are a specific length, set AVERAGE_RECORD_LENGTH to that length.

If record lengths are well distributed, AVERAGE_RECORD_LENGTH should be set to the median record length. That is, half the records are larger and half are smaller than the value of AVERAGE_RECORD_LENGTH.

If you do not specify AVERAGE_RECORD_LENGTH, the arithmetic mean of MAXIMUM_RECORD_LENGTH and MINIMUM_RECORD_LENGTH is used to calculate MAXIMUM_BLOCK_LENGTH. However, the AVERAGE_RECORD_LENGTH attribute is not set to this value.

COLLATE_TABLE ($COLLATE_TABLE, $CT, DCT)

The COLLATE_TABLE attribute specifies the address of the collation table. The address is a character array name that has 256 elements. When the FORTRAN interface is later called to open the file, the table is copied from address DCT to a location with a known entry point name. CTN is then used internally to send the entry point name to the system before the real OPEN. (Preserved attribute)

COLLATE_TABLE_NAME ($COLLATE_TABLE_NAME, $CTN, CTN)

The COLLATE_TABLE_NAME attribute specifies the name of the collation table. The table exists as a program in a library or as an entry point already loaded. The collate table is specified using COLLATE_TABLE_NAME and a string. The COLSEQ routine is used to create a table named FTV$USER_COLLATE_TABLE which can be specified as the COLLATE_TABLE_NAME. (Preserved attribute)

The name can be 31 characters long and cannot start with a digit. Valid characters are A through Z, 0 through 9, _, #, @, and $.

Once a file is open, COLLATE_TABLE_NAME cannot be set but can be fetched.

For information on creating a collation table, see appendix H.

DATA_PADDING ($DATA_PADDING, $DP, DP)

The DATA_PADDING attribute specifies the percentage of empty space to be left when records are written to the data blocks at file creation time. The empty space allows for easy file expansion during later file processing operations. DATA_PADDING is used only at file creation time. It applies only for the period between the first open and close. (Preserved attribute)

The value can be from 0 to 99.

Omission for a new file results in no (zero) padding.

EMBEDDED_KEY ($EMBEDDED_KEY, $EK, EMK)

The EMBEDDED_KEY attribute is used to locate the primary key. (Preserved attribute)

EMBEDDED_KEY can be set as follows:

'YES' (FORTRAN); 'YES' or 'y' or 'TRUE' or 'T' (SCL)

Primary key is stored within the record. This is the default value.

'NO' (FORTRAN); 'NO' or 'N' or 'FALSE' or 'F' (SCL)

Primary key is stored separately from the record.

Omission results in embedded keys (TRUE or YES).

ERROR_COUNT ($ERROR_COUNT, $EC, ECT)

The ERROR_COUNT attribute indicates the number of trivial errors that have occurred on the file since the last open. (Returned attribute)

ERROR_EXIT_NAME ($ERROR_EXIT_NAME, $EEN, EXN)

The ERROR_EXIT_NAME attribute is the name of an error exit routine, existing as a program in a library or as an entry point already in memory. ERROR_EXIT_NAME can be set only when the file is not open. At the next open, the system will load the routine and link it up. (Temporary attribute)

In FORTRAN, ERROR_EXIT_NAME is a character string up to 31 characters in length. The string must be the name of a subroutine in the global library set of the job or else the routine identified by the string must be already loaded in the task using the CALL FILEIS statement. Valid characters are A through Z, 0 through 9, _, #, @, and $.

There is no default name. If you do not specifiy a name, no error processing procedure is used.

ERROR_EXIT_NAME can be cleared using the CALL STOREF statement to store blanks in the ERROR_EXIT_NAME attribute. (The STOREF call takes effect on the next open.)

ERROR_EXIT_PROCEDURE ($ERROR_EXIT_PROCEDURE, $EEP, EX)

The ERROR_EXIT_PROCEDURE attribute specifies where control is to transfer if a fatal or trivial error occurs. ERROR_EXIT_PROCEDURE can be set only while the file is open. (Temporary attribute)

ERROR_EXIT_PROCEDURE is a pointer variable to the error processing procedure. In FORTRAN, ERROR_EXIT_PROCEDURE is a subroutine name that is declared as EXTERNAL.

There is no default value. The current error processing procedure address is used.

ERROR_EXIT_PROCEDURE can be cleared by using the CALL STOREF statement to store a value of zero in the ERROR_EXIT_PROCEDURE attribute.

ERROR_LIMIT ($ERROR_LIMIT, $EL, ERL)

The ERROR_LIMIT attribute is the limit to the number of trivial errors that can occur before a fatal error occurs. ERROR_LIMIT is compared to ERROR_COUNT to determine when the error limit has been reached. (Temporary attribute)

You can set ERROR_LIMIT to a value between 0 and 69904.

Omission causes zero to be used meaning that an unlimited number of trivial errors can occur.

FILE_IDENTIFIER ($FILE_IDENTIFIER, $FI)

The FILE_IDENTIFIER attribute specifies a left-justified file identifier for use by a CYBIL subprogram. The IFETCH call is the only FORTRAN call that can access the FILE_IDENTIFIER attribute. The FILE_IDENTIFIER attribute can then be passed to a CYBIL subprogram, which can use the FILE_IDENTIFIER attribute in CYBIL/file interface calls.

An important restriction on using CYBIL subprograms for file interface operations is that a file opened by a FORTRAN program must be closed by FORTRAN call; otherwise, the results of the operations are undefined. If a CYBIL subprogram calls AMP$STORE with the file identifier received from the FORTRAN program, subsequent IFETCH calls may return out of date information. Parameters which are also FIT fields, such as FILE_POSITION, and WORKING_STORAGE_AREA, are not updated by file interface calls made outside the FORTRAN program.

Calling CYBIL subprograms from a FORTRAN program is described in chapter 7.

ERROR_STATUS ($ERROR_STATUS, $ES, ES)

The ERROR_STATUS attribute indicates the value of the most recent error condition to occur on the file. ERROR_STATUS is cleared at the beginning of each file processing request. See appendix B for the list of condition codes. (Returned attribute)

ESTIMATED_RECORD_COUNT ($ESTIMATED_RECORD_COUNT, $ERC, ERC)

The ESTIMATED_RECORD_COUNT attribute is your estimate of the maximum number of records the file will hold. ESTIMATED_RECORD_COUNT is used only at open-new time as a possible guide to choosing a block size. (Preserved attribute)

You can set ESTIMATED_RECORD_COUNT to any integer value.

If ESTIMATED_RECORD_COUNT has not been defined, the value of RECORD_LIMIT (FLM) is used. If RECORD_LIMIT is not defined, the estimate of 100000 records is used.

FATAL/NONFATAL (FNF)

The FATAL/NONFATAL attribute indicates whether the error that occurred is fatal or nonfatal. (Returned attribute)

The values of the FNF attribute are as follows:

0 The error is nonfatal.

1 The error is fatal.

FILE_ORGANIZATION ($FILE_ORGANIZATION, $FO, FO)

The FILE_ORGANIZATION attribute specifies the organization of the file. The only organization you can specify is indexed sequential. (Preserved attribute)

FILE_ORGANIZATION can be set by the FILEIS call to one of the following values:

'INDEXED_SEQUENTIAL' or 'IS' (SCL)

'IS' (FORTRAN)

A fatal error occurs if you set FILE_ORGANIZATION to any other value in a FILEIS call.

Omission causes indexed sequential file organization to be used because FO automatically is set to 'IS' by the CALL FILEIS statement.

FILE_POSITION ($FILE_POSITION, $FP, FP)

The FILE_POSITION parameter value indicates the position of the file after the last file operation. (Returned attribute)

The returned value will be one of the following decimal values:

1 File is positioned at beginning-of-information (BOI).

16 File is positioned at end-of-record (EOR).

64 File is positioned at end-of-information (EOI).

When the file is opened, but before any records are processed, FILE_POSITION has the same value as OPEN_POSITION. Normally, this value is BOI.

FORCED_WRITE ($FORCED_WRITE, $FW, FWI)

The FORCED_WRITE attribute is used for information on whether the file is to be written to mass storage immediately after modification or whether modified blocks can remain in memory without a mass storage backup copy until the next flush or close request. (Preserved attribute)

FORCED_WRITE is set as follows:

'UNFORCED' (FORTRAN); 'NO' or 'N' or 'FALSE' or 'F' (SCL)

Modified blocks can remain in memory without a mass storage backup copy.

'NO' (FORTRAN); 'FORCED_IF_STRUCTURE_CHANGE' or 'FISC' (SCL)

Modified blocks are written to mass storage immediately after any structure change to the file that affects more than one block.

'YES' (FORTRAN); 'YES' or 'Y' or 'TRUE' or 'T' (SCL)

Each modified block is written to mass storage immediately after modification occurs.

Omission causes NO to be used.

INDEX_LEVELS ($INDEX_LEVELS, $IL, NL)

The INDEX_LEVELS attribute is a target of the number of index levels in the indexed sequential file. You can set INDEX_LEVELS at file creation time and an indexed sequential file is created with the specified number of index levels. INDEX_LEVELS is used only at file creation time. (Preserved attribute)

The value can be from 1 to 15.

Omission causes the estimate of 2 records to be used.

INDEX_PADDING ($INDEX_PADDING, $IP, IP)

The INDEX_PADDING attribute specifies the percentage of empty space to be left in each index block at file creation time. The empty space allows for easy file expansion during later file processing operations. INDEX_PADDING is used only at file creation. It applies only between the first open and close. (Preserved attribute)

The value can be from 0 to 99.

Omission for a new file results in no (zero) padding.

KEY_ADDRESS ($KEY_ADDRESS, $KA, KA)

The KEY_ADDRESS parameter is used for file processing calls. The function of KEY_ADDRESS depends on the operation; either you place the key of the record to be processed in KEY_ADDRESS, or the key of the record that was processed is returned to KEY_ADDRESS. (Parameter)

KEY_LENGTH ($KEY_LENGTH, $KL, KL)

The KEY_LENGTH attribute specifies the length of the primary key being defined. You must set KEY_LENGTH before a new file can be opened. (Preserved attribute)

For files with embedded keys, the value can be from 1 to 255, but not greater than MINIMUM_RECORD_LENGTH. For files with nonembedded keys, the value can be from 1 to 255.

There is no default; you must set KEY_LENGTH at file creation.

KEY_POSITION ($KEY_POSITION, $KP, RKP)

The KEY_POSITION attribute indicates the byte offset into each record where the primary key begins. KEY_POSITION should be specified if the key is embedded (EMK=YES). KEY_POSITION is ignored for files with nonembedded keys (EMK=NO). (Preserved attribute)

KEY_POSITION can have a value from zero to MAXIMUM_RECORD_LENGTH specified for the file; however, KEY_POSITION + KEY_LENGTH must not be greater than MINIMUM_RECORD_LENGTH. (The value of KEY_POSITION + 1 defines the first character of the key. For example, if KEY_POSITION equals three, the key begins with the fourth character of the record.)

Omission for a new file causes zero to be used.

KEY_RELATION ($KEY_RELATION, $KR, REL)

The KEY_RELATION (rel) parameter specifies the relation of the primary keys of records in the file to the value at KEY_ADDRESS. KEY_RELATION is significant only for START (the file is positioned at the first record satisfying the relation) and GET (the first record satisfying the relation is returned to WORKING_STORAGE_AREA) operations. (Parameter)

KEY_RELATION can have the following values:

'EQ' (FORTRAN); 'EQUAL_KEY' or 'EK' (SCL)

The primary key of the record is equal to the primary key specified by the KEY_ADDRESS (ka) parameter (default).

'GE' (FORTRAN); 'GREATER_OR_EQUAL_KEY' or 'GOEK' (SCL)

The primary key of the record is greater than or equal to the key specified by the KEY_ADDRESS (ka) parameter.

'GT' (FORTRAN); 'GREATER_KEY' or 'GK' (SCL)

The primary key of the record is greater than the key specified by the KEY_ADDRESS (ka) parameter.

KEY_TYPE ($KEY_TYPE, $KT, KT)

The KEY_TYPE attribute indicates the type of the primary key being defined. (Preserved attribute)

KEY_TYPE can be set as follows:

'U' (FORTRAN); 'UNCOLLATED' or 'U' (SCL)

Uncollated-character keys can be from 1 to 255 characters long and are compared using the ASCII collating sequence.

'I' (FORTRAN); 'INTEGER' or 'I' (SCL)

Integer keys can be from 1 to 8 bytes (each byte is 8 bits) long and are compared as signed integers.

'S' (FORTRAN); 'COLLATED' or 'C' (SCL)

Collated-character keys can be from 1 to 255 characters long and are compared using a collation table. If you specify this key type, you must supply an explicit collation table; there is no system-supplied default collation table. Keys are not stored in collated form; therefore, collation tables need not be one-to-one. (Hardware instructions for operating on collated keys are used.)

Omission for a new file causes U to be used.

LAST_OPERATION ($LAST_OPERATION, $LO, LOP)

The LAST_OPERATION attribute indicates the last operation requested for the file. (Returned attribute)

The following values can be returned:

1 Open request (CALL OPENM)
2 Close request (CALL CLOSEM)
3 Random read request (CALL GET)
4 Sequential read request (CALL GETN)
5 Write request (CALL PUT)
7 Sequential read request (CALL GETNR)
8 Delete request (CALL DLTE)
9 Replace request (CALL REPLC)
10 Rewind request (CALL REWND)
11 Put/replace request (CALL PUTREP)
12 Skip forward request (CALL SKIP)
13 Skip backward request (CALL SKIP)
14 Start request (CALL STARTM)

The LAST_OPERATION attribute does not have values for the CALL FLUSHM, CALL IFETCH, IFETCH, or CALL STOREF statements. The value returned if the last request was a flush, fetch or store is the value of the previous file interface call.

LEVELS_OF_INDEXING (NL)

The LEVELS_OF_INDEXING attribute indicates the number of index levels that currently exist in the indexed sequential file. (Returned attribute)

This attribute is returned by a fetch request on the INDEX_LEVEL attribute.

LOCAL_FILE_NAME ($LOCAL_FILE_NAME, $LFN, LFN)

The LOCAL_FILE_NAME parameter specifies the file's local file name. This is a required parameter; it must be specifies in a FILEIS or STOREF call before the file is opened. The value for LOCAL_FILE_NAME can be any valid SCL name.

MAJOR_KEY_LENGTH ($MAJOR_KEY_LENGTH, $MKL, MKL)

The MAJOR_KEY_LENGTH parameter is used to position the file at the first record with a key in which the first specified number of characters matches the specified major key (stored in KEY_ADDRESS). MAJOR_KEY_LENGTH is reset to zero after each CALL STARTM or CALL GET for which it is specified. (Parameter)

MAJOR_KEY_LENGTH can be set from 0 to the value of KEY_LENGTH.

There is no default.

MAXIMUM_BLOCK_LENGTH ($MAXIMUM_BLOCK_LENGTH, $MAXBL, MBL)

The MAXIMUM_BLOCK_LENGTH attribute indicates the maximum length in bytes of a block (data or index). (Preserved attribute)

MAXIMUM_BLOCK_LENGTH must be between 1 and 65536 bytes. (The value is rounded internally up to the closest power of 2 between 2048 and 65536 bytes, inclusive.)

Omission for a new file causes an appropriate value to be calculated (as described earlier in this section under Indexed Sequential File Processing Concepts).

MAXIMUM_RECORD_LENGTH ($MAXIMUM_RECORD_LENGTH, $MAXRL, MRL)

The MAXIMUM_BLOCK_LENGTH attribute specifies the length in bytes of the longest record in the file. You must set MAXIMUM_RECORD_LENGTH at file creation time. (Preserved attribute)

For indexed sequential files, MAXIMUM_RECORD_LENGTH must be between 1 and 65497 bytes.

There is no default. A fatal error occurs if you do no set MAXIMUM_RECORD_LENGTH at file creation.

MESSAGE_CONTROL ($MESSAGE_CONTROL, $MC, DFC)

The MESSAGE_CONTROL attribute controls the listing of trivial error messages (fatal error messages are always logged), statistics, and notes on the $ERROR file. (Temporary attribute)

FORTRAN values for MESSAGE_CONTROL are:

0 Only fatal error messages are logged.

1 Trivial error messages are logged.

2 Informative messages and statistics are logged.

3 All are logged.

4 Informative messages are logged.

5 Statistics written are logged.

6 Trivial error and informative messages are logged.

7 Trivial error messages and statistics are logged.

SCL values for MESSAGE_CONTROL are:

'MESSAGES' or 'M'

Only informative messages are logged.

'STATISTICS' or 'S'

Only statistical information is logged.

'TRIVIAL_ERRORS' or 'T'

Only trivial messages are logged.

You can specify two or more values by enclosing the values in a single pair of apostrophes and separating the values by a comma. For example:

'M,S'

If you do not set the MESSAGE_CONTROL attribute, only fatal messages are logged.

MINIMUM_RECORD_LENGTH ($MINIMUM_RECORD_LENGTH, $MINRL, MNR)

The MINIMUM_REOCRD_LENGTH attribute specifies the length in bytes of the smallest expected or possible record in the file. (Preserved attribute)

MINIMUM_RECORD_LENGTH can be from 0 through 65497 bytes, but must not be greater than MAXIMUM_RECORD_LENGTH.

For files with fixed-length records, the default value is MAXIMUM_RECORD_LENGTH. For records with embedded keys, the default value is the sum of KEY_POSITION and KEY_LENGTH. For other files, the default value is 1 byte.

OLD/NEW (ON)

The OLD/NEW attribute specifies whether the next open request is creating a new file or opening an existing file. (Temporary attribute)

The OLD/NEW attribute can have the following values:

'OLD' The file exists.

'NEW' The file is being created.

Omission causes 'OLD' to be used.

The OLD/NEW attribute can be set explicitly in the CALL FILEIS statement or indirectly in the CALL OPENM statement through the setting of the pd parameter.

OPEN/CLOSE (OC)

The OPEN/CLOSE attribute indicates whether a file is opened or closed. (Returned attribute)

The values of the OPEN/CLOSE attribute are as follows:

0 The file has never been opened.

1 The file is open.

2 The file is closed.

OPEN_POSITION ($OPEN_POSITION, $OP, OF)

The OPEN_POSITION attribute specifies the position of file when it is opened. (Temporary attribute)

OPEN_POSITION can be set as follows:

'R' (FORTRAN); '$BOI' (SCL)

File is opened at beginning-of-information.

'N' (FORTRAN); '$ASIS' (SCL)

File is opened at current position.

'E' (FORTRAN); '$EOI' (SCL)

File is opened at end-of-information.

If you do not set OPEN_POSITION, the file is opened at beginning-of-information.

RECORD_LENGTH (RL)

On a get request, RECORD_LENGTH is set internally to indicate the exact size of record that is retrieved. (Returned attribute)

On a put request, you must specify the RECORD_LENGTH if the indexed sequential file has variable or undefined type records. If the file has fixed type records, the value of the MAXIMUM_RECORD_LENGTH attribute is used as the RECORD_LENGTH. (Temporary attribute)

RECORD_LIMIT ($RECORD_LIMIT, $RL, FLM)

The RECORD_LIMIT attribute specifies the maximum number of records in the file. (Preserved attribute)

RECORD_LIMIT can be set from 1 up to 2147483648.

Omission for a new file causes 2147483648 to be used.

RECORD_TYPE ($RECORD_TYPE, $RT, RT)

The RECORD_TYPE attribute specifies the type of records in the file. (Preserved attribute)

RECORD_TYPE can be set as follows:

'U' (FORTRAN); 'UNDEFINED' or 'U' (SCL)

Undefined length records

'V' (FORTRAN); 'VARIABLE' or 'V' (SCL)

CDC variable length records

'F' (FORTRAN); 'FIXED' or 'F' (SCL)

ANSI-fixed length records

U and V type records are internally equivalent.

Omission causes U type records to be used.

RECORDS_PER_BLOCK ($RECORDS_PER_BLOCK, $RPB, RB)

The RECORDS_PER_BLOCK attribute is your estimate of the number of records each data block should contain. RECORDS_PER_BLOCK is used only at file creation time in block size calculations. (Preserved attribute)

You can set RECORDS_PER_BLOCK to any integer value. (Information on choosing a value for RECORDS_PER_BLOCK is presented earlier in this section under Indexed Sequential File Structure.)

Omission causes the estimate of 2 records per block to be used.

RESIDUAL_SKIP_COUNT ($RESIDUAL_SKIP_COUNT, $RSC, SKP)

The RESIDUAL_SKIP_COUNT attribute is set to the difference between the numbers of records you requested to be skipped (in the CALL SKIP statement) and the number of records that were actually skipped. (Returned attribute)

A value of zero indicates the skip operation completed. A nonzero value indicates that the skip operation was not completed.

SKIP_COUNT ($SKIP_COUNT, $SC, SKP)

The SKIP_COUNT parameter specifies the number of records to be skipped. The sign of the SKIP_COUNT specifies the skip direction. (Parameter)

WORKING_STORAGE_ADDRESS ($WORKING_STORAGE_ADDRESS, $WSA, WSA)

The WORKING_STORAGE_ADDRESS (wsa) parameter specifies the area in memory to be used for write and read operations on the file. (Parameter)

WORKING_STORAGE_LENGTH ($WORKING_STORAGE_LENGTH, $WSL, WSL)

The WORKING_STORAGE_LENGTH specifies the length, in bytes, of the working storage area. (Parameter)

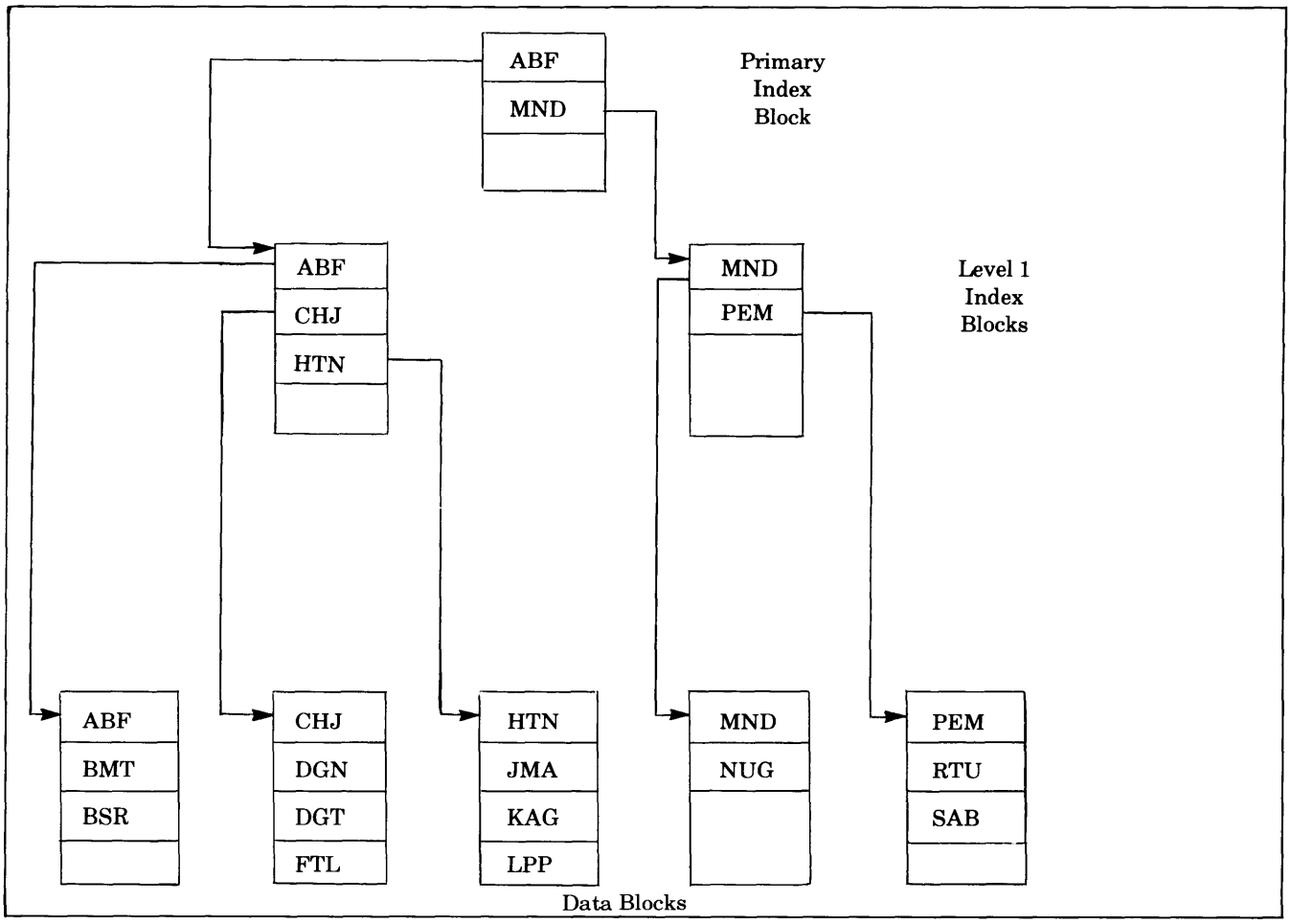
WORKING_STORAGE_LENGTH should be set to a value that is equal to or greater than the value of MAXIMUM_RECORD_LENGTH.

If WORKING_STORAGE_LENGTH is not set, the value of MAXIMUM_RECORD_LENGTH is used for read requests, and RECORD_LENGTH is used for write requests.

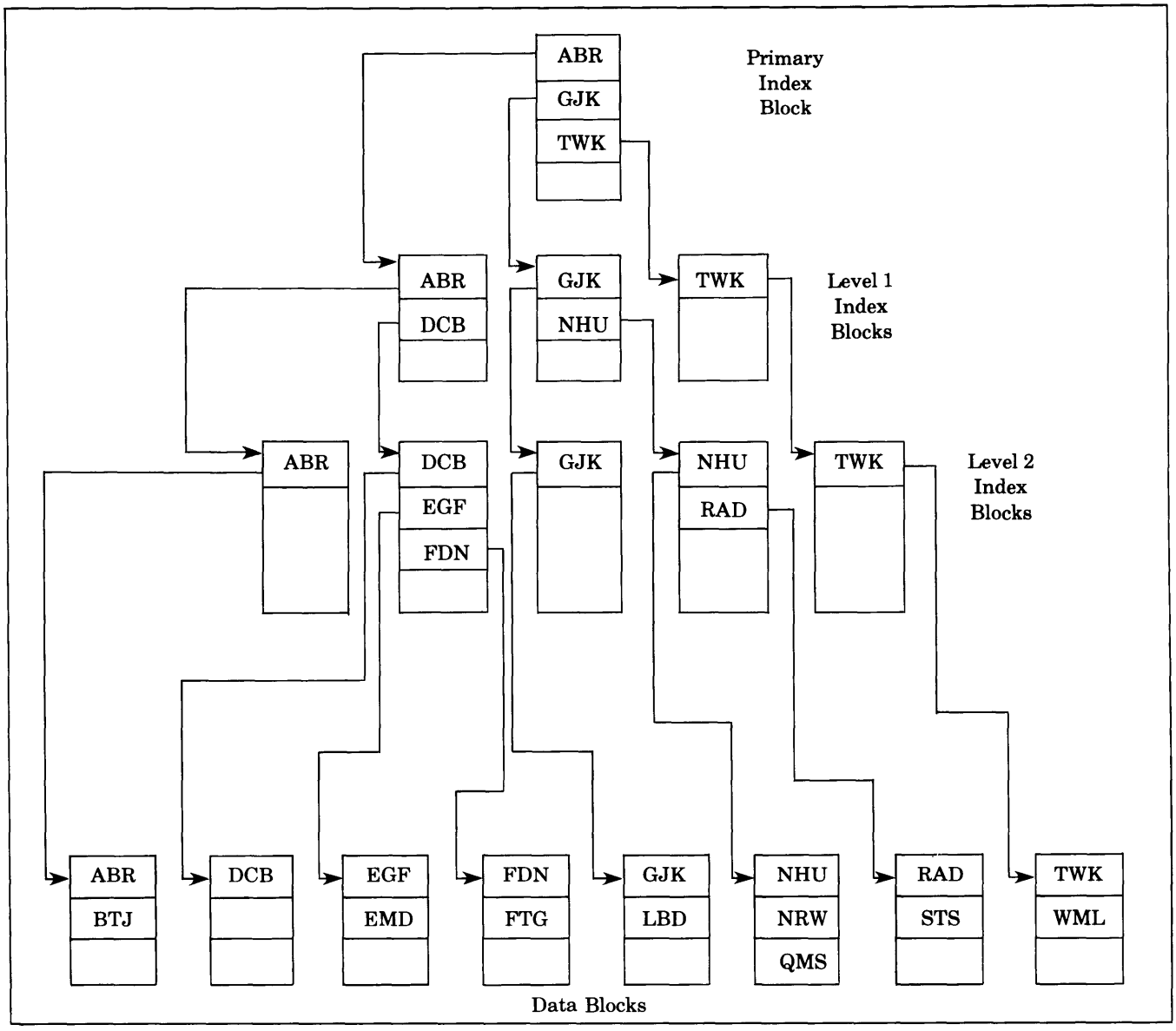
Maintaining an Indexed Sequential File

An efficient file economizes space and the time needed for record retrievals. Selecting suitable values for attributes that define file structure can help ensure that the indexed sequential file will be efficient.

The following illustration shows an example of an efficient indexed sequential file. The file has a minimum amount of extra space.



The following illustration shows an inefficient indexed sequential file. In this file there are more data blocks than are needed to contain the records in the file.



Monitoring the Growth of a File

After the file has been created you should monitor the growth of the file by issuing a fetch request to check the LEVELS_OF_INDEXING attribute.

The DISPLAY_FILE_ATTRIBUTE (DISFA) command displays the values of attributes both before and after the file is opened. The format is:

DISPLAY_FILE_ATTRIBUTE
FILE = list of file
DISPLAY_OPTION = keyword value
OUTPUT = file reference
STATUS = status variable

The parameters are:

FILE (F)

Name of file for which the selected attributes are to be displayed. This parameter is required.

DISPLAY OPTION or *DISPLAY_OPTIONS (DO)*

List of attributes whose values are to be displayed, or ALL.

OUTPUT (O)

File to which output is to be written. (Default is $OUTPUT.)

STATUS (ST)

Status of request.

The DISPLAY_FILE_ATTRIBUTE (DISFA) command can be used in any job. See the SCL Quick Reference manual for detailed information on the DISFA command.

Recreating a File

If your file becomes inefficient or if the current structure of the file does not match your needs, you should consider recreating the file. You can recreate the file as follows:

Using a FORTRAN source program

Using the File Management Utility (FMU)

In any case, before recreating the file you should analyze why and how the current file does not meet your needs, and use this information to recreate the file.

To recreate the file through a source program, you follow the procedure used to initially create the file.

The File Management Utility (FMU) can be used to recreate an indexed sequential file. FMU has other capabilities, such as allowing you to recreate the indexed sequential file with a different primary key. See the File Management Utility usage manual for a complete description of FMU.

To recreate a file using FMU, proceed as follows:

1. Use the SET_FILE_ATTRIBUTES (SETFA) command to describe the file attributes for the new indexed sequential file.

2. Use the FMU command to create the new indexed sequential file (output_file) from the existing indexed sequential file (input_file). The format of the command you need to use is as follows:

   FMU INPUT=input_file OUTPUT=output_file

The following example recreates an indexed sequential file:

```
    .
    .
    .
/set_file_attribute newisfile fo=is maxrl=65 kl=20 rt=f kt=uc ..
../ek=yes arl=65 kp=0 erc=1000000
/fmu i=isfile o=newisfile
    .
    .
    .
```

The SET_FILE_ATTRIBUTE command creates a new indexed sequential file with MAXIMUM_RECORD_LENGTH equal to 65, KEY_LENGTH equal to 20, RECORD_TYPE equal to fixed-length records, KEY_TYPE equal to uncollated keys, EMBEDDED_KEY equal to YES, AVERAGE_RECORD_LENGTH equal to 65, KEY_POSITION equal to 0, and ESTIMATED_RECORD_COUNT of 1000000. If a file attribute is not specified in the command, the value of the attribute defaults to a predefined value. For a complete description of the SET_FILE_ATTRIBUTE command, see the SCL System Interface manual.

Error Communication

The following file attributes are used for error communication:

ERROR_COUNT (ERC)
ERROR_EXIT_PROCEDURE (EX)
ERROR_EXIT_NAME (EXN)
ERROR_LIMIT (ERL)
ERROR_STATUS (ES)

ERROR_COUNT AND ERROR_LIMIT Attributes

For a trivial error, if the ERROR_LIMIT is greater than zero, ERROR_COUNT is incremented by one and compared with ERROR_LIMIT as follows:

If ERROR_COUNT is less than ERROR_LIMIT, control is passed to the error exit, if specified, or returned to the program containing the file interface call. If control returns to the calling program, you are responsible for checking the error status.

If ERROR_COUNT is equal to ERROR_LIMIT, ERROR_STATUS is set to 503255 (trivial error limit reached) and the FATAL/NONFATAL parameter is set to 1. Control is returned to the error exit or to the calling program.

ERROR_STATUS Attribute

The ERROR_STATUS attribute indicates the number of the last error that occurred.

If your program fetches the ERROR_STATUS attribute in-line (that is, IFETCH is not called from within the error procedure specified by the ERROR_EXIT_PROCEDURE attribute), only one error condition can be reported. The error condition is either 0 (no error occurred) or the number of the last error that occurred during the preceding file interface operation. (The file interface error messages are listed and described in appendix B.)

Note that if your program fetches ERROR_STATUS in-line, errors 502536 and 502540 can only be returned for a STOREF operation.

ERROR_EXIT_PROCEDURE Attribute

The ERROR_EXIT_PROCEDURE attribute is the name of an external subroutine. If an error occurs during execution of a file interface call, the value of the ERROR_EXIT_PROCEDURE attribute is checked and interpreted as follows:

EX=0 No user error routine; control is returned as a normal exit. The ERROR_STATUS attribute is set to an error code. If a fatal error is encountered, the message is written to file $ERRORS.

EX≠0 When a fatal or trivial error occurs, ERROR_STATUS is set to an error code, and control transfers to the procedure at address EX before returning to the calling program.

When the ERROR_EXIT_PROCEDURE attribute is set, the error procedure is called once for each error condition that occurs. At the time the error procedure is called, the ERROR_STATUS attribute contains the error number describing the current error; therefore, the error procedure can fetch the ERROR_STATUS attribute to determine the cause of the error.

FATAL/NONFATAL Parameter

You can use the FATAL/NONFATAL parameter to determine if processing is to continue after an error. For example, you may want processing to continue if a trivial error occurs. Therefore, after an error is detected, you can check the FNF parameter and continue processing if the error was trivial. The FNF parameter can have the following values:

0 A nonfatal error occurred.

1 A fatal error occurred.

Input/Output-Related Statements and Routines

This section describes the statements used to position files, to return status information about files and input/output operations, and to perform other tasks related to input/output. These statements are not intended for use with files processed by the file interface subprograms.

File Status Statements

FORTRAN provides three statements that can be used to establish, examine, or alter certain attributes of files used for input or output. These are the OPEN, INQUIRE, and CLOSE statements.

OPEN

You can use the OPEN statement to associate an existing file with a unit number, to create a new file and associate it with a unit number, or to change certain attributes of an existing file. The OPEN statement has the form:

OPEN (*UNIT=***u**, *IOSTAT=ios, ERR=sl, FILE=fin, STATUS=sta, ACCESS=acc, FORM=fm, RECL=rl, BLANK=blnk, BUFL=bl*)

u Unit identifier specifying the unit to be opened. If FILE= is also specified, the unit becomes associated with that file.

ios Integer variable that receives an error number if an error occurs during the open; zero indicates that no errors occurred.

sl Label of an executable statement to which control transfers if an error occurs during the open.

fin Character expression (31 or fewer characters) whose value is the name of the file to be opened; must be a valid NOS/VE file name. (Other elements of a NOS/VE file reference are not permitted.) Trailing blanks are removed. This file becomes associated with unit u. Default is the name derived from the unit identifier. The file is assumed to reside in the $LOCAL catalog.

sta Character expression specifying file status. Valid values are:

'OLD'	File currently exists.
'NEW'	File does not currently exist.
'SCRATCH'	Delete the file associated with unit u upon program termination or execution of CLOSE that specifies unit u; must not appear if FILE parameter is specified.
'UNKNOWN'	File status is unknown.

Default is STATUS='UNKNOWN'

acc Character expression specifying the access method for the file:

'SEQUENTIAL'	Open the file for sequential access.
'DIRECT'	Open the file for direct access.

Default is ACCESS='SEQUENTIAL'

If the file exists when the OPEN is issued, the access method must be valid for that file. (The file must have been created with the specified access method.)

fm Character expression specifying formatting option:

'FORMATTED'	Open the file for formatted input/output.
'UNFORMATTED'	Open the file for unformatted input/output.
'BUFFERED'	Open the file for buffered input/output. (CDC Extension)

Default is FORM='FORMATTED' for sequential access files, FORM='UNFORMATTED' for direct access files.

For an existing file, the specified form must be valid for that file.

rl Positive integer variable or constant specifying the record length for a direct or sequential access file. RECL is required for a direct access file; if omitted for a sequential access file, it defaults to 150 characters.

blnk Character expression specifying blank interpretation for formatted input/output:

'NULL'	Blank values in numeric formatted input fields are ignored, except that a field of all blanks is treated as zero.
'ZERO'	Blanks, other than leading blanks are treated as zeros.

Default is BLANK='NULL'

bl This parameter is provided only for compatibility with previous versions of FORTRAN, and is disregarded.

A file specified in an OPEN statement is said to be opened with the attributes specified in the OPEN statement. A file must be opened before any I/O operations can be performed on it. However, it is not always necessary to specify the file in an OPEN statement. If a unit is referenced in an input/output operation, and the file associated with that unit was not previously declared in an OPEN statement, an implicit OPEN of the file associated with the referenced unit occurs. For example, the statement:

```
OPEN (UNIT=2, FILE='AAA', ACCESS='DIRECT', RECL=120)
```

opens file AAA for direct access input/output and associates AAA with unit 2.

Files opened implicitly are assigned default attributes when the file is first referenced in an I/O statement.

The UNIT= specifier is required in an OPEN statement; all other specifiers are optional except that you must specify the RECL specifier if a file is being opened for direct access. If a STATUS of OLD or NEW is specified, you must specify a FILE= specifier.

If you omit the FILE= specifier, the file is assumed to be the one associated with the specified unit in the PROGRAM statement (described in chapter 7). If you do not specify the unit on the PROGRAM statement, the file name is derived from the unit number. For unit numbers in the range 0 through 999, the file name is TAPEn where n is the unit number; for unit numbers having the form of a one- through seven-character name, the file name will be the same as the unit number.

A declaration in an OPEN statement overrides a declaration in a preceding PROGRAM statement, providing that no input/output operations have been performed on the file. For example, in the sequence

```
PROGRAM XX (TAPE2=/500)
.
.
.
OPEN (2, RECL=1300, FILE='FILEY')
READ (2, 100) A, B, C
```

the PROGRAM statement declares a 500 character record length for unit 2; however, the OPEN statement takes precedence over the PROGRAM statement, and the 1300 character record length is used. The READ statement reads data from FILEY.

Declarations of file properties on a SETFA command override any conflicting OPEN statement parameters for a unit associated with that file; this applies to all OPEN statements for that unit. For example, a PAGE_WIDTH parameter on a SETFA command overrides the RECL parameter value specified in an OPEN statement.

When a file is created (opened for the first time), the attributes established by the OPEN statement are permanent. A subsequent open of the file uses the original attributes. The only specifiers that can be changed are the BLANK= and UNIT= specifiers. In order to define file attributes, you must specify all of the desired attributes in the first OPEN statement (or in a SETFA command prior to execution).

Once a file has been associated with a particular unit, the file can be associated with another unit in a subsequent OPEN statement. The file is then associated with more than one unit. In this case the unit numbers refer to the same file. Actions taken on one unit also affect the other unit. For example, closing a unit closes all other units associated with the same file.

Example:

```
OPEN (2, FILE='INFIL')
.
.
.
OPEN (3, FILE='INFIL')
READ (2, 100) A, B
READ (3, 100) X, Y
```

Both READ statements read from file INFIL.

Example:

```
OPEN (3, FILE='XXX', STATUS='OLD', BLANK='ZERO')
```

When data is read from the existing file XXX, blanks will be interpreted as zeros.

Example:

```
OPEN (2, STATUS='NEW', ERR=12, FILE='NEWFL', ACCESS='SEQUENTIAL')
```

A new file, NEWFL, is associated with unit 2 and is to be a sequential access file.

If a file is associated with a unit and a succeeding OPEN statement associates a different file with the same unit, the effect is the same as performing a CLOSE without a STATUS= specifier on the currently associated file before associating the new file with the unit. For example, in the sequence

```
OPEN (2, FILE='MYFILE')
WRITE (2, '(A)') A, B, C
OPEN (2, FILE='PART2')
```

the second OPEN statement implicitly closes MYFILE before opening PART2.

When opening a standard system file, such as $INPUT or $OUTPUT, you must specify STATUS='UNKNOWN' (or omit the STATUS= specifier because the default is 'UNKNOWN') to avoid a runtime error.

CLOSE

The CLOSE statement disassociates a file from a specified unit and specifies whether the file associated with that unit is to be kept or released. The CLOSE statement has the form:

CLOSE (*UNIT*=**u**, *IOSTAT=ios, ERR=sl, STATUS=sta*)

u Unit identifier of the file to be closed.

ios Integer variable in which, upon completion of the close, contains an error number if errors occurred; 0 indicates no errors occurred.

sl Label of an executable statement to which control transfers if an error occurs during the close.

sta Character expression that specifies the disposition of the file after the close:

'KEEP'	The file is kept after the close.
'DELETE'	The file is detached after the close.

Default is STATUS='DELETE' if file status is 'SCRATCH'; otherwise, the default is STATUS='KEEP'

'KEEP' is not valid for a file whose status is 'SCRATCH'

A CLOSE statement can appear in any program unit in the program; it need not appear in the same program unit as the OPEN statement specifying the same unit.

A CLOSE statement that references a unit having no file associated with it has no effect.

After disassociating a unit by a CLOSE statement, you can associate it again within the same program to the same file or to a different file. A file associated with a unit specified in a CLOSE statement can be subsequently associated with the same unit or with another unit, provided the file still exists.

Unit equivalence established on the PROGRAM statement and file association established on the execution command are no longer in effect after the CLOSE statement is executed.

When a program terminates normally, an implicit CLOSE (u, STATUS='KEEP') occurs for each opened unit unless the status of the file was SCRATCH; in this case, a CLOSE (u, STATUS='DELETE') occurs.

When closing a standard system file, such as $INPUT or $OUTPUT, specifying STATUS='DELETE' causes a run-time error, because a standard system file cannot be detached from the job.

Example:

```
CLOSE (2, ERR=25, STATUS='DELETE')
```

INQUIRE

The INQUIRE statement returns information about a specified file or unit. This statement has the form:

INQUIRE (unfl, *IOSTAT=ios, ERR=sl, EXIST=ex, OPENED=od, NUMBER=num, NAMED=nmd, NAME=fn, ACCESS=acc, SEQUENTIAL=seq, DIRECT=dir, FORM=fm, FORMATTED=fmt, UNFORMATTED=unf, RECL=rcl, NEXTREC=nr, BLANK=blnk***)**

unfl Specifies the file or unit for which information is to be returned; unfl has one of the following forms:

UNIT=**u**	Inquire by unit; u is a unit identifier.
FILE=fin	Inquire by file; fin is a character expression whose value is a valid NOS/VE file name (other elements of a file reference are not permitted).

ios Integer variable which, upon completion of the INQUIRE, contains an error number; contains 0 if no errors occurred.

sl Label of an executable statement to which control passes if an error occurs during an inquire.

ex Logical variable that receives a value indicating whether the file or unit is valid:

.TRUE.	For inquire-by-unit, the unit is a valid unit. For inquire-by-file, the file is local to the job and contains data.
.FALSE.	The file or unit is not valid.

od Logical variable that receives one of the following values:

.TRUE.	The file (unit) is associated with a unit (file).
.FALSE.	The file (unit) is not associated with a unit (file).

num An integer variable that receives the unit number of the unit currently associated with the file; undefined if the file is not associated with a unit.

nmd Logical variable that receives one of the following values:

.TRUE.	The file has a name.
.FALSE.	The file does not have a name.

fn A character variable that receives the name of the file associated with unit u.

acc A character variable that receives a value indicating the access method of the file:

'SEQUENTIAL'	The file is opened for sequential access input/output.
'DIRECT'	The file is opened for direct access input/output.

If the file is not open, acc is undefined.

seq A character variable indicating whether the file can be opened for sequential access input/output:

'YES'	The file can be opened for sequential access input/output.
'NO'	The file cannot be opened for sequential access input/output.
'UNKNOWN'	It cannot be determined if the file can be opened for sequential access input/output.

dir A character variable indicating whether the file can be opened for direct access input/output:

'YES'	The file can be opened for direct access input/output.
'NO'	The file cannot be opened for direct access input/output.
'UNKNOWN'	It cannot be determined if the file can be opened for direct access input/output.

fm A character variable indicating whether the file is opened for formatted or unformatted input/output:

'FORMATTED'	The file is opened for formatted input/output.
'UNFORMATTED'	The file is opened for unformatted input/output.
'BUFFERED'	The file is opened for buffered input/output. (CDC Extension)

If the file has not been opened, fm is undefined.

fmt A character variable indicating whether the file can be opened for formatted input/output:

'YES'	The file can be opened for formatted input/output.
'NO'	The file cannot be opened for formatted input/output.
'UNKNOWN'	It cannot be determined if the file can be opened for formatted input/output.

unf A character variable indicating whether the file can be opened for unformatted input/output:

'YES'	The file can be opened for unformatted input/output.
'NO'	The file cannot be opened for unformatted input/output.
'UNKNOWN'	It cannot be determined if the file can be opened for unformatted input/output.

rcl An integer variable that receives the record length of a file opened for direct access. If the file is 'FORMATTED', rcl contains the record length in characters; if 'UNFORMATTED', the record length is in words; rcl is undefined if the file is not opened for direct access.

nr An integer variable; for a direct access file, nr receives the record number of the next record to be read or written. If no records have been read or written, 1 is returned. Undefined for sequential access files.

blnk A character variable indicating blank interpretation currently in effect for a file opened for formatted input/output:

'NULL'	Null blank control is in effect.
'ZERO'	Zero blank control is in effect.

Undefined if the file is not opened for formatted input/output.

There are two forms of the INQUIRE statement: inquire by unit is used to obtain information about the current status of a specified unit; inquire by file is used to obtain information about the current status of a file.

You must specify either a file name (inquire by file) or a unit specifier (inquire by unit), but not both, in an INQUIRE statement. The file or unit need not exist when INQUIRE is executed. 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 following specifiers will contain information about the file associated with the unit (provided that any conditions stated in the specifier descriptions hold):

NAMED	NAME	ACCESS
SEQUENTIAL	DIRECT	FORM
FORMATTED	UNFORMATTED	RECL
OPENED	NEXTREC	EXIST
NUMBER	ACCESS	BLANK

If a file name is specified, following parameters will contain information about the file and the unit with which it is associated (provided that any conditions stated in the specifier descriptions hold):

NAMED	NAME	SEQUENTIAL
DIRECT	FORMATTED	UNFORMATTED
OPENED	EXIST	NUMBER
ACCESS	FORM	RECL
NEXTREC	BLANK	

If a file is specified that is associated with more than one unit, the NUMBER parameter will contain one of the unit numbers or names.

If you specify an invalid file or unit, no error results but certain parameters are not assigned values. Note that if you specify a unit that is not associated with a file, only the OPENED, IOSTAT and EXIST specifiers receive values.

If an error occurs during an INQUIRE, only IOSTAT contains a value.

Example:

```
LOGICAL EX
CHARACTER*10 AC
.
.
.
INQUIRE (FILE='AFILE', ERR=100, EXIST=EX, ACCESS=AC)
```

Status information for file AFILE is returned in the variables EX and AC. If an error occurs during the INQUIRE, control transfers to statement 100.

File Positioning Statements

Three statements are provided to position files opened for sequential access: REWIND, BACKSPACE, and ENDFILE. You cannot use these statements on files open for direct access.

REWIND

The REWIND statement positions a file at beginning-of-information. This statement has the form:

REWIND (*UNIT*=**u**, *IOSTAT*=*ios*, *ERR*=*sl*)

REWIND u

u	External unit identifier. The characters UNIT= are optional.
ios	An integer variable which returns an error number; a value of 0 indicates no errors occurred.
sl	Label of an executable statement to which control transfers if an error occurs during the rewind.

The next input/output operation after a rewind references the first record in the file. If the file is already at beginning-of-information, no action is taken.

Example:

```
REWIND 3
```

The file associated with unit 3 is positioned at beginning-of-information.

BACKSPACE

The BACKSPACE statement positions the file associated with the specified unit one record in a backward direction (toward beginning-of-information). This statement has the form:

BACKSPACE (*UNIT*=**u**, *IOSTAT*=*ios*, *ERR*=*sl*)

BACKSPACE u

u, ios, and sl are as described for REWIND.

If the file is already positioned at beginning-of-information, no action is taken. You cannot not use backspace operations on direct access files or on records created by list directed or namelist output.

Example:

```
      DO 1 LUN = 1, 4
1     BACKSPACE LUN
```

The files associated with units 1 through 4 are backspaced one record.

ENDFILE

The ENDFILE statement writes an end-of-partition on the file associated with the specified unit. This statement has the form:

ENDFILE (*UNIT*=**u**, *IOSTAT*=*ios*, *ERR*=*sl*)

ENDFILE u

u, ios, and sl are as described for REWIND.

ENDFILE should not be the first operation on a file. The end-of-partition can be detected by the END= and IOSTAT= specifiers.

The following restrictions apply to ENDFILE:

ENDFILE is not permitted on units opened for direct access.

ENDFILE cannot not be used on a file processed by mass storage subroutines.

ENDFILE can only be used on files with V (variable) type records; use of ENDFILE on other record types causes an error.

Example:

```
IOUT = 7
ENDFILE (UNIT=IOUT, ERR=100)
```

An end-of-partition boundary is written on the file associated with unit 7.

Input/Output Status Checking Routines (CDC Extension)

Status checking for input/output statements such as READ and WRITE should be done with the optional specifiers IOSTAT= or END= , but can also be done with the functions UNIT, EOF, and IOCHEC. UNIT and EOF return an end-of-file status if an end-of-partition or end-of-information was read by the previous read operation.

The functions UNIT and IOCHEC return a parity error indication for every record within or spanning a block containing a parity error; such an indication, however, does not necessarily refer to the immediately preceding operation because of the record blocking/deblocking performed by the internal input/output routines.

UNIT

The UNIT function is used to check the status of a BUFFER IN or BUFFER OUT operation for an end-of-file or parity error condition on logical unit u. The UNIT function reference has the form:

UNIT (**u,** a, b)

- **u** Unit identifier.
- *a* First variable or array element of the block of memory specified in the preceding BUFFER IN or BUFFER OUT statement.
- *b* Last variable or array element of the block of memory specified in the preceding BUFFER IN or BUFFER OUT statement.

The function returns one of the following type real values:

-1. Unit ready, no end-of-file or parity error encountered on the previous operation.

0. Unit ready, end-of-file encountered on the previous operation.

+1. Unit ready, parity error encountered on the previous operation.

Although the arguments a and b are optional, you should always include them if you have selected OPT=HIGH on the FORTRAN command. Specifying a and b enables the FORTRAN compiler to associate the call to UNIT with possible changes to the values in the locations between a and b. If you call UNIT with only the argument u (as in most older programs), the compiler might not detect that values between a and b are being referenced while instructions between the BUFFER IN or BUFFER OUT statement and the UNIT call are executing. This could lead to execution errors.

Example:

```
BUFFER IN (5,1) (B(1), B(100))
IF (UNIT(5, B(1), B(100)) 12, 14, 16
```

Control transfers to the statement labeled 12, 14, or 16 if the value returned was -1., 0., or +1., respectively.

If 0. or +1. is returned, the condition indicator is cleared before control is returned to the program. UNIT should be called only for a file processed by BUFFER statements.

EOF

The EOF function is used to test for an end-of-file on a unit following a formatted, list directed, namelist, or unformatted sequential read. The EOF function reference has the form:

EOF (u)

u Unit identifier

Zero is returned if no end-of-file is encountered; a nonzero value is returned if an end-of-file is encountered. If an end-of-file is encountered, EOF clears the condition indicator before returning control.

Example:

```
IF (EOF(5) .NE. 0) GO TO 20
```

Control transfers to statement 20 if an end-of-file is encountered on unit 5.

The EOF function is provided for compatibility with previous systems, and is not intended to replace the END= or IOSTAT= specifiers in a READ statement. If the IOSTAT= or END= specifier is not used in the READ statement that reads the end-of-file, the program will be terminated by a fatal error before the subsequent EOF call is executed.

The EOF function should not be used to test for an endfile condition following read or write operations on random access files (files accessed by READMS/WRITMS) or following write operations on all types of files, because a zero value is always returned regardless of whether an end-of-file is detected.

The EOF function is of type real.

IOCHEC

The IOCHEC function tests for a parity error on a unit following a formatted, list directed, namelist, or unformatted read. The IOCHEC function reference has the form:

IOCHEC (u)

u Unit identifier

Zero is returned if no error was detected. If a parity error occurs, IOCHEC clears the parity indicator before returning control.

The IOCHEC function is of type integer.

Example:

```
      READ (UNIT=6, END=99, ERR=88) A
88    J=IOCHEC(6)
      IF (J .NE. 0) GO TO 25
```

If no parity error occurs during the READ (IOCHEC returns zero) execution continues with the statement following the IF. If a parity error is detected, control transfers to statement 25.

LENGTH

The LENGTH function or LENGTHX subroutine returns information regarding the previous BUFFER IN or READMS of the specified unit. The LENGTH reference and LENGTHX call have the forms:

LENGTH (u)

CALL LENGTHX (u, nw, ubc)

u Unit identifier.

nw Integer variable or array element to receive the number of words read.

ubc Integer variable or array element to receive the number of unused bits in the last word of the transfer. The value is rounded down to an integral number of bytes.

The values returned by LENGTHX and the value of LENGTH are all of type INTEGER.

For a file accessed by buffer statements, LENGTH or LENGTHX should be called only after a call to UNIT ensures that input/output activity is complete; otherwise, file integrity might be endangered.

Internal Data Transfer Routines

Two routines are provided for moving blocks of data from one area of memory to another.

MOVLEV

The MOVLEV call transfers blocks of noncharacter data from one area of memory to another. This call has the form:

CALL MOVLEV (a, b, n)

a Variable or array element indicating the starting location of the data to be moved.

b Variable or array element indicating the starting location of the area to receive the data.

n Integer expression specifying the number of words to be moved.

No conversion is done by MOVLEV. For instance, if data from a real variable is moved to a type integer receiving field, the result is undefined.

The block of data to be moved should be contained in one variable, array, equivalence class, or common block, and the receiving area should be contained in one variable, array, equivalence class, or common block. Otherwise, the result of the move is undefined. Also, the blocks should not overlap.

The arguments a and b must not be type character; for character data, subroutine MOVLCH should be used.

Example:

```
CALL MOVLEV (A, I, 1000)
```

1000 words are moved from locations starting with A to locations starting with I. No conversion is performed.

Example:

```
DOUBLE PRECISION D1(500), D2(500)
CALL MOVLEV (D1, D2, 1000)
```

Because array D1 is defined as double precision, the word count is set to twice the size of D1 so that the entire array is moved.

MOVLCH

The MOVLCH call transfers character data from one area of memory to another. This call has the form:

CALL MOVLCH (a, b, n)

a Variable, array element, or substring name indicating the starting location of the character string to be moved.

b Variable, array element, or substring name indicating the starting location of the receiving area.

n Integer expression specifying the number of characters to be moved.

The block of data to be moved should be contained in one variable, array, equivalence class, or common block, and the receiving area should be contained in one variable, array, equivalence class, or common block. Otherwise, the result of the move is undefined. Also, the blocks should not overlap.

Both arguments a and b must be of type character; a diagnostic is issued if one, or both, is of any other type. Note that moving a single character variable or array element to another single character variable or array element is easier to do with a character assignment statement.

Example:

```
CHARACTER*123, CH1(10), CH2(5)
CALL MOVLCH (CH1(8), CH2(3), 369)
```

The last three elements of character array CH1 are moved to the last three elements of character array CH2. Because each element is 123 characters long, the character count is set to 3 * 123 = 369.

File Connection Routines

Two routines are provided for connecting and disconnecting a file from the terminal.

CONNEC

The CONNEC call connects a file to the terminal. This call has the form:

CALL CONNEC (u, *cs***)**

u Unit identifier.

cs Character set designator; an integer expression having one of the following values:

0	ASCII-128
1	ASCII-128
2	ASCII-256

If cs is omitted, the character set defaults to ASCII-128.

If a program to be run interactively calls for input/output operations through a remote terminal, all files to be accessed through the terminal must be formally associated with the terminal when the files are referenced.

Files $INPUT and $OUTPUT are automatically connected to the terminal. (However, you can override this connection.) Thus, whenever data is read from file $INPUT, the data must be entered through the terminal. Whenever data is written to file $OUTPUT, the data is displayed at the terminal.

You can connect any file from within the program by using the CALL CONNEC statement. You can also connect a file by specifying a REQUEST_TERMINAL command (described in the SCL System Interface manual).

If a unit specified in a CONNEC call is associated with a file that exists as a non-terminal file but is not connected at the time of the call, the file's contents are lost.

Data input or output through a terminal is represented in the ASCII 128-character set shown in appendix B.

DISCON

The DISCON call disconnects a file from the terminal. This call has the form:

CALL DISCON (u)

u Unit identifier

The file associated with the specified unit is detached, and data written to the file is lost unless the file is permanent. A DISCON call is ignored if the file is not connected.

Program Units 7

An executable FORTRAN program consists of one or more program units with the restriction that exactly one must be a main program unit (usually called simply a main program).

A program unit is a group of FORTRAN statements, with optional comments, terminated by an END statement. The types of program units are main programs and subprograms. Subprograms can be subroutine subprograms, function subprograms, or block data subprograms.

Figure 7-1 illustrates a FORTRAN program consisting of three program units: a main program named AVG, a subroutine subprogram named DIVIDE, and a function subprogram named NADD.

```
      PROGRAM AVG
      INTEGER NUMBER(10), NSUM
      REAL RESULT
      OPEN (1, FILE='DATA')
      READ (1, 100) (NUMBER(I), I=1,10)
      NSUM = NADD(NUMBER)
      CALL DIVIDE(NSUM,10,RESULT)
      PRINT 200, (NUMBER(I), I=1,10), RESULT
      STOP
100   FORMAT (10I1)
200   FORMAT ('1THE AVERAGE OF ',/10(' ',I1),/,' 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 7-1. Main Program, Function, and Subroutine Example

Subroutine subprograms, function subprograms, intrinsic functions, and statement functions are referred to as procedures. The only subprogram that is not a procedure is a block data subprogram, which is not executable.

Statement functions are the only procedures that cannot be compiled independently of other procedures. Statement functions are defined within program units and are compiled in-line.

Subroutine subprograms can be written by the programmer. In addition, the FORTRAN library provides function and subroutine subprograms that are of general utility and which can be referenced in any FORTRAN program. The FORTRAN-supplied subprograms are described in chapter 9.

A function subprogram can be an external function or an intrinsic function. Intrinsic functions are supplied by the FORTRAN library and can be referenced by any FORTRAN program. The intrinsic functions are described in chapter 8. External functions are provided by the programmer.

The following table summarizes the characteristics of program units:

Program Unit	Characteristics	How Identified
Main program	Defines main entry point for FORTRAN program. Every program must have a main program unit.	Usually begins with PROGRAM statement.
Subroutine	Can be called from other program units in program. Returns values through argument list or common.	Begins with FUNCTION statement.
External function	Can be called from other program units. Return single value through function name. Can also communicate through argument list and common.	Begins with FUNCTION statement.
Statement function (not a program unit)	Calculates a single result for a program unit; cannot be referenced outside defining program unit.	Defined within a program unit; single statement definition.
Block data subprogram	Provides initial values for named common blocks.	Begins with BLOCK DATA statement.

Main Programs

A main program is a program unit that does not begin with a SUBROUTINE, FUNCTION, or BLOCK DATA statement. Usually, a main program begins with a PROGRAM statement, but this statement is optional. Execution of any FORTRAN program begins with the main program unit.

A main program can contain any FORTRAN statements except FUNCTION, SUBROUTINE, BLOCK DATA, or ENTRY. The main program should have a PROGRAM statement and at least one executable statement, and it must have an END statement as the last statement. An executable program should not have more than one main program unit. (If more than one main program exists, the last one loaded is used.)

The main program can be compiled independently of any subprograms. However, when a main program is loaded into memory for execution, all the required subprograms must also be loaded and ready for execution.

Figure 7-1 shows an example of a main program. The program, named AVG, reads numbers from file DATA and calls function NADD and subroutine DIVIDE to perform calculations.

PROGRAM Statement

The PROGRAM statement defines the name that is used as the entry point name and as the object program name for the loader. The PROGRAM statement also declares certain properties of input/output units to be used by the program. (Unit declaration on the PROGRAM statement is provided for compatibility with previous versions of FORTRAN. Use of the OPEN statement is preferred.) The PROGRAM statement has the forms:

PROGRAM name

PROGRAM name (upar, ..., *upar*) (CDC Extension)

name Program name. If the PROGRAM statement is omitted, name defaults to START#.

upar Declares an input/output unit in one of the following forms:

unit

Name of a unit to be used by the main program or its subprograms; one through seven characters. Maximum number of units is 49.

unit=n

This form is provided for compatibility with previous versions of FORTRAN, and is interpreted as if =n had been omitted.

unit=/r

Specifies the maximum record length in characters for list directed, formatted, and namelist lines; default length is 150 characters. Maximum value is 65535 characters.

unit=n/r

This form is provided for compatibility with previous versions of FORTRAN; n is disregarded and r is as described above.

altunit=unit

Altunit and unit are unit names; altunit is usually in the form TAPEu, where u is an integer in the range 0 through 999. This form specifies that the unit names are equivalent. The record length is as previously specified (or defaulted) for the unit.

Example:

```
PROGRAM PROGA (AFILE,TAPE2=AFILE)
```

This statement assigns the name PROGA to the program, and equivalences the name TAPE2 to AFILE. When any input/output statement in the program references unit 2, the reference applies to unit AFILE.

Example:

```
PROGRAM FIRST
```

This statement assigns the name FIRST to the program. The parameter list is legally omitted.

PROGRAM Statement Usage (Does Not Apply to ANSI FORTRAN)

The PROGRAM statement can declare units that are used in the program and in any subprograms that are called. If you omit this statement from the main program, the program is assumed to have the name START#.

Units referenced in input/output statements need not be declared in a PROGRAM statement. Alternatively, units can be declared in an OPEN statement. If a unit is not declared on the PROGRAM statement or in an OPEN statement, an implicit open occurs on the first reference to the unit.

In most cases, the OPEN statement is preferred over PROGRAM statement unit declarations because OPEN provides more options, more flexibility, and is consistent with ANSI FORTRAN.

In the absence of any other specification of the file name, FORTRAN adds the characters TAPE as a prefix to the unit number to form the file name. For instance, TAPE3 is the file name assigned to unit number 3 and TAPE5 is the file name assigned to unit number 5. TAPE5 and TAPE05 do not specify the same file name.

When unit names are made equivalent, the record length specified for the unit to the right of the equal sign also applies to the specified alternate unit. Therefore, any attempt to specify record length for the alternate unit results in an error.

Subprograms

A subprogram is a program unit that is physically included only once in a FORTRAN program but can be executed many times. It can be called from the main program or from another subprogram.

A subprogram is subordinate to a main program; that is, a subprogram cannot be executed independently of a main program. It is defined separately and can be compiled independently of a main program. However, a subprogram can be executed only after the main program begins execution. You generally write subprograms to perform calculations that are required repeatedly by a program.

The types of subprograms are subroutines, functions (external and intrinsic), and block data subprograms.

Subroutine Subprogram

A subroutine subprogram begins with a SUBROUTINE statement and ends with an END statement. The SUBROUTINE statement has the form:

SUBROUTINE name *(d, ..., d)*

name	Name of the subroutine subprogram
d	Dummy argument that can be a variable name, array name, dummy procedure name, or asterisk

If there are no dummy arguments, you can specify either SUBROUTINE name () or SUBROUTINE name.

The SUBROUTINE statement must appear as the first statement of the subroutine subprogram and contains the symbolic name that is the main entry point of the subprogram. The name must not be the same as any other program unit or entry name. Also, the name cannot be the same as any other symbolic name in the subroutine.

Examples:

```
SUBROUTINE GETVAL(X(1),VAR,RVAL,N,Z)

SUBROUTINE ARC
```

In the last example, the argument list is legally omitted.

Subroutines can contain any statements except a PROGRAM, BLOCK DATA, FUNCTION, or another SUBROUTINE statement. Control is returned to the calling program unit when a RETURN or END statement is encountered.

Subroutines can communicate with other program units through the argument list. The arguments specified in the subroutine argument list are called dummy arguments, and are associated with actual arguments specified in the CALL statement that calls the subroutine. Subroutines can also communicate with other program units through common blocks. (See Procedure Communication.)

Figure 7-1 shows an example of a subroutine subprogram. The subprogram, named DIVIDE, receives values through the argument list. The dummy arguments are N, I, and Q. When the subroutine is called in the main program by the CALL DIVIDE statement, the actual arguments NSUM, 10, and RESULT are associated with the dummy arguments. The result of the subroutine is stored in the dummy argument Q which is associated with the actual argument RESULT in the main program.

Function Subprogram

Function subprograms can be external functions or intrinsic functions. Both external and intrinsic functions are specified externally from the program unit that references them.

External Functions

An external function begins with a FUNCTION statement. This statement has the form:

type **FUNCTION name** (*d, ..., d*)

type	Type indicator. Can be INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, BOOLEAN, or CHARACTER*len, where len is the length of the result of the character function. If omitted, the function name determines the type according to the rules described in chapter 3.
name	Name of the function subprogram.
d	Dummy argument that can be a variable name, array name, or dummy procedure name.

A function name must not be the same as any other name, except a variable name or common block name.

The function statement specifies the symbolic name that is used as the main entry point of the function. You can specify additional entry points using ENTRY statements (see Multiple Entry Points).

Examples of FUNCTION statements:

```
FUNCTION FN(X)

INTEGER FUNCTION DZ(V1,V2,V3)

CHARACTER*3 FUNCTION VCHARS (CH)

CHARACTER*(*) FUNCTION APPLE (TXT)
```

The last statement declares a character function to have the length specified by the calling program unit. The length of the value returned is determined by the length declared for function APPLE in the referencing program unit.

Control transfers to a function subprogram when an expression containing the function name is executed in a program unit. Control returns to the calling program unit when a RETURN or END statement is executed in the function.

A function subprogram returns a single value to the calling program unit through the function name. At some point within the function, an assignment statement must be executed that defines the function name (or an entry name of the same type as the function name). The function name can be referenced or redefined later in the function. When control returns to the calling program unit, the value of the function reference is the value defined for the function name.

A function subprogram can also communicate with the referencing program unit through a list of arguments or through common blocks.

The name specified in a FUNCTION or ENTRY statement must not appear in any other nonexecutable statement, except a type statement. If the type of a function is specified in a FUNCTION statement, then the function name cannot appear in a type statement.

The type of the function name must be the same in the referencing program unit and the referenced function subprogram. In the absence of explicit typing, the type of the function is determined by the first character of the function name. Implicit typing by the IMPLICIT statement takes effect only when the function name is not explicitly typed. The name cannot have its type explicitly specified more than once.

If the function name is of type character, then each entry name must be type character and vice versa. The function name and entry names must have the same length. For example, if the function name has a length of (*), all entry names must have a length of (*).

The length declared for a type character function name must be the same in the referencing program unit and in the referenced function.

Function subprograms can contain any statements except PROGRAM, BLOCK DATA, SUBROUTINE, or another FUNCTION statement. They begin with a FUNCTION statement and end with an END statement. Control is returned to the referencing program unit when a RETURN or END is encountered; a RETURN statement of the form RETURN exp (described under Referencing a Subprogram) in a function subprogram is not allowed.

An example of a function subprogram is shown in figure 7-1. The function, named NADD, receives an array IARRAY through the argument list. Function NADD sums the values in the array and assigns the result to the function name. In the main program the function is referenced by the statement:

```
NSUM = NADD(NUMBER)
```

The actual argument NUMBER is associated with the dummy argument IARRAY. When execution of the function is complete, the result is stored in the variable NSUM and execution continues with the next statement.

Intrinsic Functions

Intrinsic functions are supplied by the FORTRAN library. The rules for using intrinsic functions are the same as for user-written external functions. An IMPLICIT statement does not change the type of an intrinsic function. Chapter 8 describes the intrinsic functions, including generic and specific names, function definitions, type of arguments, and type of results.

Block Data Subprogram

A block data subprogram is a nonexecutable specification subprogram that can be used to specify initial values for variables and array elements in named common blocks. A program can have more than one block data subprogram.

A block data subprogram is identified by a BLOCK DATA statement. This statement has the form:

BLOCK DATA *name*

name Name of the block data subprogram. If omitted, name defaults to BLKDAT# (CDC FORTRAN only).

The BLOCK DATA statement must appear as the first statement of the block data subprogram. The name used for the block data subprogram must not be the same as any variables or symbolic constants in the subprogram. The name must not be the same as any other program unit or entry name in the program. Only one block data subprogram can be unnamed.

Block data subprograms can contain IMPLICIT, PARAMETER, DIMENSION, type, COMMON, SAVE, EQUIVALENCE, LEVEL, or DATA statements. A block data subprogram ends with an END statement. Data can be entered into more than one common block in a block data program. All variables having storage in the named common must be specified even if they are not all assigned initial values.

Example:

```
BLOCK DATA ANAME
COMMON /CAT/X,Y,Z /DOG/R,S,T
COMPLEX X,Y
DATA X,Y /2*(1.0,2.7)/, R/7.6543/
END
```

The block data subprogram ANAME enters data into common blocks CAT and DOG. Initial values are defined for variables X and Y in block CAT and variable R in block DOG. No initial values are defined for variables Z, S, or T.

Statement Functions

A statement function consists of a single statement and calculates a single result. It is defined by the programmer, and applies only to the program unit containing the statement. The general form of a statement function is:

name (d, ..., *d*) = exp

name	Function name
d	Dummy argument
exp	Expression

A statement function is a nonexecutable statement. It must appear after the specification statements and before the first executable statement in the program unit.

Examples:

```
ADD(A,B,C,D) = A + C + B + D

AVRG(SUM,N) = SUM/N
```

ADD is a statement function with dummy arguments A, B, C, and D. AVRG is a statement function with dummy arguments SUM and N.

A statement function name is defined with the value of the expression after execution. During execution, the actual argument expressions are evaluated, converted if necessary to the types of the corresponding dummy arguments according to the rules for assignment (ANSI FORTRAN performs no conversion), and passed to the function. Thus, an actual argument cannot be an array name or a function name. In addition, if a character variable or array element is used as an actual argument, a substring reference to the corresponding dummy argument must not be specified in the statement function expression. The expression of the function is evaluated, and the resulting value is converted as necessary to the data type of the function.

A statement function name must not be the same as an array, variable, symbolic constant, intrinsic function, or dummy procedure name in the same program unit. The function name cannot be an actual argument and must not appear in an INTRINSIC or EXTERNAL statement. If you use the statement function in a function subprogram, then the statement function can contain a reference to the name of the function subprogram or any of its entry names as a variable, but not as a function.

Each variable reference in the expression can be either a reference to a variable within the same program unit or to a dummy argument of the statement function. Statement functions can reference dummy arguments that appear in a SUBROUTINE, FUNCTION, or ENTRY statement. Statement function dummy arguments can have the same names as variables defined elsewhere in the same program unit without conflict. Any reference to the name inside the function refers to the dummy argument, and any reference to the name outside the function definition refers to the variable.

Procedure Communication

Communication between the referencing program unit and the referenced procedure can be through common blocks or by passing actual arguments to the procedure. (You cannot use common blocks to pass data to intrinsic functions or statement functions; data is passed to these functions through an argument list.) You can use common blocks and argument lists to pass data to external (user-written) procedures, but procedure names can be passed to external procedures only through an argument list.

CALL Statement

A subroutine subprogram is executed when a CALL statement naming the subroutine is encountered in a program unit. The CALL statement has the form:

CALL name (*a*, ..., *a*)

name Subroutine name; cannot be the same as a variable name in the calling program unit.

a Actual argument that can be one of the following:

- A constant (including a symbolic constant)
- A variable, array, or array element name
- An expression with operators (except for a character expression involving concatenation of a dummy argument with length (*))
- An intrinsic function name
- An external subroutine or function name
- A dummy subroutine or function name
- An alternate return specifier of the form *sl, where sl is the label of an executable statement that appears in the same program unit as the CALL statement

If the subroutine has no argument list, the form CALL name () or CALL name can be used.

Note that a CALL statement can be used to call subprograms written in languages other than FORTRAN.

The CALL statement can contain actual arguments and statement labels which must correspond in order, number, and type to those in the subroutine definition. An actual argument of type boolean can have a corresponding dummy argument of type integer or real. An actual argument of type integer or real can have a corresponding dummy argument of type boolean.

An actual argument in a subroutine call can be a dummy argument name that appears in the dummy argument list of the subprogram containing the subroutine call. An asterisk dummy argument cannot be used as an actual argument.

A subroutine must not directly or indirectly call itself.

External Function Reference

An external function is executed when the function name is referenced in an expression. The general form of a function reference is:

name (*a*, ..., *a*)

name Function name

a Actual argument that can be one of the following:

- A constant (including a symbolic constant or an extended Hollerith constant.)
- A variable, array, or array element name
- An expression with operators (except a character expression involving concatenation of a dummy argument with length (*))
- An intrinsic function name
- An external subroutine or function name
- A dummy procedure name

A function must not directly or indirectly reference itself. The function reference can appear anywhere in an expression where an operand of the same type can be used.

The type of the function result is the type of the function name. The arguments must agree in order, number, and type with the corresponding dummy arguments. An actual argument of type boolean can have a corresponding dummy argument of type integer or real. An actual argument of type integer or real can have a corresponding dummy argument of type boolean.

Statement Function Reference

A statement function is evaluated when the name is referenced in an expression. The general form of a statement function reference is:

name (*a*, ..., *a*)

name	Statement function name
a	Actual argument that can be any expression except a character expression involving concatenation of an operand whose length specification is an asterisk unless the operand is a symbolic constant

The actual arguments are evaluated and (in CDC FORTRAN only) converted to the type of the corresponding dummy arguments; the resulting values are used in place of the corresponding dummy arguments in evaluation of the statement function expression.

A statement function must not directly or indirectly call itself. The statement function reference can appear anywhere in an expression where an operand of the same type can be used.

The type of the statement function result is the type of the statement function name. The arguments must agree in order and number with the corresponding dummy arguments.

A statement function can be referenced only in the program unit where the statement function appears.

Example:

```
F(X,Y) = SQRT(X**2 + Y**2)
.
.
.
Z = F(C(1), C(2))
```

These statements define and reference a statement function. When the function reference is executed, the actual arguments C(1) and C(2) are substituted for the dummy arguments X and Y, and the function is evaluated. The effect is the same as the following assignment statement:

```
Z = SQRT(C(1)**2 + C(2)**2)
```

Example:

```
AVRG(S,N) = S/N
.
.
.
X = A + B  + AVRG(TOT, NSUM)
```

AVRG is a statement function with dummy arguments S and N. When AVRG is referenced, actual arguments TOT and NSUM are substituted for the dummy arguments, the function is evaluated, and the result is used in the expression to calculate a value for X.

Arguments

Arguments in the argument list of a SUBROUTINE or FUNCTION statement are called dummy arguments. Arguments in the argument list of a CALL statement or function reference are called actual arguments.

The referencing program unit passes actual arguments to the referenced procedure. The procedure receives values from the actual arguments and returns values to the referencing program unit. An actual argument can be a variable name, array name, or array element reference. In addition, an actual argument can be one of the following provided that the associated dummy argument is a variable that is not defined during execution of the referenced procedure:

A constant (including an extended Hollerith constant)

A symbolic constant

A function reference

An expression involving operators

An expression enclosed in parentheses

An actual argument that is an extended Hollerith constant is treated as if the first element of a one-dimensional boolean array had been specified. However, the associated dummy argument must not be stored into during execution of the referenced procedure.

Subprograms use dummy arguments to indicate the types of actual arguments, the number of arguments, and whether each argument is a variable, array, subprogram, or statement label. Dummy arguments must conform to the following rules:

Dummy arguments for statement functions must be variables.

A dummy argument appearing in a SUBROUTINE, FUNCTION, or ENTRY statement must not appear in EQUIVALENCE, DATA, PARAMETER, SAVE, INTRINSIC, or COMMON statements except as a common block name.

Dummy arguments used in array declarations for adjustable dimensions must be type integer. Dummy arguments representing array names must be dimensioned (by a DIMENSION, type, or COMMON statement).

A dummy argument that is used as if it were a function or subroutine name is called a dummy procedure name. The actual argument associated with a dummy procedure name must be an intrinsic function, external function, or another dummy procedure.

Argument Association

When a procedure is referenced, the actual arguments and dummy arguments are matched and each actual argument replaces a corresponding dummy argument. The first (leftmost) actual argument is matched with the first dummy argument, the second actual argument is matched with the second dummy argument, and so forth. The number of actual arguments and dummy arguments must be the same, and each actual argument must have the same type as its corresponding dummy argument, except that a boolean actual argument can correspond to a dummy argument having any of the arithmetic types. If the actual argument is an expression, substring reference, or array subscript, it is evaluated before the arguments are associated. If the actual argument is a subprogram name, the subprogram must be available for execution at the time of the reference to the subprogram.

A dummy argument is undefined unless it is associated with an actual argument. Argument association can exist at more than one level of subprogram reference and terminates within a program unit at the execution of a RETURN or END statement.

Example:

```
CALL BSUB (I+J, VAR, X(3))
.
.
.
SUBROUTINE BSUB (N, A, B)
```

The actual arguments are I+J, VAR, and X(3). When the CALL is executed, they become associated with the dummy arguments N, A, and B, respectively.

A subprogram reference can cause a dummy argument to be associated with another dummy argument in the referenced subprogram. Any dummy arguments that become associated with each other can be referenced but must not be stored into during the execution of the subprogram. For example, if a subroutine is defined as

```
SUBROUTINE ALPHA(X, Y)
```

and referenced with

```
CALL ALPHA (A, A)
```

then the dummy arguments X and Y would each be associated with the actual argument A. X and Y would be associated with each other and therefore must not be stored into.

A subroutine reference can cause a dummy argument to become associated with an entity in a common block. For example, if a subroutine contains the statements

```
SUBROUTINE ALPHA (X)
COMMON Y
```

and the referencing program unit contains

```
COMMON A
CALL ALPHA (A)
```

then the actual argument A causes the dummy argument X to become associated with Y, which is in blank common. In this case, X and Y cannot be stored into during execution of the subroutine.

Variables as Arguments

A variable in a dummy argument list can be associated with a variable, array element, substring, or expression in the actual argument list. A subprogram can define or redefine a dummy argument if the associated actual argument is a variable name, array element name, or substring name. A subprogram cannot redefine a dummy argument if the associated actual argument is a constant, a symbolic constant, a function reference, an expression using operators, or an expression enclosed in parentheses. (An attempt to do so results in a run-time error.)

Arrays as Arguments

Dummy arguments that represent array names must be dimensioned by a DIMENSION or type statement. The actual argument array and the dummy argument array can differ in the number and size of the dimensions. A dummy argument array can be associated with an actual argument that is an array, array element, or array element substring.

If the actual argument is a noncharacter array name, the size of the dummy argument array must not exceed the size of the actual argument array. Each actual argument array element is associated with the dummy argument array element that has the same subscript value as the actual argument array element.

If the actual argument is a noncharacter array element name, the size of the dummy argument cannot exceed (as+1-av), where as is the size of the actual argument array and av is the subscript value of the array element. For example, if the program unit has the following statements:

```
DIMENSION ARRAY(20)
.
.
.
CALL CHECK (ARRAY(3))
```

then the value of as is 20, and av is 3. The maximum dummy array size is 18 for the following subroutine:

```
SUBROUTINE CHECK(DUMMY)
DIMENSION DUMMY(18)
.
.
.
```

The actual argument array elements are associated with dummy argument array elements, starting with the first element passed. In the example, DUMMY(2) is associated with ARRAY(4), and DUMMY(18) is associated with ARRAY(20).

Character Arguments

When character data is passed to a subprogram, both the dummy and actual arguments must be of type character, and the length of the actual argument must be greater than or equal to the length of the dummy argument. If the length of the actual argument of type character is greater than the length of the dummy argument, only the leftmost characters of the actual argument, up to the length of the dummy argument, are associated with the dummy argument.

If a dummy argument is an array name, the length restriction applies to the entire array and not to each array element. The length of array elements in the dummy argument can be different from the length of array elements in the actual argument. The total length of the actual argument array must be greater than or equal to the total length of the dummy argument array.

When an actual argument is a character substring, the length of the actual argument is the length of the substring. If the actual argument expression involves concatenation, the sum of the lengths of the operands is the length of the actual argument.

The association for character array elements is basically the same as for noncharacter array elements. The actual argument for a character array element can be an array name, array element name, or character substring name. If the actual argument begins at character position n of an array, then the first character position of the dummy argument array becomes associated with character position n of the actual argument array, the second character position of the dummy argument array becomes associated with character position n+1 of the actual argument array, and so forth to the end of the actual argument array. For example, if a program unit has the following statements:

```
DIMENSION A(2)
CHARACTER A*2
.
.
.
CALL SUB (A)
```

and subroutine SUB has the following statements:

```
SUBROUTINE SUB(B)
DIMENSION B(2)
CHARACTER B*1
```

then the first character of A(1) corresponds to B(1) and the second character of A(1) corresponds to B(2).

Procedure Names as Arguments

You can pass subroutine names and intrinsic or external function names through the argument list. A dummy argument that corresponds to an actual argument that is a procedure name must be used as a procedure name.

If the dummy argument is used as if it were an external function, the corresponding actual argument must be an intrinsic or external function, or dummy procedure name. In this case, the type of the dummy argument must agree with the type of the result of all actual arguments that become associated with the dummy argument.

If the dummy argument has the same name as an intrinsic function but is associated with an actual argument that is an external function, the dummy argument refers to the external function, and not the intrinsic function.

If the dummy argument is referenced as a subroutine, the actual argument must be a subroutine name.

Subprogram names can be passed through more than one level of subprogram reference. At each level, the dummy argument must conform to the preceding rules.

Asterisks as Arguments

A dummy argument that is an asterisk can only appear in the argument list of a SUBROUTINE or ENTRY statement in a subroutine subprogram. The actual argument is an alternate return specifier in the CALL statement.

Common Blocks

Common blocks can be used to communicate values among program units. The variables and arrays in a common block can be defined and referenced in all program units that contain a declaration of that common block. Common blocks are described in chapter 3.

Example:

```
PROGRAM AVR
COMMON ANUM(10), STORE
.
.
.
SUBROUTINE SUM
COMMON A(10), B
.
.
.
```

The array ANUM in program AVR and the array A in subroutine SUM share the same locations in blank common. The variables STORE and B share the same location. Values stored into ANUM in the main program are available to SUM.

Example:

```
PROGRAM COM
COMMON /CB/ARR(3)
.
.
.
SUBROUTINE SUB
COMMON /CB/A, B, C
```

Variable A in SUB shares the same location as ARR(1) in the main program, B shares the same location as ARR(2), and C shares the same location as ARR(3).

A reference to data in a common block is valid if the data is defined and is the same type as the type of the name used in the main program or subprogram. The exceptions to agreement between the type in common and the type of the reference are as follows:

Either part of a complex entity can be referenced as real.

A boolean entity can be referenced as integer.

In a subprogram, entities declared in a named common block can either remain defined or become undefined at execution of an END or RETURN statement. If a named common block with the same name has been declared in a program unit that is directly or indirectly referencing the subprogram, the entities remain defined. Entities specified in a SAVE statement remain defined. Entities that are initially defined by DATA statements, and have neither been redefined nor become undefined, remain defined. Execution of a RETURN or END statement does not cause entities in blank common, or entities in any named common block that appears in the main program, to become undefined.

Adjustable Dimensions

Adjustable dimensions allow you to create a more general subprogram that can accept varying sizes of array arguments. For example, a subroutine with a fixed array can be declared as:

```
SUBROUTINE SUM(A)
DIMENSION A(10)
```

The maximum array size subroutine SUM can accept is 10 elements. If the same subroutine is to accept an array of any size, it can be written as:

```
SUBROUTINE SUM(A,N)
DIMENSION A(N)
```

In this case, the value N is passed as an actual argument.

Character strings and arrays can also be adjustable, as in the following subroutine:

```
SUBROUTINE MESSAG (X)
CHARACTER X*(*)
PRINT *, X
```

The subroutine declares X with a length of (*) to accept strings of varying size. Note that the length of the string is passed implicitly (rather than explicitly as an actual argument).

Another form of adjustable dimension is the assumed-size array. In this case, the upper bound of the last dimension of the array is specified by an asterisk. The value of the dimension is not passed as an argument. You are responsible for ensuring that the array in the calling program is large enough to contain all the elements stored into it in the subroutine.

Example:

```
      SUBROUTINE CAT (A,M,N,B,C)
      REAL A(M), B(N), C(*)
      DO 10 I = 1,M
10    C(I) = A(I)
      DO 20 I = 1,N
20    C(I+M) = B(I)
      RETURN
      END
```

Subroutine CAT places the contents of array A, followed by the contents of array B, into array C. The dimension of C in the calling program must be greater than or equal to M+N.

Use of the asterisk form of the adjustable dimension prevents subscript checking for the array, so you should be careful not to reference outside the array bounds.

Assumed-Length Character Strings

If you specify the length of a type character dummy argument as (*), the dummy argument assumes the length of the associated actual argument for each reference to the subroutine or function. A character dummy argument with length (*) is called an assumed-length character string. If the associated actual argument is an array name, the length assumed by the dummy argument is the length of each array element in the associated actual argument.

Example:

```
PROGRAM MN
CHARACTER*3 CC, A(4)
.
.
.
CALL TSUB (CC, A(1)(2:3))
.
.
.
END
SUBROUTINE TSUB(CHAR, Z)
CHARACTER*(*) CHAR, Z(4)
```

The dummy argument CHAR in subroutine TSUB will have length 3 and each element of the array Z will have length 2.

Multiple Entry

The ENTRY statement defines an entry point for a subroutine or function subprogram. This statement has the form:

ENTRY epname *(d, ..., d)*

epname Entry point name

d Dummy argument that can be one of the following:

A variable name

An array name

A dummy procedure name

An asterisk, in a subroutine only

If the entry point has no arguments, either the form ENTRY epname or ENTRY epname () can be used.

Each subprogram has a primary entry point established by the SUBROUTINE or FUNCTION statement that begins the program unit. A subroutine call or function reference usually invokes the subprogram at the primary entry point, and the first statement executed is the first executable statement in the program unit. You can use ENTRY statements to define other entry points.

An ENTRY statement can appear anywhere in a subprogram after the SUBROUTINE or FUNCTION statement except between a block IF statement and its corresponding END IF statement, or between a DO statement and the terminating statement of the DO loop.

When an entry name is used to reference a subprogram, execution begins with the first executable statement that follows the referenced entry point. An entry name is available for reference in any program unit, except in the subprogram that contains the entry name. The entry name can appear in an EXTERNAL statement and (for a function entry name) in a type statement.

Each reference to a subprogram must use an actual argument list that corresponds in number of arguments and type of arguments to the dummy argument list in the corresponding SUBROUTINE, FUNCTION, or ENTRY statement. The dummy arguments for an entry point can therefore be different from the dummy arguments for the primary entry point or another entry point. Type agreement is not required for actual arguments that have no type, such as a dummy subroutine name. A dummy argument cannot be used in an executable statement of a subprogram unless the argument has appeared in a physically preceding FUNCTION, SUBROUTINE, or ENTRY statement.

Example:

```
SUBROUTINE ALPHA (ARR,N)
DIMENSION ARR (N)
DO 20 I = 1,N
.
.
.
ENTRY BETA (ARR,N,K)
DO 40 I = 1,N,K
.
.
.
```

Subroutine ALPHA can be entered either through entry point ALPHA, in which case the first statement executed is DO 20 I = 1,N, or through entry point BETA, in which case the first statement executed is DO 40 I = 1,N. Note that the SUBROUTINE and ENTRY argument lists legally contain different numbers of arguments.

RETURN

The RETURN statement transfers control from a referenced subprogram back to the referencing program unit. This statement has the form:

RETURN *exp*

exp Arithmetic or boolean expression. If exp is not of type integer, the value INT(exp) is used. The expression can be used only in a subroutine. If exp is omitted, the RETURN statement is known as a simple return. If exp is specified, the RETURN statement is known as an alternate return.

When a RETURN statement without the optional expression is executed, control transfers to the statement immediately following the statement that called the referenced subprogram. If the referenced subprogram is a function subprogram, the function value is returned through the function name, the expression containing the function reference is evaluated, the result is stored, and execution continues.

RETURN statements are valid in main programs and in subroutine and function subprograms. Execution of a RETURN statement in a main program has the same effect as a STOP or END statement: the program terminates and control returns to the operating system.

A subprogram can contain more than one RETURN statement. You can place them anywhere in the subprogram where a return operation is desired. Note that if a RETURN statement is not the last executable statement of a subprogram, the statement following the RETURN can be executed only as a result of a flow control statement, such as IF or GO TO, that transfers control to that statement. You can omit the RETURN statement entirely, in which case control returns to the calling program unit when an END statement is encountered in the flow of execution (chapter 5).

Alternate Return

Execution of a RETURN or END statement in a subroutine returns control to the executable statement following the CALL statement in the referencing program unit. Control can be returned to a different statement in the referencing program unit if the RETURN exp form of the RETURN statement is used. Alternate returns can be used only in subroutine subprograms.

An alternate return returns control to a specified point other than the next executable statement following the subprogram reference. The specified point is a statement label in the referencing program unit. The statement labels eligible for alternate return must be included in the actual argument list, with each label preceded by an asterisk. In the SUBROUTINE statement, the dummy argument corresponding to each statement label actual argument must be an asterisk. Other arguments can be included in the actual and dummy argument lists, but the asterisks in the dummy argument list must correspond in number and position to the statement labels in the actual argument list. For example, the statement

```
SUBROUTINE SUB (A, * *)
```

contains two asterisk dummy arguments to be used with an alternate return. This subroutine could be called with the statement

```
CALL SUB (XVAL, *10, *20)
```

where 10 and 20 are the labels of executable statements appearing within the calling program.

The value of exp in the statement RETURN exp determines the statement to which control return. If the value of exp is 1, control returns to the first statement label specified in the actual argument list; if the value of exp is 2, control returns to the second label, and so forth. If exp is less than 1 or greater than the number of labels in the actual argument list, a simple return is performed; that is, control returns to the statement following the CALL. In the preceding example, the statement RETURN 1 in subroutine SUB returns control to the statement labeled 10, and RETURN 2 returns control to the statement labeled 20.

Example:

```
PROGRAM MAIN
CALL COMP (A, B, *20, *30, *40)
.
.
.
END

SUBROUTINE COMP (B1, B2, *, *, *)
.
.
.
RETURN I + J - 3
```

If the value of I + J - 3 is 1, control returns to statement 20; if the value is 2, control returns to statement 30; if the value is 3, control returns to statement 40.

Calling Other-Language Subprograms (CDC FORTRAN Only)

FORTRAN allows you to call subprograms written in languages other than FORTRAN. These languages include CYBIL and COBOL. All languages under NOS/VE generate a standard subprogram calling sequence. Therefore, the only restrictions are to ensure that the subprogram names and dummy arguments (if any) conform to FORTRAN requirements. The rules described earlier in this chapter for calling FORTRAN subprograms also apply to calling subprograms written in other languages.

You should be careful when accessing the same file both in a FORTRAN program in a subprogram written in another language, because of the language dependency of certain operations. The following operations always produce the expected results when performed on shared files:

All terminal input and output.

All usages where output is the only operation being performed on the shared file.

All usages where the file is written by one or more languages and then read by another language, provided that you close and then reopen the file before reading it.

Other usages may cause unexpected results and should be avoided.

Calling CYBIL Procedures

You can call a CYBIL procedure from a FORTRAN program using a standard CALL statement. Values can be passed between the CYBIL procedure and FORTRAN program through the argument list. The CYBIL procedure being called must conform to the following requirements:

The procedure name must be a valid FORTRAN name (1 through 7 letters or digits, beginning with a letter).

The procedure must be declared with the XDCL attribute.

The dummy (formal) parameters must be declared VAR and must correspond in number and data type to the FORTRAN actual arguments.

The arguments passed to the CYBIL subprogram can be variables, constants, symbolic constants, arrays, or expressions with operators.

The correspondence of data types between FORTRAN and CYBIL is as follows:

FORTRAN	CYBIL
INTEGER	INTEGER
LOGICAL	-
CHARACTER	STRING(*) or CHARACTER
COMPLEX	-
BOOLEAN	-
REAL	REAL
DOUBLE PRECISION	-

The dummy arguments can be subranges of any of the above types.

The storage sequence for CYBIL arrays having dimension greater than one differs from that of FORTRAN. In FORTRAN, arrays are stored in columnwise order, whereas in CYBIL, arrays are stored in rowwise order. Thus, for example, the elements of a FORTRAN array dimensioned (3,2) are stored in ascending locations as follows:

(1,1) (2,1) (3,1) (1,2) (2,2) (3,2)

A CYBIL array dimensioned (3,2) is stored in ascending locations as follows:

(1,1) (1,2) (2,1) (2,2) (3,1) (3,2)

The following example shows a FORTRAN main program that calls a CYBIL procedure named CT. The main program reads a character string and a single character from the terminal and passes them to the CYBIL procedure. The CYBIL procedure counts the number of occurrences of the character within the string and returns the result to the main program. The main program prints the result and terminates.

```
      PROGRAM STRING
      CHARACTER STR*60, C

C  Request and read input values.

      PRINT *, ' Type input string'
      READ (*,*) STR
      PRINT *, ' Type character to be counted'
      READ (*,*) C

C  Call CYBIL procedure.

      CALL CT (C, STR, N)

C  Print results.

      PRINT 99, STR, C, N
99    FORMAT (' Input string= ', A60, /' Character= ', A1,
     +        /' Count= ', I1)
      END
```

CYBIL Procedure:

```
MODULE char_count;
      PROCEDURE [XDCL] ct (VAR c: char;
                           VAR s: string(*);
                           VAR i: integer);
        VAR j: integer;
        i := 0
        FOR j := 1 TO STRLENGTH(s) DO
          IF s(j) = c THEN
            i := i + 1;
          IFEND
        FOREND;
      PROCEND ct;
    MODEND
```

Example of terminal dialog:

```
/fortran input=cytest binary_object=ftnbin <-- Compile FORTRAN main program.

/cybil input=cyproc binary_object=cybin <----- Compile CYBIL procedure.

/execute_task (ftnbin,cybin) <---------------- Execute program.

 Type input string <-------------------------- Prompt for input.
? 'Hi there, somebody' <---------------------- User input.

Type character to be counted <---------------- Prompt for input.
? 'e' <--------------------------------------- User input.

Input string= Hi there, somebody
Character= e                      <----------- Program output.
Count= 3
```

Calling COBOL Subprograms

You can call COBOL subprograms from a FORTRAN program using a standard CALL statement. The name of the COBOL subprogram must be a valid FORTRAN name (1 through 7 letters or digits beginning with a letter). You can pass values between the calling program and the called subprogram through an argument list or through common storage.

If you use an argument list, the actual arguments in the CALL statement are associated with dummy arguments specified in the USING clause of the PROCEDURE DIVISION statement. The actual arguments must correspond in number and data type to the dummy arguments. The rules for argument association described earlier in this chapter apply to COBOL subprograms as well.

The correspondence of data types between FORTRAN and COBOL is as follows:

FORTRAN	COBOL
REAL	COMP-1
DOUBLE PRECISION	COMP-2
INTEGER (signed)	S9(18) COMP SYNC LEFT
CHARACTER	X(n) or 9(n)
COMPLEX	-
BOOLEAN	-
LOGICAL	-

Communication through common storage is achieved through the named common block CCOMMON. In the COBOL subprogram, all data assigned to the COMMON-STORAGE section is automatically placed in CCOMMON. You can share the common storage area by declaring CCOMMON in the FORTRAN calling program.

The following example shows how a FORTRAN program can call a simple COBOL subprogram. The FORTRAN program COBTST calls the COBOL subprogram CBSUB, and passes values through the argument list and through CCOMMON. An integer value and a real value are passed to CBSUB through CCOMMON, and a character string STR1 is passed through the argument list. CBSUB displays the two numbers and the string, and returns a string to COBTST through the argument list.

Program COBTST:

```
      PROGRAM COBTST
      CHARACTER STR1*21, STR2*9
      COMMON /CCOMMON/I, X <---------------------- Items in CCOMMON share storage
      STR1 = 'Here are some numbers'               with items in the COMMON-STORAGE
      I = 4                                        section.
      X = 2.414

C Call COBOL subprogram.

      CALL CBSUB (STR1, STR2) <------------------- Actual arguments are associated
      PRINT *, STR2                                with items declared in LINKAGE
      END                                          section.
```

Subprogram CBSUB:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CBSUB.

DATA DIVISION.
COMMON-STORAGE SECTION.                  <---- Define items to be in common.
01 INTEGER-1   PIC 9(18) COMP SYNC LEFT.
01 REAL-1      COMP-1.

LINKAGE SECTION.
77 STRING-1 PIC X(21).  <---------------------- Define items to be passed as
77 STRING-2 PIC X(9).                          arguments.

PROCEDURE DIVISION USING STRING-1, STRING-2.
PASSING-DATA.
    DISPLAY STRING-1.
    DISPLAY "First number = " INTEGER-1.
    DISPLAY "Second number = " REAL-1.
    MOVE "Thank you" TO STRING-2.
    EXIT PROGRAM.
```

Terminal Dialog (assumes FORTRAN source is on file FPROG, and COBOL source is on file CSUB):

```
/fortran input=fprog binary_object=fbin <------ Compile FORTRAN program.

/cobol input=csub binary_object=cbin sp=true <- Compile COBOL subprogram.

/execute_task (fbin, cbin) <------------------- Execute program.

Here are some numbers
First number =                    4       <---- Program output.
Second number =    +.24140000000000E+001
Thank you
```

Intrinsic Functions 8

An intrinsic function is a compiler-defined procedure that returns a single value. Intrinsic functions are referenced in the same way as user-written (external) function subprograms. If, in a particular program unit, a variable, array, or statement function is declared with the same name as an intrinsic function, the name cannot refer to the intrinsic function within that program unit. If a function subprogram is written with the same name as an intrinsic function, use of the name references the intrinsic function, unless the name is declared as the name of an external function with the EXTERNAL statement described in chapter 3.

Intrinsic functions are typed by default and need not appear in any explicit type statement in the program. Explicitly typing a generic intrinsic function name does not remove the generic properties of the name. If you attempt to type an intrinsic function as something other than its default type, a warning message is displayed and the type statement is disregarded. For intrinsic functions that have multiple arguments, all arguments must be of the same type.

Generic and Specific Names

Certain intrinsic functions have a generic name and one or more specific names. For these functions, either the generic name or one of the specific names can be used. The generic name provides more flexibility because it can be used with any of the valid data types; except for functions performing type conversion, nearest integer, and absolute value with a complex argument, the type of the argument determines the type of the result. However, only a specific name can be used as an actual argument when passing the function name to a user-defined subprogram. Using a specific name requires a specific argument type. For example, the generic function name LOG computes the natural logarithm of an argument. Its argument can be real, double precision, or complex. The type of the result is the same as the type of the argument.

Specific function names ALOG, DLOG, and CLOG also compute the natural logarithm. The specific function name ALOG computes the log of a real argument and returns a real result. Likewise, the specific name DLOG is for double precision arguments and results, and the specific name CLOG is for complex arguments and results.

The intrinsic functions are summarized in table 8-1. The functions are listed in alphabetical order of generic name or, where no generic name exists, of specific name. An asterisk in the Generic Name column indicates that the function is a CDC extension. For specific names, the types of the arguments and results are shown. Table 8-2 shows the domain and range for a subset of the mathematical intrinsic functions.

Table 8-1. Intrinsic Functions

Generic Name	Specific Names	Type of Argument	Type of Function	Description
ABS	IABS	Integer	Integer	Absolute Value
	ABS	Real	Real	
	DABS	Double	Double	
	CABS	Complex	Real	
ACOS	ACOS	Real	Real	Arccosine
	DACOS	Double	Double	
None	AIMAG	Complex	Real	Imaginary part of complex argument
AINT	AINT	Real	Real	Truncation
	DINT	Double	Double	
None	AMAXO	Integer	Real	Maximum value
None	AMINO	Integer	Real	Minimum value
None*	AND	Any type but character	Boolean	Boolean product
ANINT	ANINT	Real	Real	Nearest whole number
	DNINT	Double	Double	
ASIN	ASIN	Real	Real	Arcsine
	DASIN	Double	Double	
ATAN	ATAN	Real	Real	Arctangent
	DATAN	Double	Double	
ATAN2	ATAN2	Real	Real	
	DATAN2	Double	Double	
None*	ATANH	Real	Real	Hyperbolic arctangent
BOOL*	–	Any type but logical	Boolean	Conversion to boolean
None	CHAR	Integer	Character	Integer conversion to character
None*	COMPL	Any type but character	Boolean	Complement
CMPLX	–	Integer	Complex	Conversion to complex
	–	Real	Complex	
	–	Double	Complex	
	–	Complex	Complex	
COS	COS	Real	Real	Cosine, argument in radians
	DCOS	Double	Double	
	CCOS	Complex	Complex	
None	COSD	Real	Real	Cosine, argument in degrees
COSH	COSH	Real	Real	Hyperbolic cosine
	DCOSH	Double	Double	
None	CONJG	Complex	Complex	Negation of imaginary part

(Continued on next page)

Table 8-1. Intrinsic Functions
(Continued from previous page)

Generic Name	Specific Names	Type of Argument	Type of Function	Description
DBLE	_	Integer	Double	Conversion to double precision
	_	Real	Double	
	_	Double	Double	
	_	Complex	Double	
DIM	IDIM	Integer	Integer	Positive difference
	DIM	Real	Real	
	DDIM	Double	Double	
None	DPROD	Real	Double	Double precision product
None	EQV	Any type but character	Boolean	Equivalence
None	ERF	Real	Real	Error function
None	ERFC	Real	Real	Complementary error function
EXP	EXP	Real	Real	Exponential function
	DEXP	Double	Double	
	CEXP	Complex	Complex	
None	ICHAR	Character	Integer	Character conversion to integer
None	INDEX	Character	Integer	Index of a substring
INT	INT	Integer	Integer	Conversion to integer
	INT	Real	Integer	
	IFIX	Real	Integer	
	IDINT	Double	Integer	
	_	Complex	Integer	
None	LEN	Character	Integer	Length of character string
None	LGE	Character	Logical	Lexically greater than or equal
None	LGT	Character	Logical	Lexically greater than
None	LLE	Character	Logical	Lexically less than
None	LLT	Character	Logical	Lexically less than
LOG	ALOG	Real	Real	Natural Logarithm
	DLOG	Double	Double	
	CLOG	Complex	Complex	
LOG10	ALOG10	Real	Real	Common Logarithm
	DLOG10	Double	Double	
None	MASK	Integer or Boolean	Boolean	Mask
MAX	MAX0	Integer	Integer	Largest value
	AMAX1	Real	Real	
	DMAX1	Double	Double	
None	MAX1	Real	Integer	

(Continued on next page)

Table 8-1. Intrinsic Functions

(Continued from previous page)

Generic Name	Specific Names	Type of Argument	Type of Function	Description
MIN	MIN0	Integer	Integer	Smallest value
	AMIN1	Real	Real	
	DMIN1	Double	Double	
None	MIN1	Real	Integer	
MOD	MOD	Integer	Integer	Remaindering
	AMOD	Real	Real	
	DMOD	Double	Double	
None	NEQV	Any type but character	Boolean	Nonequivalence
NINT	NINT	Real	Integer	Nearet integer
	IDNINT	Double	Integer	
None	OR	Any type but character	Boolean	Boolean sum
None	RANF	None	Real	Random number generator
REAL	FLOAT	Integer	Real	Conversion to real
	REAL	Integer	Real	
	–	Real	Real	
	–	Complex	Real	
	SNGL	Double	Real	
None	SHIFT	Any type but character for a1; integer or Boolean for a2	Boolean	Shift
SIGN	ISIGN	Integer	Integer	Transfer of sign
	SIGN	Real	Real	
	DSIGN	Double	Double	
SIN	SIN	Real	Real	Sine (argument in radians)
	DSIN	Double	Double	
	CSIN	Complex	Complex	
None	SIND	Real	Real	Sine (argument in degrees)
SINH	SINH	Real	Real	Hyperbolic sine
	DSINH	Double	Double	
SQRT	SQRT	Real	Real	Square root
	DSQRT	Double	Double	
	CSQRT	Complex	Complex	
TAN	TAN	Real	Real	Tangent (argument in radians)
	DTAN	Double	Double	
None	TAND	Real	Real	Tangent (argument in degrees)
TANH	TANH	Real	Real	Hyperbolic tangent
	DTANH	Double	Double	
None	XOR	Any type but character	Boolean	Exclusive OR

Table 8-2. Mathematical Intrinsic Functions

Funtion	Domain	Range
ACOS(a) DACOS(a)	|a| ≤ 1	0 ≤ ACOS(a) ≤ pi
ASIN(a) DASIN(a)	|a| ≤ 1	-pi/2 ≤ ASIN(a) ≤ pi/2
ATAN(a) DATAN(a)	-infinity ≤ a ≤ infinity	-pi/2 ≤ ATAN(a) ≤ pi/2
ATAN2 (a1,a2) DATAN2 (a1,a2)	a2<0, at<0 a2<0, a1≥0 a2=0, a1<0 a2=0, a1>0 a2=0, a1<0 a2=0, a1≥0 a2=0, a1=0 (error)	-pi < ATAN2 (a1,a2) < - pi/2 pi/2 ≤ ATAN2 (a1,a2) ≤ pi ATAN2 (a1,a2) = -pi/2 ATAN2 (a1,a2) = pi/2 -pi/2 < ATAN2 (a1,a2) < 0 0 ≤ ATAN2 (a1,a2) < pi/2
ATANH (a1,a2) DCOS(a)	|a| < 2**47	-1 ≤ COS(a) ≤1
CCOS (ar,ai)	|ar| < 2**47 |ai| < 4095*Log(2)	
COSD(a)	|a| < 2**47	-1 ≤ COSD(a) ≤ 1
COSH(a) DCOSH(a)	|a| < 4095*Log(2)	
ERF(a)	0 ≤ a ≤ infinity	-1 ≤ ERF(a) ≤ 1
ERFC(a)	0 ≤ a ≤ 25.923	0 ≤ ERFC(a) ≤ 2
EXP(x) DEXP(x)	|x| < 4095*Log(2)	
CEXP(ar,ai)	|ar| < 4095*Log(2) |ai| < 2**47	
LOG(a) ALOG(a) DLOG(a)	a > 0	LOG(a) < 4095*Log (2)
CLOG(ar,ai)	(ar,ai) ≠ (0,0) ar**2 + ai**2 in machine range	|imaginary part| < pi/2
LOG10(a) ALOG10(a) DLOG10(a)	a >0	|LOG10(a)| < 4095*Log (2)
SIN(a) DSIN(a)	|a| < 2**47	-1 ≤ SIN(a) ≤ 1
CSIN(ar,ai)	|ar| < 2**47 |ai| < 4095*Log(2)	
SIND(a) DSINH(a)	|a| < 4095*Log(2)	

(Continued on next page)

Table 8-2. Mathematical Intrinsic Functions

— (Continued from previous page)

Funtion	Domain	Range
SQRT(a) DSQRT(a)	a ≥ 0	SQRT(a) ≥ 0
CSQRT(ar,ai)	(ar**2+ai**2) + |ar| in machine range	value in right half of plane (ar ≥ 0)
TAN(a) DTAN(a)	|a| < 2**47	
TAND(a)	|a| < 2**47 a cannot be exact odd multiple of 90	
TANH(a)		-1 ≤ TANH(a) ≤ 1

Function Descriptions

Following are descriptions of the intrinsic functions. The generic and specific names are listed in alphabetical order.

ABS

ABS(a) is a generic function that returns the absolute value (magnitude) of an integer, real, complex, or double precision argument. The result is integer, real, or double precision, depending on the argument type. For a complex argument, the result is the square root of (ar**2 + ai**2), where ar is the real part of the argument and ai is the imaginary part. The specific names are IABS, ABS, DABS, and CABS.

ACOS

ACOS(a) is a generic function that returns the arccosine of a real or double precision argument. The result is in radians. The result is real or double precision, depending on the argument type. The specific names are ACOS and DACOS.

AIMAG

AIMAG(a) is a specific function that returns the imaginary part of a complex argument. The real result is ai, where the complex argument is (ar,ai). AIMAG does not have a generic name.

AINT

AINT(a) is a generic function that returns a whole number after truncation. The result is real. For a real or double precision argument, the result is 0 if |a| < 1. If |a| ≥ 1, the result is the largest whole number with the same sign as argument a that does not exceed |a|. The specific names are AINT and DINT.

ALOG

ALOG(a) is a specific function that returns the natural logarithm (logarithm base e) of a real argument. The result is real. The argument must be greater than zero. The generic name is LOG.

ALOG10

ALOG10(a) is a specific function that returns the common logarithm (logarithm base 10) of a real argument. The result is real. The argument must be greater than zero. The generic name is LOG10.

AMAX0

AMAX0(a, ..., a) is a specific function that returns the largest value from the 2 through 500 arguments. The arguments are integer, and the result is real. There is no generic name.

AMAX1

AMAX1(a, ..., a) is a specific function that returns the largest value from the 2 through 500 arguments. The arguments are real, and the result is real. The generic name is MAX.

AMIN0

AMIN0(a, ..., a) is a specific function that returns the smallest value from the 2 through 500 arguments. The arguments are integer, and the result is real. There is no generic name.

AMIN1

AMIN1(a, ..., a) is a specific function that returns the smallest value from the 2 through 500 arguments. The arguments are real, and the result is real. The generic name is MIN.

AMOD

AMOD(a1, a2) is a specific function that returns the remainder of a1 divided by a2 (a1 modulus a2). The result is a1 - INT(a1/a2) * a2. The arguments are real and the result is real. If a2 is zero, results are undefined. The generic name is MOD.

AND

AND(a, ..., a) is a specific function that returns the boolean product of the 2 through 500 arguments. The arguments can be any type but character, and the result is boolean. The result is the same as for the boolean .AND. operator.

ANINT

ANINT(a) is a generic function that returns the nearest whole number. The result is defined as INT(a+.5) if a is positive or zero, and INT(a-.5) if a is negative. The argument is real or double precision, and the result has the same type as the argument. The specific names are ANINT and DNINT.

ASIN

ASIN(a) is a generic function that returns the arcsine of a real or double precision argument. The result is in radians. The result is real or double precision, depending on the argument type. The specific names are ASIN and DASIN.

ATAN

ATAN(a) is a generic function that returns the arctangent of a real or double precision argument. The result is in radians. The result is real or double precision, depending on the argument type. The specific names are ATAN and DATAN.

ATANH (CDC Extension)

ATANH(a) is a specific function that returns the hyperbolic arctangent of a real argument. The result is real.

ATAN2

ATAN2(a1, a2) is a generic function that returns the arctangent of a1/a2. The result is as follows:

Arguments	Result
a2<0, a1<0	-pi + arctan(a1/a2)
a2=0, a1<0	-pi/2
a2=0, a1>0	pi/2
a2=0, a1=0	error
a2<0, a1≥0	pi + arctan(a1/a2)
a2>0	arctan(a1/a2)

The result is expressed in radians. The result is real or double precision, depending on the type of the arguments. The arguments must not both be zero. The specific names are ATAN2 and DATAN2.

BOOL (CDC Extension)

BOOL(a) is a generic function that performs type conversion and returns a boolean value. The argument can be integer, real, double precision, complex, character, or boolean. For an integer, real, or boolean argument, the result is the bit string constituting the data. For a double precision or complex argument, the result is the bit string after conversion of the argument to real with REAL(a). For a character argument, the result is the value of the boolean string constant nHf, where n is the length and f is the character value; if n is greater than 8, the rightmost characters are truncated. There are no specific names.

CABS

CABS(a) is a specific function that returns the absolute value of a complex argument. The result is real. The result is the square root of (ar**2 + ai**2), where ar is the real part of the argument and ai is the imaginary part. The generic name is ABS.

CCOS

CCOS(a) is a specific function that returns the cosine of a complex argument. The result is complex. The generic name is COS.

CEXP

CEXP(a) is a specific function that returns the value of e raised to a complex power. The argument and result are complex. The generic name is EXP.

CHAR

CHAR(i) returns the character value in the ith position of a collating sequence. The argument is type integer, and the result is type character with length one. The value returned depends on the collating sequence in use. If the ASCII collating sequence is used, the argument must be in the range 0 through 255; the first character in the collating sequence corresponds to value 0, the second character to value 1, the third to value 2, and so forth. The result is the selection of a single character from the collating sequence. If, in a user-specified collating sequence, more than one character has weight a, any of the characters can be returned.

CLOG

CLOG(a) is a specific function that returns the natural logarithm (logarithm base e) of the argument. The argument and result are complex. The generic name is LOG.

CMPLX

CMPLX(a) or CMPLX(a1, a2) is a generic function that performs type conversion and returns a complex value. CMPLX can have one or two arguments. A single argument can be integer, real, double precision, or complex.

For two arguments a1 and a2, the arguments must be of the same type and must both be integer, real, or double precision. The result is complex, with REAL(a1) used as the real part and REAL(a2) used as the imaginary part.

For a single integer, real, or double precision argument, the result is complex, with REAL(a) used as the real part and the imaginary part set to zero. For a single complex argument, the result is the same as the argument.

There are no specific names.

COMPL (CDC Extension)

COMPL(a) returns a complemented value. The argument can be any type except character, and the result is boolean. If the argument is not boolean, the argument is converted with BOOL(a). The result is the result of the logical operator .NOT. on a boolean value.

CONJG

CONJG(a) is a specific function that returns the conjugate of a complex argument. The result is complex. For a complex argument (ar,ai), the result is (ar,-ai), with the imaginary part negated. CONJG does not have a generic name.

COS

COS(a) is a generic function that returns the cosine of a real, double precision, or complex argument. The argument is in radians. The result has the same type as the argument. The specific names are COS, CCOS, and DCOS.

COSD (CDC Extension)

COSD(a) is a specific function that returns the cosine of a real argument. The argument is in degrees. The result is real. COSD does not have a generic name.

COSH

COSH(a) is a generic function that returns the hyperbolic cosine of a real or double precision argument. The result is real or double precision, depending on the argument type. The specific names are COSH and DCOSH.

CSIN

CSIN(a) is a specific function that returns the sine of a complex argument. The result is complex. The generic name is SIN. See the SIN description.

CSQRT

CSQRT(a) is a specific function that returns the square root of a complex argument. The result is complex. The generic name is SQRT. See the SQRT description.

DABS

DABS(a) is a specific function that returns the absolute value (magnitude) of a double precision argument. The result is double precision. The generic name is ABS. See the ABS description.

DACOS

DACOS(a) is a specific function that returns the arccosine of a double precision argument. The result is double precision. The generic name is ACOS. See the ACOS description.

DASIN

DASIN(a) is a specific function that returns the arcsine of a double precision argument. The result is double precision. The generic name is ASIN. See the ASIN description.

DATAN

DATAN(a) is a specific function that returns the arctangent of a double precision argument. The result is double precision. The generic name is ATAN. See the ATAN description.

DATAN2

DATAN2(a1, a2) is a specific function that returns the arctangent of a1/a2. The argument and result are double precision. The result is in radians, and has the following values:

Arguments	Result
a2<0, a1<0	-pi + arctan(a1/a2)
a2=0, a1<0	-pi/2
a2=0, a1>0	pi/2
a2=0, a1=0	error
a2<0, a1≥0	pi + arctan(a1/a2)
a2>0	arctan(a1/a2)

The generic name is ATAN2. See the ATAN2 description.

DBLE

DBLE(a) is a generic function that converts the argument to double precision. The argument can be integer, real, double precision, or complex. For an integer or real argument, the result has as much precision of the significant part of the argument as the double precision field can contain. For a double precision argument, the result is the argument. For a complex argument, the real part is used, and the result has as much precision of the significant part of the real part of the argument as the double precision field can contain. There are no specific names.

DCOS

DCOS(a) is a specific function that returns the cosine of a double precision argument. The result is double precision. The generic name is COS. See the COS description.

DCOSH

DCOSH(a) is a specific function that returns the hyperbolic cosine of a double precision argument. The result is double precision. The generic name is COSH. See the COSH description.

DDIM

DDIM(a1, a2) is a specific function that returns a positive difference. The result is the value of a1 - a2 if a1 is greater than or equal to a2; if a1 is les than a2, it returns zero. The argument and result are double precision. The generic name is DIM. See the DIM description.

DEXP

DEXP(a) is a specific function that returns an exponential result. The argument and result are double precision. The generic name is EXP. See the EXP description.

DIM

DIM(a1, a2) is a generic function that returns a positive difference. The result is integer, real, or double precision, depending on the argument type. Both arguments must be the same type. The result is a1 - a2 if a1 is greater than or equal to a2, and the result is zero if a1 is less than a2. The specific names are DIM, IDIM, DDIM.

DINT

DINT(a) is a specific function that returns a whole number after truncation. The argument and result are double precision. The generic name is AINT. See the AINT description.

DLOG

DLOG(a) is a specific function that returns the natural logarithm (logarithm base e) of a double precision argument. The result is double precision. The generic name is LOG. See the LOG description.

DLOG10

DLOG10(a) is a specific function that returns the common logarithm (logarithm base 10) of a double precision argument. The result is double precision. The generic name is LOG10. See the LOG10 description.

DMAX1

DMAX1(a, ..., a) is a specific function that returns the largest value from the 2 through 500 arguments. The arguments are double precision, and the result is double precision. The generic name is MAX. See the MAX description.

DMIN1

DMIN1(a, ..., a) is a specific function that returns the smallest value of the 2 through 500 arguments. The arguments are double precision, and the result is double precision. The generic name is MIN. See the MIN description.

DMOD

DMOD(a1, a2) is a specific function that returns a1 modulus a2 (the remainder of a1 divided by a2). The result is a1 - (INT(a1/a2) * a2). The arguments and the result are double precision. If a2 is zero, results are undefined. The generic name is MOD. See the MOD description.

DNINT

DNINT(a) is a specific function that returns the nearest whole number. The argument and result are double precision. The result is defined as INT(a+.5) if a is positive or zero, and INT(a-.5) if a is negative. The generic name is ANINT. See the ANINT description.

DPROD

DPROD(a1, a2) returns the double precision product of two real arguments. The result is defined as a1*a2. The result is double precision.

DSIGN

DSIGN(a1, a2) is a specific function that performs a transfer of sign. The result is defined as |a1| if a2 is positive or zero, and -|a1| if a2 is negative. The arguments and the result are double precision. The generic name is SIGN. See the SIGN description.

DSIN

DSIN(a) is a specific function that returns the sine of a double precision argument. The argument is in radians. The result is double precision. The generic name is SIN. See the SIN description.

DSINH

DSINH(a) is a specific function that returns the hyperbolic sine of a double precision argument. The result is double precision. The generic name is SINH. See the SINH description.

DSQRT

DSQRT(a) is a specific function that returns the square root of a double precision argument. The result is double precision. The argument must not be negative. The generic name is SQRT. See the SQRT description.

DTAN

DTAN(a) is a specific function that returns the tangent of a double precision argument. The argument is in radians. The result is double precision. The generic name is TAN. See the TAN description.

DTANH

DTANH(a) is a specific function that returns the hyperbolic tangent of a double precision argument. The result is double precision. The generic name is TANH. See the TANH description.

EQV (CDC Extension)

EQV(a, ..., a) returns the equivalence of the 2 through 500 arguments. The arguments can be any type except character, and the result is boolean. The result is the same as for the boolean .EQV. operator.

ERF (CDC Extension)

ERF(a) returns an error function result. The argument and result are real. The argument must be positive. The mathematical definition is as follows:

$$ERF(x) = 2 / \sqrt{pi} \int_{0}^{x} e^{-t^2} \quad dt$$

ERFC (CDC Extension)

ERFC(a) returns a complementary error function result. The argument and result are real. The result is 1-ERF(a). The mathematical definition of ERFC is as follows:

$$ERFC(x) = 2 / \sqrt{pi} \int_{x}^{\infty} e^{-t^2} \quad dt$$

EXP

EXP(a) is a generic function that returns an exponential result (e**a). The result is real, double precision, or complex, depending on the argument type. The specific names are EXP, DEXP, and CEXP.

FLOAT

FLOAT(a) is a specific function that returns the value of the integer argument after conversion to real. The generic name is REAL. See the REAL description.

IABS

IABS(a) is a specific function that returns the absolute value (magnitude) of an integer argument. The result is integer. The generic name is ABS. See the ABS description.

ICHAR

ICHAR(a) returns the value of a character argument after conversion to integer. The value returned depends on the collating weight of the argument in the collating sequence in use. For the ASCII collating sequence, the first character in the collating sequence is at position 0, the second character at position 1, the third at position 2, and so forth. For a user-specified collating sequence, two or more characters can have the same value. The argument is a character value with a length of one character, and the value returned is the integer position of that character in the collating sequence.

IDIM

IDIM(a1, a2) is a specific function that returns the positive difference of two integer arguments. The result is a1 - a2 if a1 is greater than a2, and zero if a1 is not greater than a2. The result is integer. The generic name is DIM. See the DIM description.

IDINT

IDINT(a) is a specific function that returns the value of a double precision argument after conversion to integer. The generic name is INT. See the INT description.

IDNINT

IDNINT(a) is a specific function that returns the nearest integer. The result is INT(a+.5) if a is positive or zero, and INT(a-.5) if a is negative. The argument is double precision. The generic name is NINT. See the NINT description.

IFIX

IFIX(a) is a specific function that returns the value of the real argument after conversion to integer. The result is INT(a). The generic name is INT. See the INT description.

INDEX

INDEX(a1, a2) returns the location of substring a2 within string a1. Both arguments must be type character. If string a2 occurs as a substring within string a1, the result is an integer indicating the starting position of the substring a2 within a1. If a2 does not occur as a substring within a1, the result is 0. If a2 occurs as a substring more than once within a1, only the starting position of the first occurrence is returned.

INT

INT(a) is a generic function that performs type conversion to integer. The result is integer, and the argument can be integer, real, double precision, or complex. For an integer argument, the result is the argument. For a real or double precision argument where the |a| < 1, the result is 0. Where the |a| ≥ 1, the result is the largest integer with the same sign as argument a that does not exceed |a|. For a complex argument, the real part is used, and the result is the same as for a real argument. The specific names are INT, IFIX, and IDINT.

ISIGN

ISIGN(a1, a2) is a specific function that performs a transfer of sign. The result is |a1| if a2 is positive or zero, and -|a1| of a2 is negative. The arguments and result are integer. The generic name is SIGN. See the SIGN description.

LEN

LEN(a) returns the length of a character string. The argument is type character, and the result is an integer indicating the length of the argument.

LGE

LGE(a1, a2) returns a logical result indicating lexically greater than or equal to. The arguments are character strings. The result is true if a1 follows a2 or a1 is equal to a2 in the ASCII collating sequence (shown in appendix B); the result is false otherwise. If the arguments are of unequal length, the shorter argument is treated is if it were extended on the right with blanks to the length of the longer argument.

LGT

LGT(a1, a2) returns a logical result indicating lexically greater than. The arguments are character strings. The result is true if a1 follows a2 in the ASCII collating sequence (shown in appendix B); the result is false otherwise. If the arguments are of unequal length, the shorter argument is treated is if it were extended on the right with blanks to the length of the longer argument.

LLE

LLE(a1, a2) returns a logical result indicating lexically less than or equal to. The arguments are character strings. The result is true if a1 precedes a2 or a1 is equal to a2 in the ASCII collating sequence (shown in appendix B); the result is false otherwise. If the arguments are of unequal length, the shorter argument is treated as if it were extended on the right with blanks to the length of the longer argument.

LLT

LLT(a1, a2) returns a logical result indicating lexically less than. The arguments are character strings. The result is true if a1 precedes a2 in the ASCII collating sequence (shown in appendix B); the result is false otherwise. If the arguments are of unequal length, the shorter argument is treated is if it were extended on the right with blanks to the length of the longer argument.

LOG

LOG(a) is a generic function that returns the natural logarithm (logarithm base e). The result is real, double precision, or complex, depending on the argument type. The specific names are ALOG, DLOG, and CLOG.

LOG10

LOG10(a) is a generic function that returns a common logarithm (logarithm base 10). The result is real or double precision, depending on the argument type. The argument must be greater than zero. The specific names are ALOG10 and DLOG10.

MASK (CDC Extension)

MASK(a) returns a boolean result. The argument is integer or boolean in the range 0 through 64. The result is a word of left-justified one bits followed by (64 - a) zero bits. If a is less than zero, the result is the same as for a = 0.

MAX

MAX(a, ..., a) is a generic function that returns the largest value from the 2 through 500 arguments. The result is integer, real, or double precision, depending on the type of the arguments. The specific names are MAX0, AMAX1, and DMAX1.

MAX0

MAX0(a, ..., a) is a specific function that returns the largest value from the 2 through 500 integer arguments. The result is integer. The generic name is MAX. See the MAX description.

MAX1

MAX1(a, ..., a) is a specific function that returns the largest value from the 2 through 500 real arguments. The result is integer. There is no generic name for MAX1. See the MAX description.

MIN

MIN(a, ..., a) is a generic function that returns the smallest value from the 2 through 500 arguments. The result is integer, real, or double precision, depending on the type of arguments. The specific names are MIN0, AMIN1, and DMIN1.

MIN0

MIN0(a, ..., a) is a specific function that returns the smallest value from the 2 through 500 integer arguments. The result is integer. The generic name is MIN. See the MIN description.

MIN1

MIN1(a, ..., a) is a specific function that returns the smallest value from the 2 through 500 real arguments. The result is integer.

MOD

MOD(a1, a2) is a generic function that returns a1 modulus a2 (the remainder of a1 divided by a2). The result is integer, real, or double precision, depending on the argument type. The result is a1 - (INT(a1/a2) * a2). If a is zero, results are undefined. The specific names are MOD, AMOD, and DMOD.

NEQV (CDC Extension)

NEQV(a, ..., a) returns the nonequivalence of the arguments. The result is boolean, and the 2 through 500 arguments are any type but character. The result is the same as for the boolean exclusive or (.NEQV.) operator.

NINT

NINT(a) is a generic function that returns the nearest integer. The result is integer, and the argument can be real or double precision. If the argument is zero or positive, the result is (INT(a+.5)). If the argument is negative, the result is (INT(a-.5)). The specific names are NINT and IDNINT.

OR (CDC Extension)

OR(a, ..., a) returns the boolean sum of the arguments. The result is boolean, and the 2 through 500 arguments are any type but character. The result is the same as for the boolean .OR. operator.

RANF (CDC Extension)

RANF returns a random number. Successive calls to RANF yield a random sequence of numbers. Since there is no argument, RANF is referenced as RANF( ). The result is real and is in the range 0 < result < 1. You can reinitialize the seed by calling the RANSET function described in chapter 9.

REAL

REAL(a) is a generic function that performs type conversion and returns a real result. The argument can be integer, real, double precision, or complex. For an integer or double precision argument, REAL(a) is as much precision of the significant part of the argument as a real item can contain. For a complex argument (ar,ai), the result is ar. The specific names are REAL, FLOAT, and SNGL.

SHIFT (CDC Extension)

SHIFT(a1, a2) returns a shifted result. The argument a1 is any type but character, and argument a2 is integer or boolean. The boolean result is a1 shifted a2 bit positions. The shift is left circular if a2 is positive, or right with sign extension and end off if a2 is negative. Argument a2 is in the range -64 through +64.

SIGN

SIGN(a1, a2) is a generic function that returns a value after a transfer of sign. The result is integer, real, or double precision, depending on the argument type. The result is |a1| if a2 is zero or positive. The result is -|a1| if a2 is negative. The specific names are SIGN, ISIGN, and DSIGN.

SIN

SIN(a) is a generic function that returns the sine of the argument. The argument is in radians. The result is real, double precision, or complex, depending on the argument type. The generic name is SIN. See the SIN description.

The specific names are SIN, DSIN, and CSIN.

SIND (CDC Extension)

SIND(a) returns the sine of a real argument. The result is real. The argument is in degrees.

SINH

SINH(a) is a generic function that returns a hyperbolic sine of a real or double precision argument. The result is real or double precision, depending on the argument type. The specific names are SINH and DSINH.

SNGL

SNGL(a) is a specific function that returns the value of a double precision argument after conversion to single precision real. The generic name is REAL. See the REAL description.

SQRT

SQRT(a) is a generic function that returns a principal square root of a real, double precision, or complex argument. The result is real, double precision, or complex, depending on the argument type. The argument must not be negative. The specific names are SQRT, DSQRT, and CSQRT.

TAN

TAN(a) is a generic function that returns the tangent of a real or double precision argument. The argument is in radians. The result is real or double precision, depending on the argument type. The specific names are TAN and DTAN.

TAND (CDC Extension)

TAND(a) is a specific function that returns the tangent of a real argument. The result is real. The argument is in degrees. TAND does not have a generic name.

TANH

TANH(a) is a generic function that returns the hyperbolic tangent of a real or double precision argument. The result is real or double precision, depending on the argument type. The specific names are TANH and DTANH.

XOR (CDC Extension)

XOR(a, ..., a) is a specific function that returns the exclusive OR of the 2 through 500 arguments. The arguments can be any type but character, and the result is boolean. The result is the same as for the boolean exclusive or (.XOR.) operator. There are no generic names.

FORTRAN-Callable Subprograms 9

FORTRAN provides a number of subroutines and function calls that enable you to access other CDC products as well as perform a variety of other operations. The FORTRAN-callable subprograms include:

FORTRAN-Sort/Merge subprograms

System Command Language subprograms

Utility subprograms

FORTRAN-Sort/Merge Subprograms

Sort/Merge is a generalized utility program that is used to rearrange the records of a file in a specified sequence. The sort feature of Sort/Merge allows you to sort records into a desired order. Data to be sorted can be any of the FORTRAN data types. The merge feature enables you to combine the contents of two or more previously sorted files into a single file. The sort or merge can be based on one of the standard collating sequences which you select.

FORTRAN provides a collection of subroutine calls that allow you to call Sort/Merge directly from a FORTRAN program. The following paragraphs provide a summary of the Sort/Merge calls. Before using the FORTRAN-Sort/Merge routines you should be familiar with the Sort/Merge utility as described in the Sort/Merge usage manual.

Calling Sort/Merge

A sort or merge operation is defined by a sequence of Sort/Merge calls. The first call in the sequence must be either an SM5SORT call or an SM5MERG call. This call specifies which operation (sort or merge) is to be performed. The last call must be an SM5END call. This call terminates the sequence of calls and begins the sort or merge operation.

The remaining calls are optional and can apppear in any order. As a minimum, however, you will generally specify the input file or files containing records to be sorted or merged (CALL SM5FROM), the output file to receive the sorted or merged records (CALL SM5TO), and the data field to be used as the record key (CALL SM5KEY). Other calls enable you to:

Specify an owncode routine to be executed at a predefined point.

Control error processing.

Define your own collating sequence.

Specify sum fields in a file.

The following example shows a sequence of calls that perform a simple sort operation:

```
CALL SM5SORT (ARR)
CALL SM5FROM ('AFILE')
CALL SM5TO ('BFILE')
CALL SM5KEY (1,6,'ASCII','A')
CALL SM5END
```

Records are read from file AFILE, sorted, and written to file BFILE. The record key is six characters long. Records are arranged in ascending order of record key using the ASCII collating sequence.

Record Keys

Sort/Merge uses a record key to determine the order in which records will be written to the output file. The record key is a data field within the record which you select with a call to SM5KEY. SM5KEY specifies the position, length, and data type of the key. The data types that you can specify in the SM5KEY call are described in table 9-1.

Table 9-1. Sort/Merge Data Types

Data Type	Description
NUMERIC_LO	Numeric characters with leading overpunch sign.
NUMERIC_LS	Numeric characters with leading separate sign.
NUMERIC_TO	Numeric characters with trailing overpunch sign.
NUMERIC_TS	Numeric characters with trailing separate sign.
NUMERIC_NS	Numeric characters without a sign.
NUMERIC_FS	Numeric characters with leading blanks and sign.
BINARY	Interpreted as unsigned binary value. Length specified as number of bytes.
BINARY_BITS	Same as BINARY, with length specified as number of bits.
INTEGER	Interpreted as two's complement signed binary value. Length specified as number of bytes.
INTEGER_BITS	Same as INTEGER, with length specified as number of bits.
PACKED	Packed decimal format with sign.
PACKED_NS	Usigned packed decimal format.
REAL	Normalized singleprecision floating-point.

For numeric keys, records are arranged in numeric order. For type character keys, records are arranged using either a predefined collating sequence or a user-defined collating sequence.

The collating sequence used for character keys can be one of the predefined collating sequences or a user-defined collating sequence. The predefined collating sequences are:

ASCII
ASCII6
COBOL6
DISPLAY
EBCDIC
EBCDIC6

Owncode Routines

Sort/Merge allows you to specify owncode routines to be executed at predefined points during Sort/Merge processing. You can write routines to to add, modify, or delete records during the sort or merge operation.

An owncode routine to be executed during Sort/Mege processing must be specified in a call to SM5OWNn, where n is a value that determines when the owncode routine will be called. The parameter n is described in table 9-2. The routine name must be a quoted string. You must include the following parameters in the SUBROUTINE statement of an owncode routine:

SUBROUTINE name (rtn, reca, recla, *recb, reclb*)

rtrn Integer variable through which the routine passes instructions back to Sort/Merge. The value you return in this variable determines what action Sort/Merge is to take. The values you can assign to rtrn, and their meanings, are described in table 9-3.

reca Array in which a record is passed. For owncodes 5, 3, and if SM5FROM files are specified, owncode 1, Sort/Merge passes a record to the routine. For owncode 1, if no SM5FROM files are specified, the routine can return a record to Sort/Merge.

recla Variable through which the length (characters) of reca is passed.

recb Array through which Sort/Merge passes a second record (applies to owncode 5 only; omit for other owncodes).

reclb Length (characters) of recb (applies to owncode 5 only; omit for other owncodes).

Table 9-2. Predefined Points for Execution of Owncode Routines

Value of n in SM50WNn	Used to:	When Executed
1	Retain, alter, or delete the current record, or to add a new record.	After each record read from file specified by SM5FROM or, if no input file specified, when Sort/Merge is ready for another record to sort. (Sort operation only.)
2	Insert a new record or continue from the next file, if any.	After each file specified by SM5FROM has been read or, if no file specified, after owncode 1 indicates that input has terminated. (Sort operation only.)
3	Retain the current record, alter the new record before the current record, or terminate output.	Before writing record to file specified by SM5TO or, if no output file speciied, when Sort/Merge is ready to ouutput a record. (Sort or merge operations.)
4	Insert a new record or complete the sort or merge.	After last record written to file specified by SM5TO or, if no file specified, after owncode 3 routine indicates that output has terminated.
5	Process records with equal keys.	When two records have equal keys.

Table 9-3. Return Codes Set by Owncode Routines

Value of n in SM50WNn	Value of Return Code	Requests Sort/Merge to:
1	0	Sort the record returned in reca.
	1	Delete the record returned in reca.
	2	In input files specified in SM5FROM: Sort both the record passed to owncode 1 and the new record in reca. Re-execute owncode 1 with new record in reca. If no input file specified: Sort record in reca and re-execute own code 1 with this record.
	3	Terminate input. If multiple files specified in SM5FROM, go to next file.
2	0	Terminate input. If multiple files specified in SM5FROM, begin processing next file.
	1	Insert first recla characters from reca after last record and re-execute owncode 2.
3	0	Write record returned in reca to output file (Sort or Merge operations).
	1	Delete record returned in reca (Sort or Merge operations).
	2	Write both record passed to owncode 3 and the new record in reca to output file. Re-execute owncode 3 with new record in reca.
	3	Terminate output. Execute owncode 4 or end Sort/Merge.
4	0	End Sort/Merge.
	1	Insert first recla characters from reca as last record. Re-execute owncode 4.
5	0	Insert first recla characters of reca as first record; insert first reclb characters of recb as second record.
	1	Replace both records with first recla characters of reca.

Sort/Merge Subprogram Calls

The following paragraphs describe the FORTRAN-Sort/Merge subprogram calls in alphabetical order.

SM5E

The SM5E call specifies the file to which Sort/Merge error messages are written. This call has the form:

CALL SM5E (file)

file Character expression specifying file to receive diagnostic messages.

If SM5E is not called, error messages are written to file $ERRORS.

SM5EL

The SM5E call specifies the severity level of errors to be written to the error file. This call has the form:

CALL SM5EL (lim)

lim Severity level of errors to be written to the error file; one of the following values:

'T' All trivial, warning, fatal, and catastrophic errors are written to the error file.

'W' All warning, fatal, and catastrophic errors are written to the error file.

'F' All fatal and catastrophic errors are written to the error file.

'C' Only catastrophic errors are written to the error file.

'NONE' No errors are listed in the error file.

Error messages of the specified severity level or higher are written. The possible error levels are described in the Sort/Merge usage manual. If SM5EL is not called, all error messages are written to the error file.

SM5END

The SM5END call terminates a sequence of Sort/Merge calls and initiates Sort/Merge processing. This call has the form:

CALL SM5END

This call must be the last in the sequence of Sort/Merge calls.

SM5EXC

The SM5EXC call specifies the exception file to which invalid records are written. This call has the form:

CALL SM5EXC (file)

file Character expression specifying a local file to which invalid records are written. The file name must be different from the file named by the SM5TO call.

Invalid records are records found to be out of order by the SM5VER call, records with invalid key or sum fields, and records from a summing overflow. If SM5EXC is not called, the invalid records will be output and the order of the invalid records will be undefined.

SM5FROM

The SM5FROM call specifies one or more files from which records are to be read for sorting or merging. This call has the form:

CALL SM5FROM (file, ..., *file*)

file Character expression specifying a file from which records are to be read for sorting or merging.

Before and after processing, the files have the default NOS/VE positioning.

SM5KEY

The SM5KEY call specifies the key field to be used for the sort or merge. This call has the form:

CALL SM5KEY (first, len, type, ad)

first Integer expression specifying the first byte or bit of the key field (numbered from the left starting with 1).

len Integer expression specifying the number of bytes or bits in the key field.

type Character expression specifying the data type (table 9-1) or, for character data, the predefined collating sequence. The predefined collating sequences are:

ASCII (selects the ASCII collating sequence)

ASCII6 (selects the NOS/VE OSV$ASCII6_FOLDED collating sequence)

COBOL6 (selects the NOS/VE OSV$COBOL6_FOLDED collating sequence)

DISPLAY (selects the NOS/VE OSV$DISPLAY64_FOLDED collating sequence)

EBCDIC (selects the NOS/VE OSV$EBCDIC collating sequence)

EBCDIC6 (selects the NOS/VE OSV$EBCDIC6_FOLDED collating sequence)

ad Character expression specifying the order of the sort or merge operation:

A Ascending order

D Descending order

Output records will be sorted or merged according to the value of the specified key field. If SM5KEY is not called, the entire record is treated as a key field of type ASCII.

SM5LIST

The SM5LIST call specifies the name of the list file. This call has the form:

CALL SM5LIST (file)

file Character expression specifying a local file name of the listing information.

Listing information inludes the Sort/Merge version and level numbers, time and date, error messages, and statistics such as the number of records sorted or merged. If SM5LIST is not called, the default list file is $LIST.

SM5MERG

The SM5MERG call signals the beginning of a sequence of Sort/Merge calls for a merge operation. This call has the form:

CALL SM5MERG (array)

array Integer array in which Sort/Merge returns up to 15 informative statistics about the current operation. In the first word, you must specify the number of values (1 through 15) you want returned. Values are returned in words 2 through 16. The array must be at least two words long, and must be long enough to contain the number of values you requested in the first word.

SM5MERG must be the first routine called for a merge operation.

SM5OFL

The SM5OFL call specifies the fixed length of all records to be supplied by a user-defined routine. This call has the form:

CALL SM5OFL (flen)

flen Integer expression specifying the fixed length, in characters, of all records to be supplied by a user-defined routine.

A fatal error occurs if a user-defined routine supplies a record of any other length. You cannot call both SM5OFL and SM5OMRL for the same sort operation. If neither SM5OFL or SM5OMRL is called, all records supplied by a user-written routine must not exceed the longest allowed record length.

SM5OMRL

The SM5OMRL call specifies the maximum length of any record supplied by a user-defined routine. This call has the form:

CALL SM5OMRL (mlen)

mlen Integer expression specifying the maximum length of all records to be supplied by a user-defined routine.

SM5OMRL need not be called if Sort/Merge has an input or output file with a maximum record length at least as long as the maximum record length of the user-supplied records. You cannot call both the SM5OFL and SM5OMRL routines for the same sort operation. If all records supplied by the user-defined routine have the same length, SM5OFL should be called instead of SM5OMRL.

SM5OWNn

The SM5OWNn call specifies a user-written (owncode) routine to be executed each time point n is reached during the sort or merge. This call has the form:

CALL SM5OWNn (rtn)

n Integer expression having the value 1, 2, 3, 4, or 5. (See table 9-2 for an explanation of this parameter).

rtn Character expression specifying the name of a user-defined routine to be executed each time point n is reached during the sort or merge operation.

SM5RETA

The SM5RETA call specifies whether input records having equal keys are to be written in the same order they are read. This call has the form:

CALL SM5RETA (*opt*)

opt Character expression having one of the following values:

YES Records with equal keys will retain their original order.

NO Records with equal keys will not necessarily retain their original order.

If this argument is omitted (no argument list specified), the default is YES.

If you select the 'YES' option and specify more than one input file, the order in which you specify the input files is the order in which records with equal keys will be written. If you omit the SM5RETA call, records with equal keys might not be written in the same order as they are read.

SM5SEQx

The SM5SEQx calls enable you to define a collating sequence for use by Sort/Merge. Sort/Merge uses the collating sequence to determine the sort or merge order for character data.

A collating sequence consists of a series of value steps from low value to high value. Each value step comprises at lease one character representation. When a value step contains more than one character, all characters within the step are collated equally.

You define a collating sequence by specifying a sequence of SM5SEQx calls. You can define up to 82 collating sequences by specifying a separate series of SM5SEQx calls for each collating sequence. The SM5SEQx routines are:

SM5SEQN	Assigns a name to the collating sequence.
SM5SEQS	Assigns collating positions to the characters in the sequence.
SM5SEQA	Causes characters to be altered on output.
SM5SEQR	Defines a special step in the collating sequence.

The SM5SEQN and SM5SEQS calls are required to define a collating sequence. The SM5SEQA and SM5SEQR calls are optional.

The following descriptions are in alphabetical order.

SM5SEQA

The SM5SEQA call is used with the SM5SEQS call, and causes characters to be altered on output. This call has the form:

CALL SM5SEQA (opt)

opt Character expression having one of the following values:

YES, Y Characters are altered.

NO, N Characters are not altered.

If 'YES' or 'Y' is selected, then the character specified in the first collating position in the preceding SM5SEQS call is substituted for the characters in the remaining positions in the SM5SEQS call. For example, the following alls cause asterisks and ampersands to be changed to slashes on output:

```
CALL SM5SEQS ('/', '*', '$')
CALL SM5SEQA ('YES')
```

A subsequent call to SM5SEQA with 'NO' or 'N' selected restores normal character output.

SM5SEQN

The SM5SEQN call signals the start of a collating sequence definition and specifies the name of the collating sequence. This call has the form:

CALL SM5SEQN (name)

name Your collating sequence name.

The collating sequence name cannot be the same as the predefined collating sequence names and cannot be the same as a collating sequence you have already defined.

SM5SEQR

The SM5SEQR call defines a special step in a collating sequence. The special step consists of the remainder of characters not specified in the preceding SM5SEQS call. This call has the form:

CALL SM5SEQR (opt)

opt Character expression having one of the following values:

YES, Y Define a special step.

NO, N Do not define a special step.

For example, the following calls define a value step that consists of all nondigits and nonletters:

```
CALL SM5SEQS ('0'..'9')
CALL SM5SEQR ('YES')
CALL SM5SEQS ('A'..'Z')
```

These calls define a collating sequence consisting of value steps in the following order: digits in numeric order, nondigits and nonletters, and letters in alphabetical order. The nondigits and nonletters have equal collating positions.

SM5SEQS

The SM5SEQS call assigns collating positions to the characters in a collating sequence. This call has the form:

CALL SM5SEQS (char)

char One or more character expressions whose values are the characters to be assigned to a single step of the collating sequence. Multiple character expressions are separated by commas.

The first SM5SEQS call specifies the first value step (or range of steps), the second SM5SEQS call specifies the second value step (or range of steps), and so on until the collating sequence is completely defined.

For example, the following calls define a collating sequence consisting of three steps. The first step consists of the characters a and A, the second step consists of b and B, and the third step consists of c and C.

```
CALL SM5SEQS ('a', 'A')
CALL SM5SEQS ('b', 'B')
CALL SM5SEQS ('c', 'B')
```

SM5SORT

The SM5SORT call signals the beginning of a sequence of Sort/Merge calls for a sort operation and specifies an array in which Sort/Merge is to return informative statistics about the sort operation. This call has the form:

CALL SM5SORT (array)

array Integer array in which Sort/Merge returns up to 15 informative statistics about the current operation. In the first word, you must specify the number of values (1 through 15) you want returned. Values are returned in words 2 through 16. The array must be at least two words long, and must be long enough to contain the number of values you requested in the first word.

The specific information returned is described in the Sort/Merge usage manual.

SM5ST

The SM5ST call returns the severity level of the most severe error encountered during the sort or merge operation. This call has the form:

CALL SM5ST (lev)

lev Variable in which Sort/Merge returns an integer indicating the highest severity level of all errors detected during the sort or merge:

0	No errors
10	Trivial errors
20	Warning errors
30	Fatal errors
40	Catastrophic errors

SM5SUM

The SM5SUM call specifies the location within the records of the fields that are to be summed. This call has the form:

CALL SM5SUM (first, length, stype, rep)

first First byte or bit of the sum field.

length Number of bytes or bits in the sum field.

stype Name of a numeric data format or collating sequence.

rep Number of times the field should be repeated to the right.

The fields summed are those that belong to records having equal key values. The fields to be summed may contain any type of numeric data except for REAL. Sum fields and key fields cannot overlap. You cannot specify both the SM5RETA and SM5SUM calls in the same sort or merge operation. If SM5SUM is not called, records with equal key values are not combined into a single record.

SM5TO

The SM5TO call specifies the file to receive the sorted or merged output records. This call has the form:

CALL SM5TO (file)

file Character expression specifying the file to receive the sorted or merged records.

Before and after processing, the specified file has the default NOS/VE positioning. If you do not call SM5TO, records are written to file NEW.

SM5VER

The SM5VER call directs Sort/Merge to check the order of your merge input records. The records must be in sorted order. This call has the form:

CALL SM5VER (option)

option Character expression having one of the following values:

YES The order of the merge input records will be checked.

NO The order of the merge input records will not be checked.

If you specify YES and a record is out of order, Sort/Merge issues a diagnostic message, writes the record that is out of order to the exception file, and deletes the record from the output file. Sort/Merge then continues the merge operation.

If you omit the SM5VER call, the order of the merge input records will not be checked.

System Command Language Subprograms

FORTRAN provides a set of subprograms that enable you to communicate with the NOS/VE System Command Language (SCL). The SCL subprograms allow a FORTRAN program to pass values to and receive values from SCL. The values are specified as parameters on the execution command that begins execution of the program. In addition, the SCLCMD call enables you to execute any SCL command from within a FORTRAN program. The following information provides only a brief introduction to the use of SCL parameters. For complete information, refer to the System Command Language Definition manual.

SCL Parameters

An execution command can contain a list of parameters to be used for communicating with SCL. The parameters can be specified positionally or by parameter name, and must be separated by a space or a comma.

Parameters specified by name have the following form:

name=value

where name is the parameter name and value is the parameter value. The parameter value can be specified as:

A single value element

A value set, consisting of a series of value elements separated by commas or spaces and enclosed in parentheses

A value list, consisting of a series of value sets separated by commas or spaces and enclosed in parentheses

A value element is a single value, or a value range represented by a lower bound and an upper bound separated by two periods:

lower-bound..upper-bound

The most common way of specifying a parameter value is by a single value element.

Each SCL parameter value is defined to have a specific kind. The valid value kinds are FILE, NAME, STRING, REAL, INTEGER, BOOLEAN, STATUS, and ANY. The kind of a parameter value must match the kind of value defined for that parameter in the C$ PARAM directive.

With the exception of the predefined $PRINT_LIMIT and STATUS parameters and parameters to be used for file name substitution, each SCL parameter that appears on the execution command must be defined in the source program by the C$ PARAM directive. The $PRINT_LIMIT and STATUS parameters, and the method of file name substitution, are described in chapter 10.

Examples:

```
LGO PAR=3.5
```

The value 3.5 is specified for the parameter PAR.

```
LGO P1=((1,2,3), (4,5,6))
```

A value list is specified for parameter P1. The list contains two value sets, and each value set contains three value elements.

```
CREATE_VARIABLE V KIND=STRING VALUE='AFILE'
.
.
.
LGO FP=V
```

The string variable V is specified for the parameter FP.

FORTRAN provides two classes of SCL subprograms: parameter interface subprograms and variable interface subprograms. The parameter interface subprograms enable you to obtain information about SCL parameters specified on the execution command, including whether or not a particular parameter is present, the type of the parameter, and the parameter values.

The variable interface subprograms are used to retrieve the values of SCL variables and to store values into SCL variables.

An example of a FORTRAN program that uses the SCL subprograms is presented in chapter 11.

C$ PARAM Directive

The C$ PARAM directive defines parameters that are to be specified on the execution command. The general form of the C$ PARAM directive is

C$ PARAM ('pdef, ..., *pdef'*)

where pdef is a valid SCL parameter definition. The format for parameter definitions is described in the SCL Language Definition manual.

The characters C$ must appear in positions 1 and 2, and the string PARAM must begin in or after position 7. The C$ PARAM directive cannot be continued on a subsequent line.

Any parameter that appears on the execution command must have been defined by a C$ PARAM directive in the program to be executed. The C$ PARAM directive defines such parameter properties as:

Parameter name

Whether or not the parameter is required on the execution command

Parameter default values

Parameter kind

Allowable number of value sets

Allowable number of value elements in a value set

A parameter specified on the execution command must conform to its definition in the C$ PARAM directive.

Only one C$ PARAM directive can be specified in a program, and it must appear in the main program. It can be placed anywhere after the PROGRAM statement, and can define one or more parameters. A C$ PARAM directive in a subprogram has no effect.

If a program contains a C$ PARAM directive, the PROGRAM statement should contain only the program name; no file equivalencing can be done.

Example:

```
C$    PARAM ('A:INTEGER; B:VAR OF STRING')
```

This directive defines two parameters. Values specified for parameter A will be of kind integer, and values specified for parameter B will be SCL variables of kind string.

Parameter Interface Subprograms

The parameter interface subprograms retrieve parameter names, parameter values, and other information.

A separate subprogram is provided for retrieving each of the different parameter kinds. You must use the call that corresponds to the kind of the parameter you wish to access. Each subprogram call returns a single value element of a parameter value.

You communicate with SCL through arguments specified in the subprogram calls. Each call requires you to specify certain information about the parameter. Information you need to specify depends on how the parameter was defined in the C$ PARAM directive. This information includes:

Parameter name

Value set number of the desired value (1 if parameter is defined with a single element)

Value number of the desired value (1 if parameter is defined with a single element)

For range parameters, whether the value is the upper or lower range bound.

For subprogram arguments in which character values are returned, values shorter than the argument length are left-justified and blank-filled; longer values are truncated on the right.

The parameter interface subprogram calls are listed below in alphabetical order.

GETBVAL

The GETBVAL call returns the value of an SCL boolean parameter. (SCL boolean corresponds to FORTRAN type logical.) This call has the form:

CALL GETBVAL (param, setnum, valnum, lhi, val, log)

param Character expression specifying the name of the SCL boolean parameter whose value is to be returned. If the length exceeds 31 characters, the excess characters are truncated on the right.

setnum Integer expression specifying the value set number of the requested value. Specify 1 if value sets are not defined for the parameter.

valnum Integer expression specifying the value number within the value set of the requested value. Specify 1 if multiple values are not defined for the parameter.

lhi For range parameters, a character expression specifying whether the requested value is the lower or upper range bound. One of the strings 'LOW' or 'HIGH'. You must specify either 'LOW' or 'HIGH' regardless of whether or not the parameter is defined as a range.

val Variable to receive the value of the specified boolean parameter. Must be declared type logical.

log Integer variable to receive a number indicating how the boolean value was specified. Returns one of the following values:

0 Value specified as TRUE or FALSE
1 Value specified as YES or NO
2 Value specified as ON or OFF

You can use this value to determine the appropriate form to use for responses to the boolean parameter.

GETCVAL

The GETCVAL call returns the value of a STRING, NAME, or FILE parameter. This call has the form:

CALL GETCVAL (param, setnum, valnum, lhi, len, val)

param Character expression specifying the name of the STRING, FILE, or NAME, parameter whose value is to be returned. If the length exceeds 31 characters, the excess characters are truncated on the right.

setnum Integer expression specifying the value set number of the requested value. Specify 1 if value sets are not defined for the parameter.

valnum Integer expression specifying the value number within the value set of the requested value. Specify 1 if multiple values are not defined for the parameter.

lhi For range parameters, character expression specifying whether the requested value is the lower or upper range bound. One of the strings 'LOW' or 'HIGH'. You must specify either 'LOW' or 'HIGH', regardless of whether or not a parameter is defined as a range.

len Integer variable to receive the length of the requested value.

val Character variable to receive the value of the specified parameter. Up to 256 characters are returned. Value shorter than the argument is left justified and blank-filled; value longer is truncated on the right.

GETIVAL

The GETIVAL call returns the value of an SCL integer parameter. This call has the form:

CALL GETIVAL (param, setnum, valnum, lhi, val, rad)

param Character expression specifying the name of the SCL integer parameter whose value is to be returned. If the length exceeds 31 characters, the excess characters are truncated on the right.

setnum Integer expression specifying the value set number of the value to be returned. Specify 1 if value sets are not defined for the parameter.

valnum Integer expression specifying the value number within the value set of the value to be returned. Specify 1 if multiple values are not defined for the parameter.

lhi For range parameters, a character expression specifying whether the requested value is the lower or upper range bound. One of the strings 'LOW' or 'HIGH'. You must specify either 'LOW' or 'HIGH', regardless of whether or not the parameter is defined as a range.

val Integer variable to receive the value of the INTEGER parameter.

rad Integer variable to receive the base (radix) of the integer value. One of the values 2, 8, 10, or 16 is returned. The radix returned is the one specified in the last assignment to the variable; if none was specified, 10 is returned. You can use this value to determine the appropriate radix to use for responses to the integer parameter.

GETSCNT

The GETSCNT function returns an integer indicating the number of value sets specified for the parameter. The function reference has the form:

GETSCNT (param)

param Character expression specifying the name of the parameter for which the number of value sets is to be returned. If the length exceeds 31 characters, the excess characters are truncated on the right.

GETSCNT must be declared type integer in the calling program.

GETSVAL

The GETSVAL call returns the values of an SCL status parameter. This call has the form:

CALL GETSVAL (param, setnum, valnum, lhi, nml, id, cond, len, text)

param Character expression specifying the name of the STATUS parameter for which status values are to be returned. If the length exceeds 31 characters, the excess characters are truncated on the right.

setnum Integer expression specifying the value set number of the value to be returned. Specify 1 if value sets are not defined for the parameter.

valnum Integer expression specifying the value number within the value set of the value to be returned. Specify 1 if multiple values are not defined for the parameter.

lhi Character expression specifying whether the requested value is the lower or upper range bound. One of the strings 'LOW' or 'HIGH'. You must specify either 'LOW' or 'HIGH' regardless of whether or not a range is defined for the parameter.

nml Logical variable to receive the value (TRUE or FALSE) of the NORMAL field. If nml is TRUE, the remaining fields are undefined.

id Character variable to receive the value of the ID field of the status variable. Should be at least two characters long.

cond Integer variable to receive the value of the CONDITION field of the status variable.

len Integer variable to receive the length (characters) of the value returned in the text argument.

text Character variable to receive the value of the text field of the status variable. Up to 256 characters are returned. Value shorter than the argument is left justified and blank-filled; value longer is truncated on the right.

GETVCNT

The GETVCNT function returns the number of values in the specified value set of the specified parameter. The function reference has the form:

param Character expression specifying the name of the parameter for which the number of values in the specified value et is to be returned. If the length exceeds 31 characters, the excess characters are truncated on the right.

setnum Integer expression specifying the value set number for which the number of values is to be returned.

GETVCNT must be declared type integer in the calling program.

GETVREF

The GETVREF call returns a variable reference specified for a parameter. This call has the form:

CALL GETVREF (param, setnum, valnum, lhi, ref, knd, lbnd, ubnd, len)

param Character expression specifying the name of the parameter for which the variable reference is to be returned. If the length exceeds 31 characters, the excess characters are truncated on the right.

setnum Integer expression specifying the value set number of the requested variable reference. Specify 1 if value sets are not defined for the parameter.

valnum Integer expression specifying the value number within the value set of the requested variable. Specify 1 if multiple values are not defined for the parameter.

lhi Character expression specifying whether the requested value is the lower or upper range value. One of the strings 'LOW' or 'HIGH'. You must specify either 'LOW' or 'HIGH' regardless of whether or not a range is defined for the parameter.

ref Character variable to receive the variable reference as it appears on the execution command. Value shorter than the argument is left justified and blank-filled; value longer is truncated on the right.

knd Character variable to receive a string indicating the kind (type) of variable. Contains one of the strings 'STRING', 'INTEGER', 'BOOLEAN', or 'STATUS'.

lbnd Integer variable to receive the lower bound of a variable for which a range is defined.

ubnd Integer variable to receive the upper bound of a variable for which a range is defined.

len Integer variable to receive the length of a string variable.

The argument ref contains the variable name in character format. This variable name can be passed to the variable interface routines, which can retrieve or alter the value of the variable.

SCLKIND

The SCLKIND call returns a string indicating the SCL kind (type) of a parameter. This call has the form:

CALL SCLKIND (param, setnum, valnum, kind)

param Character expression specifying the name of the parameter for which the kind (type) is to be returned. If the length exceeds 31 characters, the excess characters are truncated on the right.

setnum Integer expression specifying the value set number of the parameter value.

valnum Integer expression specifying the value number within the value set of the parameter value.

kind Character variable to receive a string indicating the parameter kind. One of the following values is returned:

'FILE'	'STATUS'
'NAME'	'STRING'
'INTEGER'	'BOOLEAN'
'ANY'	

TSTPARM

The TSTPARM function returns the logical value .TRUE. if the specified parameter appeared on the execution command. Otherwise, TSTPARM returns the value .FALSE. . The TSTPARM function reference has the form:

TSTPARM (param)

param Character expression specifying the name of the parameter to be tested. If the length exceeds 31 characters, the excess characters are truncated on the right.

TSTPARM must be declared type logical.

TSTRANG

The TSTRANG function determines whether a parameter value was specified as a range. The function returns the logical value .TRUE. if the named value was specified on the execution command as a range. Otherwise, TSTRANG returns the logical value .FALSE. The TSTRANG function reference has the form:

TSTRANG (param, setnum, valnum)

param Character expression specifying the name of the parameter to be tested. If the length exceeds 31 characters, the excess characters are truncated on the right.

setnum Integer expression specifying the value set number of the value to be tested. Specify 1 if value sets are not defined for the parameter.

valnum Integer expression specifying the value number within the value set of the value to be tested. Specify 1 if multiple values are not defined for the parameter.

TSTRANG must be declared type logical.

Variable Interface Subprograms

The variable interface subprogram calls are used to retrieve or alter the values of existing SCL variables and to define new variables to be passed to SCL. (SCL variables referenced by the variable interface calls need not be specified on the execution command.) For SCL variables specified as parameter values on the execution command, the GETVREF parameter interface call retrieves variable names, which you specify in subsequent variable interface calls to identify the variable you wish to access.

There are two types of variable interface calls: calls that retrieve the value of a variable (routine names that begin with letters RED) and calls that store a value into a variable (routine names that begin with letters WRT). In addition, the CREV routine is used to define a new variable, and the DELV routine deletes the definition of a variable.

A separate subroutine is provided for each variable kind. When using the variable interface calls to access an SCL variable, you must use the call that corresponds to the kind of that variable.

New variables must be defined by a call to the CREV subroutine before they can be referenced by the variable interface calls.

SCL variables can be defined as scalar variables or arrays. In the following subprograms, the arguments val, bool, len, rad, dig, exponen, norm, id, cond, and text can be variables or arrays, but must be declared with enough locations to receive a value corresponding to each element of the requested SCL variable. When a variable interface subprogram specifies an SCL array, the entire array is referenced; you cannot reference individual elements within an SCL array.

The variable interface subprograms are described below in alphabetical order.

ABORT

The ABORT call sets the fields of the variable specified by the status parameter on the execution command and terminates execution. This call has the form:

CALL ABORT (id, cond, text)

id Character expression whose value is to be placed in the id field of the status parameter. If more than two characters are specified, excess characters are truncated on the right.

cond Integer expression whose value is to be placed in the condition field of the status parameter.

text Character expression whose value is to be stored in the text field of the status parameter. If more than 256 characters are specified, excess characters are truncated on the right.

Invalid data for the id field is printed as ID=XX; invalid data for the condition field is printed as CC=XX.

CREV

The CREV call defines a new SCL variable. This call has the form:

CALL CREV (var, knd, len, lb, ub, scope)

var Character expression specifying the name of the SCL variable being defined. Must be a valid SCL variable name.

knd Character expression specifying the SCL kind (type) of the variable. One of the strings 'STRING', 'INTEGER', 'BOOLEAN', or 'STATUS'.

len For a type STRING variable, an integer expression specifying the length of the variable elements. Maximum value is 256. Not used for non-STRING variable.

lb Integer expression defining the lower bound of the array being defined. Specify 1 if a scalar variable is being defined.

ub Integer expression defining the upper bound of the array being defined. Specify 1 if a scalar variable is being defined.

scope Character expression defining the scope of the variable. One of the following values:

'LOCAL'	Variable is created local to the block.
'XDCL'	Variable is created with the externally declared (XDCL) attribute.
'XREF'	Variable is created with the externally referenced (XREF) attribute.
'JOB'	Variable is created in the job block with the XDCL attribute.
'name'	Variable is created in the utility block specified by name, with the XDCL attribute.

DELV

The DELV call removes the definition of an SCL variable defined by a previous CREV call. This call has the form:

CALL DELV (var)

var Character expression specifying an SCL variable name. Maximum length of an SCL variable name is CHARACTER*256.

If a program unit contains a DELV call, the CREV call that creates the variable must be in the same program unit.

REDBVAR

The REDBVAR call returns the values of an SCL boolean variable. This call has the form:

CALL REDBVAR (ref, dim, val, bool)

ref Character expression specifying a valid SCL boolean variable reference, with the variable name and subscript if appropriate. Maximum length of an SCL variable reference is 256 characters.

dim Integer expression specifying the dimension of val. Specify 1 if val is a variable.

val Logical variable or array to receive the values of the boolean variable. If ref specifies an SCL array, val should contain enough elements to receive a value for each array element.

bool Integer variable or array indicating how the boolean variable was specified; each word receives one of the following numbers:

0	Variable value was specified as TRUE or FALSE
1	Variable value was specified as YES or NO
2	Variable value was specified as ON or OFF

One value is returned for each value of the requested variable.

REDCVAR

The REDCVAR call returns the values of an SCL string variable. This call has the form:

CALL REDCVAR (ref, dim, len, val)

ref Character expression specifying the name of the SCL string variable whose value is to be returned. Must be a valid SCL variable reference, with variable name and subscript if appropriate. Maximum length of an SCL variable reference is 256 characters.

dim Integer expression specifying the dimension of val. Specify 1 if val is a variable.

len Integer variable or array to receive the length of the SCL string variable. If ref specifies an SCL array, len must contain enough elements to receive a value for each array element.

val Character variable or array to receive the value of the SCL variable. If ref specifies an SCL array, val should contain enough elements to receive a value for each element of ref. Each value can be up to 256 characters long. Values shorter than the length of val are left-justified and blank-filled. Longer values are truncated on the right.

REDIVAR

The REDIVAR call returns the values of an SCL integer variable. This call has the form:

CALL REDIVAR (ref, dim, vals, rad)

ref Character expression specifying the name of the SCL integer variable for which values are to be returned. Must be a valid SCL variable reference, with variable name and subscript if appropriate. Maximum length of an SCL variable reference is 256 characters.

dim Integer expression specifying the dimension of val. Specify 1 if val is a variable.

val Integer variable or array to receive the value of the SCL variable; if ref specifies an SCL array, val must contain enough elements to receive all of the array values.

rad Integer variable or array to receive the base (radix) of the SCL integer variable. If ref specifies an SCL array, rad must contain enough elements to receive a value for each value of the array. Each base is one of 2, 8, 10, or 16. The radix returned is the one specified in the last assignment to the SCL variable; if none was specified, 10 is returned.

REDSVAR

The REDSVAR call returns the values of an SCL status variable. This call has the form:

CALL REDSVAR (ref, dim, norm, id, cond, text)

ref Character expression specifying the name of the STATUS variable for which a value is to be returned. Maximum length of a STATUS variable reference is 256 characters.

dim Integer expression specifying the dimension of the norm, id, cond, and text arrays; specify 1 if norm, id, cond, and text are variables.

norm Logical variable or array to receive the value of the NORMAL field of the STATUS variable. If ref specifies an SCL array, norm must contain enough elements to receive a value for each element of ref. If norm is TRUE for a particular value, the corresponding ID, CONDITION, and TEXT fields are undefined.

id Character variable or array to receive the value of the ID field of the status variable. If ref specifies an SCL array, id must contain enough elements to receive a value for each element of ref. Each element of id should be at least two characters long.

cond Integer variable or array to receive the value of the CONDITION field of the status variable. If ref specifies an SCL array, cond must contain enough elements to receive a value for each element of ref.

text Character variable or array to receive the value of the text field of the status variable. If ref specifies an SCL array, text must contain enough elements to receive a value for each element of ref. Up to 256 characters can be returned for each element of ref. Values shorter than the length of the text argument are left-justified and blank-filled. Longer values are truncated on the right.

If norm receives a value of true for a particular status value, the corresponding values of cond, id, and text are undefined.

WRTBVAR

The WRTBVAR call stores values into the specified SCL boolean variable or array. This call has the form:

CALL WRTBVAR (ref, dim, val, bool)

ref Character expression specifying the SCL boolean variable for which values are to be stored. Maximum length of a variable name is CHARACTER*256.

dim Integer expression specifying the dimension of val; specify 1 if val specifies a single value.

val Logical expression specifying the value or values to be stored.

bool Integer expression indicating how the boolean values are specified. The indicators are:

0 Values specified as TRUE or FALSE
1 Values specified as YES or NO
2 Values specified as ON or OFF

You must specify one of the preceding values for each value specified for val.

SCL type boolean corresponds to FORTRAN type logical.

WRTCVAR

The WRTCVAR call stores values into the specified SCL string variable. This call has the form:

CALL WRTCVAR (ref, dim, len, str)

ref Character expression specifying the name of the SCL string variable into which values are to be stored. Maximum length of an SCL variable name is 256 characters.

dim Integer expression specifying the dimension of str; specify 1 if str specifies a single value.

len Integer expression specifying the lengths of the elements of str. You must specify one len value for each element of str.

str Character expression whose value is a string to be stored. If ref specifies an SCL array, str can be an array containing one element for each value to be stored in ref. SCL strings can be up to 256 characters long.

WRTIVAR

The WRTIVAR call stores values into the specified SCL integer variable. This call has the form:

CALL WRTIVAR (ref, dim, val, rad)

ref Character expression specifying the SCL integer variable into which values are to be stored. Must be a valid SCL variable reference, with variable name and subscript if appropriate. Maximum length of an SCL variable reference is 256 characters.

dim Integer expression specifying the dimension of val; specify 1 if val specifies a single value.

val Integer expression specifying the value to be stored in the SCL variable. If ref specifies an array, val can be an array containing one value for each element of ref.

rad Integer expression specifying the base (radix) of the values to be stored.

If val is an array, rad should be an array containing a value for each value being stored.

WRTSVAR

The WRTSVAR call stores values into the specified SCL status variable. This call has the form:

CALL WRTSVAR (ref, dim, norm, id, cond, text)

ref Character expression specifying the name of the status variable into which values are to be stored. Must be a valid SCL variable reference, with variable name and subscript or field name if appropriate. Maximum length of an SCL status variable reference is 256 characters.

dim Integer expression specifying the dimension norm, id, cond, and text; specify 1 if norm, id, cond, and text specify single values.

norm Logical expression specifying the value to be stored in the NORMAL field. If ref is an SCL array, norm can be an array containing one value for each element of ref. Each value must be specified as TRUE or FALSE.

If TRUE is specified for a particular value, the corresponding CONDITION, ID, and TEXT fields are undefined and no values need to be specified for the id, cond, and text arguments.

id Character expression specifying the value to be stored in the ID field. If ref is an SCL array, id can be an array containing one value for each element of ref.

cond Integer expression specifying the value to be stored in the CONDITION field. If ref specifies an array, cond can be an array containing one value for each element of ref.

text Character expression specifying the value to be stored in the TEXT field. If ref specifies an array, text can be an array containing one value for each element of ref. Up to 256 characters can be stored in a text field.

Command Interface Subprogram (SCLCMD)

The SCLCMD call specifies a command string that is passed to SCL and executed. This call has the form:

CALL SCLCMD (string)

string Character expression specifying a valid SCL command string. Maximum length of a command is 256 characters.

Example:

```
CALL SCLCMD ('SETFA FILE=ABC MAXIMUM_RECORD_LENGTH=500')
```

This call executes a SETFA command.

Example:

```
CALL SCLCMD ('DISPLAY_VALUE $FILE (AFILE,FILE_ORGANIZATION)')
```

This example shows how to execute an SCL function from a FORTRAN program. The SCLCMD call executes a DISPLAY_VALUE command which references the $FILE function. The $FILE function displays the value of the FILE_ORGANIZATION attribute of a file named AFILE.

Utility Subprograms

The utility subprograms described in the following paragraphs are supplied by the FORTRAN library. A user-supplied subprogram with the same name as a library subprogram overrides the library subprogram. The following table presents a summary of the FORTRAN utility subprograms:

Category of Subprogram	Subprogram Name	Description
Random Number Generation	RANSET	Initializes RANF function.
	RANGET	Returns current seed of RANF function.
Debugging	DUMP	Produces a memory dump and terminates program.
	PDUMP	Produces a memory dump and returns control to program.
	STRACE	Produces a subprogram traceback.
Error Handling	LEGVAR	Tests a variable for an infinite or indefinite value.
	SYSTEM	Issues a run-time error message.
	SYSTEMC	Alters internal error processing specifications.
	LIMERR	Inhibits program termination for errors caused by invalid input data.
	NUMERR	Returns number of errors that have occurred since last LIMERR call.
Collating Sequence Control	COLSEQ	Selects a collating weight table.
	WTSET	Modifies a collating weight table.
	CSOWN	Defines a collating sequence.
Input/Output Status	UNIT	Checks status of BUFFER IN or BUFFER OUT.
	EOF	Tests for end-of-file.
	LENGTH and LENGTHX	Returns number of words transmitted by previous BUFFER IN or BUFFER OUT.
	IOCHEC	Tests for parity error.
Mass Storage Input/Output	OPENMS	Opens a random access file.
	WRITMS	Writes a record to a random access file.
	READMS	Reads a record from a random access file.
	CLOSMS	Closes a random acces file.
	STINDX	Specifies a subindex for a random access file.
Miscellaneous Intput/Output	MOVLEV	Moves a block of noncharacter data from one area of memory to another.
	MOVLCH	Moves a block of character data from one area of memory to another.
	CONNEC	Connects a file to the terminal.
	DISCON	Disconnects a file from the terminal.

(Continued on next page)

(Continued from previous page)

Category of Subprogram	Subprogram Name	Description
Miscellaneous Utility Subprograms	DATE	Returns the current date.
	JDATE	Returns the current Julian date.
	TIME	Returns the current clock time.
	SECOND	Returns elapsed time (CPU seconds) since beginning of job.
	DISPLA	Places a message in the job log.
	REMARK	Places a message in the job log.
	SSWTCH	Tests the system sense switches.
	EXIT	Terminates program execution.

Random Number Generation

The following subprograms are used in conjunction with the RANF intrinsic function to control the generation of random numbers.

RANSET

The RANSET call specifies the starting value (seed) for the RANF function. This call has the form:

CALL RANSET (n)

n Initial (seed) value; an integer, real, or boolean expression

A subsequent call to RANF uses the value to calculate a random number.

RANSET may alter the seed value you supply; thus, the value returned by a subsequent call to RANGET may differ from the value you specified in a RANSET call.

RANGET

The RANGET obtains the current seed of RANF between 0 and 1. This call has the form:

CALL RANGET (n)

n Real, integer, or boolean variable or array element to receive the current seed value

The value returned can be passed to RANSET at a later time to regenerate the same sequence of random numbers.

Debugging Subprograms

FORTRAN provides several subprograms to aid in the debugging process.

DUMP and PDUMP

The DUMP call and the PDUMP call produce a memory dump on file $OUTPUT in the requested format. These calls have the forms:

CALL DUMP (a, b, f, ..., a, b, f)

CALL PDUMP (a, b, f, ..., a, b, f)

a Variable or array element that is the first word of the area to be dumped.

b Variable or array element that is the last word of the area to be dumped.

f Decimal integer specifying the format of the dump:

0 Hexadecimal dump
1 Real dump
2 Integer dump
3 Octal dump

DUMP and PDUMP are identical, except that PDUMP returns control to the calling program and DUMP terminates program execution. The maximum number of occurences of the triplet a, b, f is 20.

STRACE

The STRACE call provides traceback information. This call has the form:

CALL STRACE

The traceback begins with the subroutine calling STRACE and continues through the chain of calling subroutines, until the main program is reached. Traceback information is written to the file OUTPUT.

Error Handling Subprograms

The following subprograms provide error handling capabilities.

LEGVAR

The LEGVAR function checks the value of a type real variable to determine whether the variable contains an indefinite or infinite (out-of-range) value. The LEGVAR function reference has the form:

LEGVAR (a)

a Real variable

LEGVAR returns one of the following values:

-1 Variable contains an indefinite value.

0 Variable contains a valid value.

1 Variable contains an infinite value.

LEGVAR is of type integer.

SYSTEM

The SYSTEM call enables you to issue a runtime error message. This call has the form:

CALL SYSTEM (ernum, msg)

ernum Error number. An integer expression whose value is in the range 0 through 4999 decimal, specifying an offset from the FORTRAN base error code value 585000.

msg Error message in the form of a character constant

If the specified error number corresponds t one of the FORTRAN runtime error numbers, then when the error occurs the specified message is written to file $ERRORS. Error numbers used by FORTRAN retain the severity associated with them. Error numbers 51 (nonfatal) and 52 (fatal) are reserved for your use. If an error number greater than the highest number defined for FORTRAN, 52 is substituted. If error number zero is specified, the message is ignored and control is returned to the calling program. Each line is printed unless the sum of lines written to $OUTPUT and $ERRORS exceeds the print limit, in which case the job is terminated.

FORTRAN runtime error messages, and associated error numbers, are listed in the Diagnostic Messages for NOS/VE manual.

Example:

CALL SYSTEM (3, 'CHECK DATA')

When runtime error 3 occurs, the message CHECK DATA is printed.

SYSTEMC

The SYSTEMC call enables you to alter the internal specifications that regulate error processing. This call has the form:

CALL SYSTEMC (ernum, slist, recov)

ernum Integer expression whose value determines the error number for which nonstandard recovery is to be implemented. The value must be in the range 0 through 49999, and specifies an offset from the FORTRAN base error code value 585000.

slist Six-word integer array containing the following error processing specifications:

word 1 1 = fatal, 0 = nonfatal.

word 2 Print frequency.

word 3 Frequency increment.

word 4 Print limit.

word 5 Recovery routine selector:

1 = Recovery routine provided.
2 = Recovery routine not provided.

word 6 Maximum traceback limit applicable to all errors. Default is 20.

recov Name of user-written recovery routine. (Routine name must be declared in EXTERNAL statement.)

These specifications are ignored for erroneous data input from a connected (terminal) file.

The following error processing operations can be controlled by SYSTEMC:

Print frequency. The default print frequency value is zero. If the value is changed to n by a call to SYSTEMC, diagnostic and traceback information is listed every nth time until the print limit is reached.

Frequency increment. The default frequency increment is 1. This specification can be changed by a call to SYSTEMC if the call specifies the print frequency as zero. If the frequency increment is zero, diagnostic and traceback information is not listed; if it is 1, such information is listed until the print limit is reached; if it is set to n, information is listed only the first n times unless the print limit is reached first.

Print limit. The default print limit is 10.

Severity level. The severity levels for an error are fatal and nonfatal. The severities of the system-defined errors are given in appendix B. All errors that you define using the listed numbers in a SYSTEM call retain the indicated severity. However, the severity of any error can be changed by a call to SYSTEMC.

Recovery selector. Specifies whether or not a recovery routine is provided. A user-specified error recovery routine activated by a call to SYSTEMC can be canceled by a subsequent call with word 5 of speclist set to zero.

Maximum traceback limit. The default value is 20. If this value is changed to n by a call to SYSTEMC, then when a traceback is issued only the first n routines in the traceback chain are printed.

A negative value for any word in the slist array indicates that the current value of that specification is not to be changed.

If SYSTEMC has been called, an error summary is issued at job termination indicating the number of times each error occurred since the first call to SYSTEMC.

For an error detected by a FORTRAN-supplied math function, a user-supplied error recovery routine should be a function subprogram of the same type as the function detecting the error; the arguments of the functions must agree in number and type. For any other error, a user-supplied error recovery routine should be a subroutine subprogram.

When an error previously referenced by a SYSTEMC call is detected, the following sequence of operations is initiated:

1. Diagnostic and traceback information is printed in accordance with the internal error processing specifications as modified by the SYSTEMC call. The traceback information is terminated for any of the following conditions:

   Calling routine is a main program.

   Maximum traceback limit is reached.

2. If a nonfatal error occurs for which a SYSTEMC call has provided a recovery routine, control returns to the routine that called the routine detecting the error.

3. If the error is fatal, the job is terminated.

4. An error summary is printed at job termination.

Example:

```
EXTERNAL ERRFN
DIMENSION IRAY(6)
DATA IRAY/6*-1/
IRAY(4) = 0
CALL SYSTEMC (115, IRAY, ERRFN)
.
.
.
```

These statements suppress printing of error message 115 and provide a recovery routine named ERRFN.

LIMERR and NUMERR

The LIMERR call and NUMERR function reference enable you to input data to a program without the risk of termination when improper data is input. These calls have the forms:

CALL LIMERR (lim)

NUMERR ()

lim Integer expression whose value is the limit for the number of errors.

When LIMERR is called, the program does not terminate when data errors are encountered until the number of errors occurring after the call exceeds the value of the argument lim.

LIMERR can be used to inhibit job termination when data is input with a formatted, NAMELIST, internal file, or list directed read or with DECODE statements. LIMERR has no effect on the processing of errors in data input from a terminal file since you can correct those interactively.

LIMERR initializes an error count and specifies a maximum limit on the number of data errors allowed before termination. The specified limit continues in effect for all subsequent READ statements until the limit is reached. LIMERR can be reactivated with another call, which will reinitialize the error count and reset the limit. A LIMERR call with lim specified as zero nullifies a previous call; improper data will then result in job termination as usual.

When improper data is encountered in a formatted or namelist READ (or in a DECODE statement) with LIMERR in effect, the bad data field is bypassed, and processing continues at the next field. When improper data is encountered in a list directed READ, control transfers to the statement immediately following the READ statement.

The NUMERR function returns the number of data errors that have occurred since the last LIMERR call. The previous error count is lost when LIMERR is called, and the count is reinitialized to zero. NUMERR is of type integer.

The following example illustrates the use of LIMERR and NUMERR:

```
CALL LIMERR (200)
READ (1,125) (A(I), I=1, 1500)
IF (NUMERR() .GT. 0) THEN
    CALL LIMERR (200)
    .
    .
    .
END IF
READ (1, 125) (B(I), I=1, 1500)
```

When LIMERR is called, a limit of 200 errors is established and the error count is set to zero. After A is read, NUMERR() is checked. If no errors occurred during the read, control transfers to the statement following ENDIF. If errors occurred, LIMERR reinitializes the error count and the rest of the block IF is executed.

Had LIMERR not been called, any invalid data encountered during the read operation would have caused a fatal error.

Collating Sequence Control Subprograms

Character relational expressions are evaluated according to a collating sequence determined by a collation weight table. A collation weight table is a one-dimensional integer array 256 words long with a lower bound of zero. Each element of the weight table has a value between zero and 255 inclusive. The 256 words correspond to the 256 characters of the ASCII character set. The collating weight for the character with the hexadecimal character code of i is the value of the element i of the weight table. The ASCII characters, the corresponding character codes, and the available weight tables are given in appendix B.

The value of a weight table element need not be unique within the table; that is, several characters can have the same collating weight.

You can use a default collating sequence, you can select one of the predefined collating sequences, or you can define your own collating sequence. The default collating sequence is the ASCII (fixed) collating sequence.

To select one of the predefined collating sequences, or to define your own collating sequence, you must specify the DEFAULT_COLLATION=USER parameter on the FORTRAN command, or precede the first character comparison in the program by a C$ COLLATE (USER) directive. Either of these specifications automatically selects the OSV$DISPLAY64_FOLDED collating sequence. You can subsequently select one of the other predefined tables by calling the COLSEQ routine, modify any of the predefined tables by calling the WTSET routine, or define your own table by calling the CSOWN routine.

The relational operations .EQ., .LT., .LE., .GT., and .GE., and the intrinsic functions CHAR and ICHAR use the collating sequence currently in effect as established by the DEFAULT_COLLATION parameter or C$ COLLATE directive, and the routines COLSEQ, WTSET, and CSOWN. The collating sequence used by the intrinsic functions LGE, LGT, LLE, and LLT, always use the ASCII collating sequence regardless of the collating sequence you specify.

COLSEQ

The COLSEQ call selects a processor-defined weight table. This call has the form:

CALL COLSEQ (cexp)

cexp Charater expression whose value when any trailing blanks are removed is one of the following (lowercase characters are equivalent):

ASCII	selects the full 256-character standard ASCII collating sequence (default)
ASCII6	selects the OSV$ASCII6_FOLDED collating sequence
COBOL6	selects the OSV$COBOL6_FOLDED collating sequence
DISPLAY	selects the OSV$DISPLAY64_FOLDED collating sequence
INSTALL	selects the OSV$COBOL6_FOLDED collating sequence

The INSTALL option selects the same collating sequence as COBOL6. The weight tables for the above collating sequences are given in appendix B.

The following example selects a collating sequence identical to COBOL6, with the exception that characters $ and . sort equally ('$' .EQ. '.'):

```
CALL COLSEQ ('COBOL6')
CALL WTSET ('$', ICHAR('.'))
```

The COBOL standard collating sequence is selected, and the entry for the character code $ (24 hex) is reset to 12 (the value of the weight table indexed by 2E hex ('.')). (ICHAR is an intrinsic function that returns the collating weight of the specified character.)

WTSET

The WTSET call modifies the weight table specified in the COLSEQ call. The WTSET call has the form:

CALL WTSET (ind, wt)

ind Character expression of length 1

wt Integer expression with a value in the range 0 through 255

If ind is a character expression with value c, the element of the weight table indexed by the character code of c is replaced with wt.

CSOWN

The CSOWN call defines a partial collating sequence. This call has the form:

CALL CSOWN (str)

str Character expression whose value is a sequence of 1 through 256 characters for which a weight table is to be defined. For a string

c(1)c(2)...c(n)

no c(i) can equal c(j) unless i equals j.

CSOWN explicitly defines the weight table elements for the specified characters and then sets all other elements to zero. For i from 1 through n, the weight of c(i) is set to i-1.

The following example illustrates the creation of a user-defined collating sequence:

```
      CHARACTER CTAB*4
      CTAB(1:1) = 'Z'
      CTAB(2:2) = CHAR(9)
      CTAB(3:3) = '$'
      CTAB(4:4) = '#'
C$    COLLATE (USER)
      CALL CSOWN (CTAB)
```

The characters to be in the collating sequence are stored in the character variable CTAB. Note that the CHAR function is used to convert the element 9 to an ASCII character. (The character corresponding to the value 9 is the horizontal tab character, which does not have a graphic representation.)

The C$ COLLATE (USER) directive must be specified before the CSOWN call; otherwise, the CSOWN call will be ignored. The CSOWN call establishes the new collating sequence, in which Z has weight 0, the horizontal tab has weight 1, $ has weight 2, and # has weight 3. All other characters have weight 0.

Miscellaneous Utility Subprograms

The following subprograms provide for passing information between a user program and the operating system.

DATE

The DATE function returns the current date as the value of the function. The DATE function reference has the form:

DATE ()

The value returned is in the form yyyy-mm-dd, where yyyy is the year, mm is the number of the month, and dd is the day within the month. The value returned is type character with a length of 10. DATE must be declared type CHARACTER*10 in the calling program.

JDATE

The JDATE function returns the current Julian date as the value of the function. The JDATE function reference has the form:

JDATE ()

The value returned has the form yyddd, where yy is the year and ddd is the number of the day within the year. The value returned is type character with a length of 5. JDATE must be declared type CHARACTER*5 in the calling program.

TIME or CLOCK

The TIME function or CLOCK function returns the current reading of the system clock as the value of the function. These functions have the forms:

TIME ()

CLOCK ()

The value returned is in the form hh:mm:ss, where hh is hours from 0 through 23, mm is minutes, and ss is seconds. The value returned is type character with a length of 8. TIME and CLOCK must be declared type CHARACTER*8 in the calling program.

SECOND

The SECOND function returns the central processor time in seconds that has elapsed since the beginning of the job. The SECOND function reference has the form:

SECOND ()

There is no argument, and the result is real.

DISPLA

The DISPLA call places a name and a value in the job log file. This call has the form:

CALL DISPLA (h, k)

h Character expression to be displayed. Must not exceed 256 characters.

k Real or integer expression whose value is to be displayed.

The character constant h cannot be more than 256 characters; k is a real or integer variable or expression and is displayed as an integer or real value.

REMARK

The REMARK call places a message in the job log file. This call has the form:

CALL REMARK (h)

h Character expression to be placed in the job log file

The maximum message length is 256 characters.

SSWTCH

The SSWTCH call tests the system sense switches. This call has the form:

CALL SSWTCH (i, j)

i Integer expression whose value is a sense switch number. The value must be 1 through 6.

j Integer variable or array element to receive a value indicating the setting of the sense switch:

1 Sense switch i is on.

2 Sense switch i is off.

If i is out of range, an informative diagnostic is printed, and j is set to 2. The sense switches are set or reset by operations personnel or by the SET_SENSE_SWITCH command.

EXIT

The EXIT call terminates program execution and returns control to the operating system. This call has the form:

CALL EXIT

NOTE

Use of the STOP statement is preferable to CALL EXIT.

Input/Output-Related Subprograms

These subprograms perform operations closely related to input/output, and are described in detail in chapter 6. The input/output-related subprograms are:

I/O status routines:

UNIT
EOF
LENGTH
LENGTHX
IOCHEC

Mass storage I/O routines:

OPENMS
WRITMS
READMS
CLOSMS
STINDX

Miscellaneous routines:

MOVLEV
MOVLCH
CONNEC
DISCON

Compilation and Execution 10

NOS/VE provides commands for compiling and executing FORTRAN programs. The FORTRAN compiler reads the input source program and produces an output object program. The object program can be loaded into memory and executed by an execution command.

FORTRAN Command

The FORTRAN compiler is called and executed by the FORTRAN command. This command selects a variety of compiler options, including files to be used for input and output and type of output to be produced. The FORTRAN command has the form:

FORTRAN or **FTN**

INPUT=file reference
BINARY_OBJECT=file reference
LIST=file reference
COMPILATION_DIRECTIVES=boolean
DEBUG=keyword value
DEFAULT_COLLATION=keyword value
ERROR=file reference
ERROR_LEVEL=keyword value
EXPRESSION_EVALUATION=list of keyword value
FORCED_SAVE=boolean
LIST_OPTIONS=list of keyword value
MACHINE_DEPENDENT=keyword value
ONE_TRIP_DO=boolean
OPTIMIZATION=keyword value
RUNTIME_CHECKS=list of keyword value
SEQUENCED_LINES=boolean
STANDARDS_DIAGNOSTICS=keyword value
TERMINATION_ERROR_LEVEL=keyword value
STATUS=status variable

The terms file reference, keyword value, boolean, and status variable are defined in the SCL Language Definition manual.

Parameters

Parameters on the FORTRAN command select the desired options. The parameters can be specified either by name or positionally (parameter name omitted), and must be separated by a comma or one or more blanks. If a comma is used, it can be followed by one or more blanks. Parameters specified by name can appear in any order. The required order for parameters specified positionally is shown in the preceding format description.

When you specify a parameter positionally, you must indicate the position of any omitted parameters that precede the specified parameter by commas; the first omitted parameter is indicated by two successive commas, and each additional omitted parameter is indicated by an additional comma. Thus, if n parameters are omitted, n+1 commas are required. For example, the following commands are equivalent:

```
FORTRAN INPUT=SFILE OPTIMIZATION=HIGH

FORTRAN SFILE,,,,,,,,,,,,HIGH
```

The second command specifies the INPUT and OPTIMIZATION parameters positionally; the positions of the omitted BINARY_OBJECT, LIST, and COMPILATION_DIRECTIVES parameters are indicated by four successive commas.

If you omit any parameter from the FORTRAN command a default is automatically provided. Since this default corresponds to the most commonly used option for the parameter, in most cases you need specify only a few parameters or none at all. The parameter names, the processor-supplied defaults, and brief descriptions are presented in table 10-1.

Table 10-1. FORTRAN Command Parameters

Parameter Name	Short Form	Description	Default
BINARY_OBJECT	B	Binary output file	B=$LOCAL.LGO
COMPILATION_DIRECTIVES	CD	C$ directive suppression	Directives recognized.
DEBUG	D	Debugging options	DEBUG=NONE
DEFAULT_COLLATION	DC	Collating sequence control	DC=FIXED
ERROR	E	File to receive error information	E=$ERRORS
ERROR_LEVEL	EL	Severity level of error messages to be printed	EL=W
EXPRESSION_EVALUATION	EE	Order of evaluation of expressions	EE=NONE
FORCED_SAVE	FS	Save variables and arrays in subprograms	Variables and arrays not saved.
INPUT	I	Source input file	I=$INPUT
LIST	L	Output listing file	L=$LIST (batch) L=$NULL (interactive)
LIST_OPTIONS	LO	Output listing options	LO=S
MACHINE_DEPENDENT	MD	Macine dependencies	Machine dependencies not flagged.
ONE_TRIP_DO	OTD	Minimum trip count for DO loops	Zero trip loops.
OPTIMIZATION	OPT	Compiler optimization level	OPT=DEBUG
RUNTIME_CHECKS	RC	Runtime range checking of subscript and substring expressions	Range checking performed.
SEQUENCED_LINES	SL	Sequencing format of source program	Nonsequenced format
STANDARDS_DIAGNOSTICS	SD	Diagnose non-ANSI usages	SD=NONE
STATUS	_	SCL variable to receive error status information	none
TERMINATION_ERROR_LEVEL	TEL	Severity level of errors for which STATUS returned	TEL=F

Unrecognizable parameters prevent compilation from beginning. Conflicting options are either resolved by the compiler or prevent compilation from beginning, depending on the severity of the conflict. Any resolution performed by the compiler is indicated by a job log entry.

You must not specify any parameter more than once. If you do so, the operating system issues an error message and the compilation does not begin.

Parameter Formats

The following paragraphs briefly describe the formats of parameters used on the FORTRAN command. For more information on SCL parameter formats in general, refer to the SCL Language Definition manual.

FORTRAN command parameters have the following general format:

parameter name=value list

There are five general types of parameters used in the FORTRAN command. Four of these types are described below. The fifth type, of which there is only a single parameter (the STATUS parameter), is described under Parameter Options.

The first type of parameter requires you to specify a file reference. These parameters have the form:

parameter name=file reference

A file reference identifies a local or permanent file. A file reference consists of a file path (which includes the file name), a cycle reference (for permanent files), and a file position. The general form of a file reference is:

path.*cycle reference.file position*

The cycle reference and file position are optional. The simplest form of a file reference is a file name. Refer to the SCL Language Definition manual for more information on file references.

Example:

```
FORTRAN INPUT=INFILE BINARY_OBJECT=FAM.USE.BIN
```

This command specifies the file name INFILE for the INPUT parameter; the binary object file BIN is retrieved from family FAM with user name USE.

The second type of parameter requires you to specify a keyword corresponding to the desired option. These parameters have the form:

parameter name=keyword value

For example, the command

```
FORTRAN ERROR_LEVEL=W
```

specifies the keyword W for the ERROR_LEVEL parameter.

The third type of parameter has two possible settings: on or off. These parameters have the form:

parameter name=boolean

To turn the option on, specify parameter name=ON. To turn the option off specify parameter name=OFF. An example of this type of parameter is the COMPILATION_DIRECTIVES parameter. The command

```
FORTRAN COMPILATION_DIRECTIVES=ON
```

selects the option to process compilation directives.

The fourth type of parameter allows you to select a list of options. These parameters have the form:

parameter name=list of keyword value

A particular option is selected by specifying the keyword associated with that option; if the keyword is omitted, the option is not selected. If you select more than one option, each keyword must be separated by a comma and the list of keywords must be enclosed in parentheses. If you select a single option, the parentheses can be omitted. All options for a particular parameter, including the default options, can be deselected by specifying parameter name=NONE. For example, the statement

```
FORTRAN LIST_OPTIONS=(A,O,SA) RUNTIME_CHECKS=NONE
```

selects the A, O, and SA options for the LIST_OPTIONS parameter, and deselects all options for the RUNTIME_CHECKS parameter.

Parameter Names

Each parameter name (except the STATUS parameter) has a long form and an abbreviated form. In addition, some of the parameter keywords have both a long and an abbreviated form. Both forms have the same meaning and either can be specified. For example, the following commands are equivalent:

```
FORTRAN B=BIN TEL=F

FORTRAN BINARY_OBJECT=BIN TERMINATION_ERROR_LEVEL=FATAL
```

The parameter names and associated abbreviations are shown in table 10–1.

File Positioning

The input and output files specified on the FORTRAN command have a default position that can be altered by use of a file position indicator. The file position indicator allows you to specify how a particular file is to be positioned before it is used. The file position indicators are:

$BOI	Positions the file at the beginning-of-information (BOI)
$EOI	Positions the file at the end-of-information (EOI).
$ASIS	Does not position the file (the file is read or written beginning at its current position).

If you omit the file position indicator, the file remains at its current position (or is repositioned according to the OPEN_POSITION file attribute), with the exception that the system file $OUTPUT is positioned at EOI. (Although $OUTPUT is normally positioned at BOI, it is connected to the physical file OUTPUT which is positioned at EOI. As long as this connection remains intact, the effect is to position $OUTPUT at EOI.)

Example:

```
FORTRAN INPUT=SPROG.$ASIS BINARY_OBJECT=:FAM.USE.BIN.$EOI
```

The source input file SPROG is read beginning at the current position. The binary output file BIN is retrieved from family FAM with user name USE, and is positioned at the end-of-information.

Parameter Options

Following are descriptions of the options for each of the FORTRAN command parameters. The parameters are listed in alphabetical order. The heading that precedes each parameter description consists of the parameter name followed by the abbreviation enclosed in parentheses.

BINARY_OBJECT (B)

The BINARY_OBJECT parameter specifies the local file to receive the binary object code produced by the compiler. Options are:

Omitted

Same as BINARY_OBJECT=$LOCAL.LGO.

BINARY_OBJECT=file reference

Binary object code is written to the specified file.

BINARY_OBJECT=$NULL

Object code is written to file $NULL. (The object code is discarded.)

COMPILATION_DIRECTIVES (CD)

The COMPILATION_DIRECTIVES parameter controls the recognition of C$ directives within the source program. Options are:

Omitted

Same as COMPILATION_DIRECTIVES=ON.

COMPILATION_DIRECTIVES=ON

C$ directives are processed.

COMPILATION_DIRECTIVES=OFF

C$ directives are not processed. (They are treated as comments).

DEBUG (D)

The DEBUG parameter selects debugging options. Options are:

Omitted

Same as DEBUG=NONE.

DEBUG=NT

Line number and symbol tables are not placed in the object code. If NT is not selected, line number and symbol tables are generated for use by the interactive Debug facility.

DEBUG=NONE

No options are selected. (Line number and symbol tables are generated along with the object code.)

DEFAULT_COLLATION (DC)

The DEFAULT_COLLATION parameter specifies the weight table to be used for the evaluation of character relational expressions and by the CHAR and ICHAR functions. (See Collating Sequence Control in chapter 9.) Options are:

Omitted

Same as DEFAULT_COLLATION=FIXED.

DEFAULT_COLLATION=USER (DC=U)

A user-specified weight table is used.

DEFAULT_COLLATION=FIXED (DC=F)

The fixed weight table is used.

ERROR (E)

The ERROR parameter specifies the name of the file to receive compiler-generated error information. In the event of an error of ERROR_LEVEL-specified severity or higher, a diagnostic is written to the file specified by the ERROR parameter. If a listing file (LIST parameter) is also specified, the diagnostic is written to both files. Options are:

Omitted

Error information is written to the file $ERRORS.

ERROR=file reference

Error information is written to the specified file.

ERROR_LEVEL (EL)

The ERROR_LEVEL parameter determines the severity level of errors to be printed on the output listing. Selection of a particular option specifies that level and all higher (more severe) levels. Options are (in order of increasing severity):

Omitted

Same as ERROR_LEVEL=WARNING.

ERROR_LEVEL=TRIVIAL (EL=T)

Lists trivial errors. The syntax of these errors is correct but the usage is questionable.

ERROR_LEVEL=WARNING (EL=W)

Lists warning errors. These are errors where the syntax is incorrect but the compiler has made an assumption (such as adding a comma) and continued processing.

ERROR_LEVEL=FATAL (EL=F)

Lists fatal errors. These errors prevent the compiler from processing the statement containing the errors. The compiler continues processing after a fatal error.

ERROR_LEVEL=CATASTROPHIC (EL=C)

Lists catastrophic errors. These errors terminate compilation of the current program unit. Compilation continues with the next program unit.

EXPRESSION_EVALUATION (EE)

The EXPRESSION_EVALUATION parameter controls the way the compiler evaluates expressions.

When evaluating expressions, the compiler normally performs certain optimizations in order to produce more efficient object code. In most cases, these optimizations have no affect on program execution. However, under certain numerically unstable conditions, these optimizations can alter the results of execution. Such conditions usually arise when a program uses values that approach the limits of the computer, or when an operation such as a multiplication or division combines a very large operand with a very small one.

The EXPRESSION_EVALUATION parameter prevents the compiler from performing optimizations which might affect the results of execution.

The EXPRESSION_EVALUATION parameter should be used only when necessary. Options are:

Omitted

Same as EXPRESSION_EVALUATION=NONE.

EXPRESSION_EVALUATION=NONE

Causes no options to be selected.

EXPRESSION_EVALUATION=(op, ..., op)

where op is one of the following:

CANONICAL (C)

Directs the compiler to evaluate expressions strictly according to the precedence rules described in chapter 4. If you select this option, the compiler interprets each expression as if parentheses had been used to completely specify the order in which operations are performed. If you do not select this option, the compiler may reorder operations that are mathematically associative or commutative.

MAINTAIN_EXCEPTIONS (ME)

Prevents the compiler from performing optimizations that eliminate instructions that might cause run-time errors.

MAINTAIN_PRECISION (MP)

Prevents the compiler from performing optimizations that change a floating-point operation to a form that is mathematically equivalent but not computationally equivalent.

REFERENCE (R)

Causes intrinsic functions to be called by reference rather than by value.

You can use the EXPRESSION_EVALUATION parameter to detect numerically unstable conditions such as those described above. If the results obtained with a particular EXPRESSION_EVALUATION option differ significantly from the results obtained without the option, numerical instability exists.

FORCED_SAVE (FS)

The FORCED_SAVE parameter specifies whether or not the values of variables and arrays in subprograms are to be retained after execution of a RETURN or END statement. Note that for subprograms compiled under OPT=DEBUG or OPT=LOW, values are always retained regardless of the FORCED_SAVE option selected. Options are:

Omitted

Same as FORCED_SAVE=OFF.

FORCED_SAVE=ON (FS=ON)

Variable and array values are saved after execution of a RETURN or END statement. This option is equivalent to specifying a SAVE statement in every subprogram compiled.

FORCED_SAVE=OFF (FS=OFF)

Variable and array values are not saved after execution of a RETURN or END statement.

INPUT (I)

The INPUT parameter specifies the name of the file containing the input source code. Options are:

Omitted

Same as INPUT=$INPUT

INPUT=file reference

Source code to be compiled is contained in the specified file

LIST (L)

The LIST parameter specifies the file to receive the compiler output listing. This includes the source listing, diagnostics, compile-time statistics, and information requested by the LIST_OPTIONS parameter. The compiler output listing is described later in this chapter under Compiler Output Listing. Options are:

Omitted

Same as LIST=$LIST (batch jobs) or LIST=$NULL (interactive jobs). (Data written to file $NULL is discarded.)

LIST=file reference

The output listing is written to the specified file.

LIST_OPTIONS (LO)

The LIST_OPTIONS parameter specifies the information that is to appear on the compiler output listing. (This information is described later in this chapter under Compiler Output Listing.) The information is written to the file specified by the LIST parameter. The LIST_OPTIONS parameter allows you to select multiple options. Options are:

Omitted

Same as LO=S.

LIST_OPTIONS=NONE

No output listing is produced.

LIST_OPTIONS=(op, ..., op)

where op is one of the following:

A A listing of the attributes of each symbolic name used or defined within the program is produced. These attributes include data type, class, and so forth.

R A cross reference listing is produced. This listing shows the locations of the definition and use of each symbolic name in the program. Names that are defined but not referenced are not listed.

M A symbol attribute list (same as A option), DO loop map, and common block map are produced. The DO loop portion lists all DO loops in the program, including implied DO lists, and their properties. The common block portion describes the storage layout for common blocks, and the equivalence-induced storage overlap for all variables and arrays.

S A listing of the program source statements is written to the output file. Lines turned off by C$ NOLIST directives are not listed.

SA Same as S, except that lines turned off by C$ NOLIST directives are listed.

O A listing of the generated object code is provided.

MACHINE_DEPENDENT (MD)

The MACHINE_DEPENDENT parameter specifies whether the use of machine dependent capabilities within the program are to be diagnosed and if so, how severely. These capabilities include coding that depends on the number of characters in a word, such as the boolean data type, ENCODE and DECODE statements, and certain uses of BUFFER IN and BUFFER OUT. Options are:

Omitted

Same as MACHINE_DEPENDENT=NONE.

MACHINE_DEPENDENT=NONE

Machine dependent usages are not diagnosed.

MACHINE_DEPENDENT=TRIVIAL (MD=T)

Machine dependent usages are diagnosed as trivial errors.

MACHINE_DEPENDENT=WARNING (MD=W)

Warning messages are printed for machine dependent usages.

MACHINE_DEPENDENT=FATAL (MD=F)

Machine dependent usages are treated as fatal errors, which result in a nonexecutable object program.

ONE_TRIP_DO (OTD)

The ONE_TRIP_DO parameter determines the manner in which DO loops are to be interpreted by the compiler. Options are:

Omitted

Same as ONE_TRIP_DO=OFF.

ONE_TRIP_DO=ON (OTD=ON)

Minimum trip count for DO loops is one.

ONE_TRIP_DO=OFF (OTD=OFF)

Minimum trip count for DO loops is zero.

The trip count is the number of times a DO loop is executed. Specifying ONE_TRIP_DO informs the compiler that all DO loops in the program are executed at least once. This information enables the compiler to generate more efficient object code. However, if ONE_TRIP_DO is specified and the program contains zero trip loops, run-time errors can result. Most DO loops have a trip count of at least one and will execute correctly under either setting of the ONE_TRIP_DO parameter. You can override the ONE_TRIP_DO parameter with the C$ DO directive. Refer to the discussion of DO loops in chapter 5.

OPTIMIZATION (OPT)

The OPTIMIZATION parameter selects the level of optimization performed by the compiler. Options are:

Omitted

Same as OPTIMIZATION=DEBUG.

OPTIMIZATION=DEBUG

Object code is similar to that produced with OPTIMIZATION=LOW, except that it is modified for debugging use.

OPTIMIZATION=LOW

Minimum optimization is performed, resulting in faster compilation time but slower execution time.

OPTIMIZATION=HIGH

Maximum optimization is performed, resulting in slower compilation time but faster execution time.

RUNTIME_CHECKS (RC)

The RUNTIME_CHECKS parameter selects runtime range checking of subscripts and substrings. This parameter allows you to select multiple options. Options are:

Omitted

Same as RUNTIME_CHECKS=(R, S).

RUNTIME_CHECKS=NONE

Causes no options to be selected. If you select NONE, you cannot select any other options.

RUNTIME_CHECKS=R

Selects runtime range checking for character substring expressions. If a character substring expression would cause the substring to exceed the bounds declared by the CHARACTER statement, an informative diagnostic is issued and execution continues.

RUNTIME_CHECKS=S

Selects runtime range checking for subscript expressions. If a subscript expression would cause the subscript to exceed its declared dimension bounds, an informative diagnostic is issued and execution continues.

RUNTIME_CHECKS=(R, S)

Selects both the R and S options.

SEQUENCED_LINES (SL)

The SEQUENCED_LINES parameter specifies the sequencing format of the input source program. Sequenced and nonsequenced formats are described in chapter 2. (Note that the FORTRAN sequenced format is not the same as the line-numbered format produced by NOS/VE. Line numbered source programs must not be written in sequenced format.) Options are:

Omitted

Same as SEQUENCED_LINES=OFF.

SEQUENCED_LINES=ON (SL=ON)

Source program is in sequenced format.

SEQUENCED_LINES=OFF (SL=OFF)

Source program is in nonsequenced format.

STANDARDS_DIAGNOSTICS (SD)

The STANDARDS_DIAGNOSTICS parameter specifies whether the use of non-ANSI source statements are to be diagnosed and if so, how severely. Options are:

Omitted

Same as STANDARDS_DIAGNOSTICS=NONE.

STANDARDS_DIAGNOSTICS=NONE

Nonstandard usages are not diagnosed.

STANDARDS_DIAGNOSTICS=TRIVIAL (SD=T)

Nonstandard usages are treated as trivial errors.

STANDARDS_DIAGNOSTICS=WARNING (SD=W)

Nonstandard usages are treated as warning errors.

STANDARDS_DIAGNOSTICS=FATAL (SD=F)

Nonstandard usages are treated as fatal errors.

Refer to the ERROR_LEVEL parameter for descriptions of trivial, warning, and fatal errors.

STATUS

The STATUS parameter defines an SCL status variable to be set by the compiler to contain information about errors that occurred during compilation. The severity level of errors for which information is to be returned is determined by the TERMINATION_ERROR_LEVEL parameter. The status variable consists of four fields in which the following information is available to SCL after compilation is complete:

Normal field

Contains the boolean value false if compile-time errors occurred, and true if no errors occurred. If the value returned is true (no errors occurred), the remaining fields are undefined.

Identifier field

Contains the string FC (identifying the FORTRAN compiler).

Condition field

A unique integer value indicating the specific error that occurred. Error codes and corresponding error messages are listed in the Diagnostic Messages for NOS/VE manual.

Text field

A string of length 256 in which SCL places information to be substituted into the error message template for the particular error.

Options for the STATUS parameter are:

Omitted

No error status information is returned.

STATUS=var

Status information is placed in the SCL variable var.

TERMINATION_ERROR_LEVEL (TEL)

The TERMINATION_ERROR_LEVEL parameter specifies the minimum error severity level for which the compiler is to return abnormal status. The status code is returned in the SCL variable specified by the STATUS parameter after compilation completes. Information about errors having the specified severity or higher severity is returned. Refer to the ERROR_LEVEL parameter for an explanation of the error severity levels. Options are:

Omitted

Same as TERMINATION_ERROR_LEVEL=FATAL.

TERMINATION_ERROR_LEVEL=TRIVIAL (TEL=T)

Abnormal status is returned for trivial, warning, fatal, and catastrophic errors.

TERMINATION_ERROR_LEVEL=WARNING (TEL=W)

Abnormal status is returned for warning, fatal, and catastrophic errors.

TERMINATION_ERROR_LEVEL=FATAL (TEL=F)

Abnormal status is returned for fatal and catastrophic errors.

TERMINATION_ERROR_LEVEL=CATASTROPHIC (TEL=C)

Abnormal status is returned for catastrophic errors only.

FORTRAN Command Examples

Following are some examples of FORTRAN commands.

Example:

```
FORTRAN INPUT=AFILE BINARY_OBJECT=BFILE ERROR_LEVEL=FATAL
```

This statement selects the following options:

INPUT=AFILE

Source statements are read from file AFILE.

BINARY_OBJECT=BFILE

Object code is written to file BFILE.

ERROR_LEVEL=FATAL

Fatal diagnostics only are written to the output listing file.

All other parameters assume default options. (See table 10-1.)

Example:

```
FORTRAN OPTIMIZATION=HIGH
```

This statement selects the highest level of optimization. All other parameters assume default values. (See table 10-1.)

Example:

```
FORTRAN
```

This statement selects default values for all parameters. (See table 10-1.)

Compiler Output Listing

The listing produced by the FORTRAN compiler is controlled by the LIST_OPTIONS parameter on the FORTRAN command. You can select a complete listing or particular portions of the listing, or you can completely suppress the listing by specifying LIST_OPTIONS=NONE. The following paragraphs describe the output listing options.

Source Listing

The source listing is selected by specifying LIST_OPTIONS=S on the FORTRAN command. If you select the S option, or omit the LIST_OPTIONS parameter altogether, a source listing is produced.

The source listing includes all source lines submitted for compilation as part of the source input file. Listed lines are preceded by a sequence number, unless you specify otherwise through the LINE_NUMBER attribute for the source input file.

You can use the C$ NOLIST and C$ LIST directives (described in appendix F) to suppress the listing of selected source lines. If you select the LIST_OPTIONS=SA option, C$ NOLIST directives are disregarded and all source lines are listed.

If errors are detected during compilation, a descriptive message is printed in the source listing immediately after the statement containing the error.

Compilation Statistics

The compilation statistics follow the source listing. These statistics always appear, and are not affected by the LIST_OPTIONS parameter. The compilation statistics consist of the error summary and the total compilation time.

The error summary has two parts: a diagnostic summary, which lists the number of errors that occurred on each page of the output listing, and the level summary, which lists the total number of each level of error that occurred. The error levels are:

Nonstandard diagnostics
Machine-dependent diagnostics
Trivial diagnostics
Warning diagnostics
Fatal diagnostics
Catastrophic diagnostics

The error summary always appears in the output listing, regardless of which LIST_OPTIONS specification you select.

If the LIST and ERROR parameters both specify the same file (or use the default file, which is the file connected to OUTPUT) each error message, along with the line containing the error, is written twice.

The level summary is followed by the message

```
TOTAL CP SECONDS IN FORTRAN_COMPILATION = n
```

where n is the decimal number of central processor seconds required to compile the program.

If the LIST and ERROR parameters both specify the same file (or the file connected to OUTPUT) each error message (along with the line containing the error) is written twice.

Reference Map

The reference map is a dictionary of all symbolic names appearing in a program unit. The names are grouped by class and listed alphabetically within the groups. The reference map follows the source listing and the diagnostic summary (if present). The map is divided into the following sections:

Symbolic constants map
Namelist map
Variables map
Common and equivalence map
Statement labels map
DO loops map
Entry points map
Procedures map
I/O units map
Unclassified names map

The content of the reference map is controlled by the LIST_OPTIONS parameter on the FORTRAN command. The options are:

LIST_OPTIONS=A

Lists all symbolic names in the source program and their attributes (such as size, data type, relative address, and so forth).

LIST_OPTIONS=R

Lists each symbolic name in the program unit and all references to the name.

LIST_OPTIONS=(A,R)

Combines the A and R options (lists the attributes and references for each symbolic name).

LIST_OPTIONS=M

Produces a DO loop map and common/equivalence map. This option automatically selects the A option; that is, LIST_OPTIONS=M is the same as LIST_OPTIONS=(M,A), and LIST_OPTIONS=(M,R) is the same as LIST_OPTIONS=(M,A,R).

General Format of Maps

Each class of symbolic name is preceded by a subtitle line that specifies the class and the properties listed. Formats for each symbol class are different, but most contain the following information:

Properties of the symbolic name.

References to the symbol (LIST_OPTIONS=R). All line numbers refer to the source line containing the statement in which the reference occurs. A line number may be suffixed by one of the following usage symbols, which describes how the symbolic name was referenced:

- /A Attribute: The reference defines a particular attribute, such as type or dimension.
- /D Declare: The reference declares the name as a dummy argument, an entry point name, an entity in common, a namelist group name, or label of a DO loop terminator.
- /I Input: Name appears as an iolist item whose value will be read but not written.
- /M Modify: Name appears in a statement that alters its contents (assignment statement, DO statement, and so forth)
- /P Parameter: Name appears as an actual argument.
- /R Read: Name is an iolist item in an input statement, a namelist group name in a READ statement, or an I/O unit which is read.
- /S Subscript: Name appears in a subscript expression.
- /W Write: Name is an iolist item in an output statement, a namelist group name in a WRITE, PRINT, or PUNCH statement, or an I/O unit which is written.

Relative address within the program unit of the symbol. This address is given in the form

section + offset

where section is the name of the section containing the symbol and offset is the offset (decimal words) of the symbol within the section, relative to the first word of the section. In most cases, the section name will be a program, subprogram, or common block name, or one of the following symbols:

$LITERAL	Section of the compiled program containing constant data.
$STACK	Section containing variables that are allocated on the stack when the containing program unit is called.
$PARAMETER	A subset of the $STACK section containing parameter list variables allocated on the stack by the calling program.
$STATIC	Section containing variables that are statically allocated, are not in common, and are not in an explicitly named section.
$REGISTER	Variables not belonging to any memory section but existing only in a hardware register.

The following paragraphs describe the various sections of the reference map as they would appear for the full map, selected by LIST_OPTIONS=(M,A,R).

The error summary has two parts: a diagnostic summary, which lists the number of errors that occurred on each page of the output listing, and the level summary, which lists the total number of each level of error that occurred. The error levels are:

Nonstandard diagnostics
Machine-dependent diagnostics
Trivial diagnostics
Warning diagnostics
Fatal diagnostics
Catastrophic diagnostics

The error summary always appears in the output listing, regardless of which LIST_OPTIONS specification you select.

If the LIST and ERROR parameters both specify the same file (or use the default file, which is the file connected to OUTPUT) each error message, along with the line containing the error, is written twice.

The level summary is followed by the message

```
TOTAL CP SECONDS IN FORTRAN_COMPILATION = n
```

where n is the decimal number of central processor seconds required to compile the program.

If the LIST and ERROR parameters both specify the same file (or the file connected to OUTPUT) each error message (along with the line containing the error) is written twice.

Reference Map

The reference map is a dictionary of all symbolic names appearing in a program unit. The names are grouped by class and listed alphabetically within the groups. The reference map follows the source listing and the diagnostic summary (if present). The map is divided into the following sections:

Symbolic constants map
Namelist map
Variables map
Common and equivalence map
Statement labels map
DO loops map
Entry points map
Procedures map
I/O units map
Unclassified names map

The content of the reference map is controlled by the LIST_OPTIONS parameter on the FORTRAN command. The options are:

LIST_OPTIONS=A

Lists all symbolic names in the source program and their attributes (such as size, data type, relative address, and so forth).

LIST_OPTIONS=R

Lists each symbolic name in the program unit and all references to the name.

LIST_OPTIONS=(A,R)

Combines the A and R options (lists the attributes and references for each symbolic name).

LIST_OPTIONS=M

Produces a DO loop map and common/equivalence map. This option automatically selects the A option; that is, LIST_OPTIONS=M is the same as LIST_OPTIONS=(M,A), and LIST_OPTIONS=(M,R) is the same as LIST_OPTIONS=(M,A,R).

General Format of Maps

Each class of symbolic name is preceded by a subtitle line that specifies the class and the properties listed. Formats for each symbol class are different, but most contain the following information:

Properties of the symbolic name.

References to the symbol (LIST_OPTIONS=R). All line numbers refer to the source line containing the statement in which the reference occurs. A line number may be suffixed by one of the following usage symbols, which describes how the symbolic name was referenced:

- /A Attribute: The reference defines a particular attribute, such as type or dimension.
- /D Declare: The reference declares the name as a dummy argument, an entry point name, an entity in common, a namelist group name, or label of a DO loop terminator.
- /I Input: Name appears as an iolist item whose value will be read but not written.
- /M Modify: Name appears in a statement that alters its contents (assignment statement, DO statement, and so forth)
- /P Parameter: Name appears as an actual argument.
- /R Read: Name is an iolist item in an input statement, a namelist group name in a READ statement, or an I/O unit which is read.
- /S Subscript: Name appears in a subscript expression.
- /W Write: Name is an iolist item in an output statement, a namelist group name in a WRITE, PRINT, or PUNCH statement, or an I/O unit which is written.

Relative address within the program unit of the symbol. This address is given in the form

section + offset

where section is the name of the section containing the symbol and offset is the offset (decimal words) of the symbol within the section, relative to the first word of the section. In most cases, the section name will be a program, subprogram, or common block name, or one of the following symbols:

$LITERAL	Section of the compiled program containing constant data.
$STACK	Section containing variables that are allocated on the stack when the containing program unit is called.
$PARAMETER	A subset of the $STACK section containing parameter list variables allocated on the stack by the calling program.
$STATIC	Section containing variables that are statically allocated, are not in common, and are not in an explicitly named section.
$REGISTER	Variables not belonging to any memory section but existing only in a hardware register.

The following paragraphs describe the various sections of the reference map as they would appear for the full map, selected by LIST_OPTIONS=(M,A,R).

Variables Map

Variable names include local and COMMON variables and arrays, and dummy arguments. The variables map has the following form:

```
--VARIABLES--
 -NAME---SECTION+OFFSET-----SIZE-PROPERTIES-----TYPE----REFERENCES

  name   sec+off            size prop1/prop2    type    refs
  .
  .
  .
```

name — Variable name. Variables are listed in alphabetical order.

sec+off — Relative address of the variable

If name is a dummy argument, it is listed as DUMMY ARGUMENT #n, where n is the position of the argument in the dummy argument list.

size — For array names, this entry gives the total number of elements in the array. For nonarray names this entry is blank.

prop — Properties of the name, listed in the form prop1/prop2... . Each prop is one of the keywords:

UND	Variable has not been defined. A variable is defined if any of the following conditions hold: Appears as an entity in common. Is initialized in a DATA statement. Appears on the left side of an assignment statement at the outermost parenthesis level. Is the DO variable in a DO loop or I/O implied DO list. Appears as a parameter in a subroutine or function call. Appears in an input list. Appears as the IOSTAT variable in an I/O statement. Appears as a status variable in an INQUIRE statement. Appears as an extended internal file in an ENCODE statement. Appears as a standard internal file in a WRITE statement. Appears as the destination name in a BUFFER IN statement. Otherwise the variable is considered undefined. However, variables that are referenced before they are defined are not flagged.
EQV	Variable name is equivalenced.
SAV	Variable name has the SAVE property.
UNUSED	Name is not referenced in an executable statement, is not a statement function dummy argument, is not an entity in common, and does not appear as a DO variable in a DATA implied DO list.
S	Name appears only once in the entire program unit. (Check carefully for other names with similar spellings.)

type Gives the type of the variable name. One of the values LOGICAL, INTEGER, REAL, COMPLEX, DOUBLE, CHARACTER, or BOOLEAN.

For character names, the form is:

CHAR*n For variables with specified length

CHAR*(*) For variables with adjustable length

refs References and definitions associated with the variable name; listed by line number. Appears only when R option is selected. May be suffixed by a usage symbol.

Symbolic Constants Map

The symbolic constants map lists information about constants assigned symbolic names by PARAMETER statements. This map has the following form:

```
--SYMBOLIC CONSTANTS--
  -NAME----TYPE-------------------------VALUE---REFERENCES

   name    type                          value   refs
   .
   .
   .
```

name Symbolic name as declared in the PARAMETER statement.

type Data type of the name (same as type field in VARIABLES section).

value Value assigned to the name in the PARAMETER statement.

refs Source line number of statements referencing the constant; appears only if R option selected. Suffixed by a usage symbol (same as refs field in VARIABLES map).

Namelist Map

The namelist map lists information about namelist groups defined in the program unit. This map has the following form:

```
--NAMELIST--
   -NAME----SECTION+OFFSET--------REFERENCES

    name    sec+off               refs
    .
    .
    .
```

name Namelist group name.

sec+off Relative address of the first word of the namelist group.

The symbol *NONE* indicates that the namelist group was not referenced in the program unit.

refs Source line numbers of statements referencing the namelist group. (Appears only if R option selected.) Suffixed by a usage symbol:

/D Namelist group is defined.

/R Namelist is referenced in a READ statement.

/W Namelist is referenced in a WRITE statement.

Common and Equivalence Map

The common and equivalence map lists information about common blocks and equivalence declarations within the program unit. This map is selected by the M option on the LIST_OPTIONS parameter. The common and equivalence map has the following form:

```
  --COMMON+EQUIVALENCE--

 /block/  SIZE = size units save

  item1 ... itemn
  .
  .
  .
--LOCAL EQUIVALENCES--

 item1 ... itemn
 .
 .
 .
```

name	Common block name.
size	Total number of words or characters occupied by the common block.
units	WORDS for a block containing only noncharacter items. CHARACTERS if the block contains character items.
save	SAVE if the block is saved; blank otherwise.
item	Describes the storage position of a variable or array. Each item has three fields: name<first-last> or name<first:last>
name	Name of the variable or array.
first	Storage position within the block of the first element of name.
last	Storage position within the block of the last element of name.

First and last are given in decimal. The first position of a common block is numbered 1. If name is a noncharacter variable (occupies a single word), -last does not appear. If name is type character, first and last indicate character positions, and are separated by a colon. Otherwise, they indicate words and are separated by a hyphen.

Items that share storage positions because of equivalencing are enclosed in parentheses.

Statement Labels Map

The statement labels map lists information about all statement labels used in the program unit. This map has the following form:

```
  --STATEMENT LABELS--
    -LABEL-SECTION+OFFSET-----PROPERTIES-----DEF--REFERENCES-

     label sec+off            prop           def  refs
    .
    .
    .
```

label	Statement label; labels are listed in numerical order.

Revision B Compilation and Execution 10-17

Input/Output Units Map

The input/output units map lists all constant unit designators referenced in the program unit. This map has the following format:

```
--PROCEDURES UNITS--
  -NAME----ALIAS----PROPERTIES------REFERENCES-

   name    alias    prop            refs
   .
   .
   .
```

name — Value of the unit designator. If the value is an integer in the range 0 through 999, then name has the form TAPEn.

alias — The name of the alternate unit specified by an alternate unit specification on the PROGRAM statement or by default (in the case of an implied unit for which no alternate unit was specified on the PROGRAM statement). This field is blank if the unit does not have an alias.

prop — Type of I/O operation for which the unit is used:

FMT	Formatted operation
SEQ	Sequential operation
DIR	Direct access operation
BUF	Buffer I/O operation
AUX	Auxilary I/O statement

Multiple symbols are separated by slashes.

refs — Source line number of statements referencing the unit; appears only if R option selected. May be suffixed by a usage symbol.

Unclassified Names Map

The unclassified names map lists all names appearing in the program unit that could not be classified. In many cases, these names are the result of programming errors such as misspellings, and should be checked carefully. The unclassified names map has the following form:

```
--UNCLASSIFIED NAMES--
  -NAME-----PROPERTIES-----REFERENCES

   name     prop           refs
   .
   .
   .
```

name — Name as it appeared in the source program.

prop — One of the following:

UNUSED	Same as Variable Map property having the same name
S	Same as the Variable Map property having the same name

refs — Source line number where the name is referenced. (Produced only if the R option is selected.) May be suffixed by a usage symbol.

Execution Command

A compiled program is loaded and called into execution by an execution command. The execution command has the form:

file *p ... p*

file Specifies the file containing the object program to be executed.

p Optional parameter to be passed to the executing program.

You can also use the EXECUTE_TASK command to begin execution of a program. This command has the form

EXECUTE_TASK *file params*

where file specifies the name of the object file to be executed, and params is a list of parameters. Refer to the SCL System Interface manual for a detailed description of the EXECUTE_TASK command. Refer also to the Object Code Management manual for information on the system loader.

The name of the object file is established by the BINARY_OBJECT parameter on the FORTRAN command. If you omit this parameter, the file name defaults to $LOCAL.LGO.

The execution command causes the system loader to load the compiled program into memory, perform the required linking and address relocation, and initiate execution of the loaded program. If any parameters are specified on the execution command, they are passed to the program.

The optional parameters on the execution command provide a method of passing values to the program. The parameter list must conform to the format for parameter lists described in the SCL Language Definition manual. Three classes of parameters can appear on the execution call command: predefined parameters (STATUS and $PRINT_LIMIT), file names to be used for file name substitution, and user-defined SCL parameters. You can specify either file names for file name substitution or SCL parameters, but not both, on an execution command.

Parameters on an execution command can be specified by name (in the form parameter name=value) or positionally (parameter name= omitted). When you specify a parameter positionally, you must indicate the position of any omitted parameters that precede the specified parameter by commas; the first omitted parameter is indicated by two successive commas, and each additional omitted parameter is indicated by an additional comma. Thus, if n parameters are omitted, n+1 commas are required.

$PRINT_LIMIT Parameter

The $PRINT_LIMIT parameter appears on the execution command as follows:

file $PRINT_LIMIT=lim

or

file $PL=lim

where file specifies the file containing the compiled object code and lim is the desired print limit. This parameter specifies the maximum decimal number of print lines that the executing program can write to files $OUTPUT and $ERRORS. If the $PRINT_LIMIT parameter is specified positionally, it is the next to last parameter on the execution command.

Example:

```
LGO $PRINT_LIMIT=10000
```

This command sets the runtime print limit to 10000 lines.

STATUS Parameter

The STATUS parameter specifies a System Command Language variable to be used for the error status code returned by the system when runtime errors occur. The STATUS parameter appears on the execution command as follows:

file STATUS=var

where file specifies the object file name and var is a variable to receive the error status code. The STATUS variable consists of the following fields:

Normal field

Returns the logical value true if errors occurred, or false if no errors occurred. Information is returned in the identifier, condition, and text fields only if the normal field contains the value true.

Identifier field

Returns a string of length 2 identifying the processor in control when the error was issued.

Condition field

Returns the error number.

Text field

A field of length 256 characters in which SCL places a string containing delimited substrings to be substituted into the message template for the particular error.

The STATUS parameter positionally the last parameter on the execution command.

Example:

```
LGO STATUS=IERR
```

This statement defines variable IERR to be the error status variable.

User-Defined System Command Language Parameters

You can specify parameters on the execution command that are accessable within the executing program. These parameters provide a method of passing information between an executing program and the System Command Language (SCL). The parameter names and values are accessed by using the System Command Language (SCL) interface calls described in chapter 9. If any user-defined SCL parameters are to appear on an execution command, those parameters must be defined by the C$ PARAM directive within the program to be executed. A given execution command cannot contain both SCL parameters and parameters for file name substitution.

File Name Substitution

FORTRAN provides a method of substituting file names at execution time. File names specified on the execution command are substituted for file names associated with unit names declared on the PROGRAM statement.

Unit names declared on the PROGRAM statement are associated with a default file of the same name unless you substitute a different name. For units INPUT and OUTPUT, the default files are $INPUT and $OUTPUT, respectively. For other units, the default files have the same name as the unit. For example, with the PROGRAM statement

```
PROGRAM TEST(INPUT, OUTPUT, TAPE1, TAPE2)
```

the default runtime file names are:

$INPUT

$OUTPUT

TAPE1

TAPE2

Specifying unit names on the PROGRAM statement is optional; the same unit names would exist if the statement PROGRAM TEST were used. However, default file names associated with the unit names on the PROGRAM statement can be changed for a particular run by using the method of file name substitution.

Parameters to be used for file name substition have the same format as SCL parameters; however, a C$ PARAM directive in the program is not permitted. Each unit declaration on the PROGRAM statement defines a valid parameter that can appear on the execution command. If you specify a parameter and value on the execution command in the form

parameter=value

the file name indicated by value is substituted for the file name associated with the unit name indicated by parameter. If only value is specified, the file name specified by value is substituted for the file associated with the unit name having the corresponding position on the PROGRAM statement. For example, if a program begins with the statement

```
PROGRAM TEST(INPUT, OUTPUT, TAPE1, TAPE2)
```

then the following execution commands have the same effect:

```
LGO,,,MYFILE URFILE

LGO TAPE1=MYFILE TAPE2=URFILE
```

Each command causes the following default associations:

Unit name	File name used
INPUT	$INPUT
OUTPUT	$OUTPUT
TAPE1	MYFILE
TAPE2	URFILE

The commas on the first LGO command are required to indicate the omitted INPUT and OUTPUT parameters.

If an alternate unit name is specified in the PROGRAM statement, that name can be used in place of the parameter name on the execution command. For example, if a program uses the statement

```
PROGRAM TEST (INPUT, TAPE5=INPUT, OUTPUT)
```

then the following execution commands are equivalent:

```
LGO MYFILE URFILE,,CHECK

LGO TAPE5=MYFILE OUTPUT=URFILE STATUS=CHECK
```

The file substitutions are:

Unit name	File name used
INPUT	MYFILE
TAPE5	MYFILE
OUTPUT	URFILE

In the first command, the status variable CHECK is specified positionally (it is last parameter on the command); the omitted $PRINT_LIMIT parameter is indicated by two successive commas. In the second command, the status variable is specified parameter name.

The default unit/file associations can be overriden by file name specifications in OPEN statements.

Debug Utility 11

Debug is a utility program that helps you debug a program during execution.

Introduction to Debug

Debug provides an easy-to-use method of debugging a program while it is executing. Through Debug, you can stop execution at selected points, display or change the value of selected variables and arrays, and then resume execution. The displayed values are formatted according to a default format that is consistent with the formats of FORTRAN values. You specify the symbolic names of the locations you want to display; you don't have to know the machine addresses.

A primary advantage of Debug is that it is easy to use. Using Debug does not require you to modify your source code or know assembly language. Further, using Debug eliminates the need for such conventional debugging techniques as interpreting memory dumps, inserting PRINT statements within a program, and using a load map.

Other Debug features let you

- Change the values of program variables while execution is suspended.
- Display a subprogram traceback list. The traceback list begins with the current subprogram and goes backward through the sequence of called subprograms until the main program is reached.
- Create a file of Debug commands that Debug will execute only if an execution error occurs in your program. If an execution error does not occur, the program executes as though Debug were not being used.

Using Debug involves the following sequence of steps:

1. Compile your program for use with Debug.
2. Turn Debug mode on.
3. Begin the Debug session.
4. Enter Debug commands to debug your program.
5. End the Debug session.
6. Make corrections to your source program, recompile, and, if necessary, conduct additional debug sessions.

The remaining paragraphs in the introduction define what a debug session is and explore the steps listed above.

A Debug session is the sequence of interactions that takes place between you and the Debug utility. The Debug session starts when you begin execution of your program in debug mode. Once your program begins execution in debug mode, control immediately transfers to the Debug, which displays the prompt:

```
DB/
```

and waits for you to enter a Debug command. When you enter a command, Debug processes that command, displays any output or informative messages produced by the command, and then issues another DB/ prompt. This interaction continues until you end the Debug session.

A typical debug session might follow these steps:

1. You specify a name call or EXECUTE_TASK command with DEBUG_MODE=ON to begin the Debug session. Debug gains control and prompts you for a Debug command.
2. You specify the SET_BREAK command with source line numbers to set breaks where you want execution to stop.
3. You begin execution of your program by entering the RUN command. The program executes until either a break is encountered or an execution error occurs. When a break is encountered, execution is suspended.
4. You enter Debug commands to display or change the values of program variables and arrays.

5. You enter the RUN command to resume execution. Execution continues until another break is encountered or an execution error occurs. You can repeat steps 4 and 5 until the program terminates or you are ready to end the Debug session.

6. You enter the QUIT command to end the Debug session. The QUIT command returns control to the operating system.

Here is a simple Debug session:

```
/execute_task (file=lgo, debug_mode=on) <---  Begin the Debug session.
DB/set_break name=b1 line=25 <--------------  Set a break at line 25 in the program.
DB/run <------------------------------------  Begin execution of the program.
 --  DEBUG: break b1 at l=25 <--------------  The break is encountered.
DB/display_program_value name=aval <--------  Display the value of AVAL.
AVAL = 3.14159 <----------------------------  Debug displays the value of AVAL.
DB/run <------------------------------------  Resume execution.
 --  DEBUG: program terminated <------------  The program terminates.
DB/quit <-----------------------------------  End the Debug session.
```

Accessing Debug

You can access Debug while you are developing your program or you can execute your program in such a way that Debug is accessed automatically if your program fails.

Before you begin a Debug session, you must compile your source program for use with Debug.

To compile your program, enter the normal FORTRAN command without the DEBUG and OPTIMIZATION parameters.

The DEBUG parameter controls symbol and line number table generation. It can be omitted because the default setting causes the compiler to generate symbol and line number tables.

When symbol and line number tables are generated, you can reference locations (variables, arrays, and source lines) by their symbolic name rather than by their machine address.

When symbol and line number tables are not generated, you can still use Debug, but you must reference locations by their machine address. For example, the display variable X, you would have to specify the machine address of X. Specifying a machine address is more difficult than specifying a symbolic name.

The OPTIMIZATION parameter controls how much optimization the compiler does. It can be omitted because the default setting causes the compiler to perform minimal optimization.

After you compile your program for use with Debug, you are ready to begin a Debug session.

Remember that individual sites and individual users at a site can change command default settings by putting a SET_PROGRAM_ATTRIBUTES command in the system or user prolog file. The commands shown in the following paragraphs describe the default values that are set when the system was installed.

Accessing Debug During Program Development

There are several ways to access Debug during program development. All of the following sequences are valid:

Sequence 1:

1. Turn Debug on with the SET_PROGRAM_ATTRIBUTES command (described in the Object Code Management Usage manual).

The SET_PROGRAM_ATTRIBUTES command turns Debug on at the job level. This means that all subsequent programs (that do not explicitly specify values at the program level) will be executed under control of Debug, unless you specify the DEBUG_MODE=OFF parameter in a SET_PROGRAM_ATTRIBUTES or CREATE_PROGRAM_DESCRIPTION command.

2. Compile your program for use with Debug (described in the opening paragraphs in the Introduction).

3. Begin execution of your program with a name call (such as LGO) or an EXECUTE_TASK command (both are described in the FORTRAN Usage manual).

Sequence 2:

1. Compile your program for use with Debug.

2. Turn Debug on with a CREATE_PROGRAM_DESCRIPTION command (described in the Object Code Management Usage manual).

   The CREATE_PROGRAM_DESCRIPTION command turns Debug on at the program level. This means that only the specified program (or the implied program if the program or file name is not specified) will be executed under control of Debug.

3. Begin execution of your program with a name call or an EXECUTE_TASK command.

Sequence 3:

1. Compile your program for use with Debug.

2. Turn Debug on and begin execution of your program with the EXECUTE_TASK command.

   The EXECUTE_TASK command turns Debug on at the program level. This means that only the specified program will be executed under control of Debug.

When you access Debug during program development, you have the option to either use the default Debug input and output files or set the Debug input and output file values.

The default input file is COMMAND. For interactive jobs, COMMAND is your terminal. For batch jobs, COMMAND is the normal command stream. You cannot use COMMAND as input to Debug for a batch job because the normal command stream begins at the LOGIN command. Instead, you must copy your input from COMMAND to another file, and then use that file as input to Debug.

To change the Debug input file, specify the DEBUG_INPUT parameter on the SET_PROGRAM_ATTRIBUTES, CREATE_PROGRAM_DESCRIPTION, or EXECUTE_TASK command.

You can change the input file temporarily by entering an INCLUDE_FILE command (described in the SCL Language Definition Usage manual). As soon as the command is entered, commands are read from the specified file until an end-of-partition, an end-of-information, or a RUN command is encountered.

If an end-of-partition or an end-of-file is encountered, commands are again read from the file that contained the INCLUDE_FILE command.

If a RUN command is encountered, program execution is resumed; any remaining commands in the file that was included are not processed. When Debug again gains control, commands are read from the current Debug input file.

You can also change the input file temporarily by specifying the DEBUG_INPUT parameter on the CHANGE_DEFAULTS Debug command. The DEBUG_INPUT parameter changes the command source so that Debug commands are read from the specified file when Debug next gains control. Unlike the INCLUDE_FILE command, the CHANGE_DEFAULTS command has no effect on the current command source.

The default output file for the DEBUG_OUTPUT parameter is $OUTPUT. Initially, $OUTPUT is connected to the actual file OUTPUT. For interactive jobs, the actual file OUTPUT is your terminal. For batch jobs, OUTPUT is the listing file.

To change the Debug output file, specify the DEBUG_OUTPUT parameter on the SET_PROGRAM_ATTRIBUTES, CREATE_PROGRAM_DESCRIPTION, or EXECUTE_TASK command.

$OUTPUT can be connected to other files by specifying the CREATE_FILE_CONNECTION command (described in the SCL System Interface Usage manual). If the standard files $ECHO, $RESPONSE, and $ERRORS are also connected to one of the actual output files, a complete record of a Debug session can be created.

You can use the OUTPUT parameter on the Debug display commands to divert display output to another file; the diversion applies only to the command that contains the OUTPUT parameter.

You can also use the CHANGE_DEFAULTS Debug command to change the Debug output file. The DEBUG_OUTPUT parameter causes Debug to write all output to the specified file; the change takes place as soon as the command is executed.

Accessing Debug on Program Failure

Once you have a working program, you generally want Debug to gain control only if the program unexpectedly fails. The parameters that control Debug when a working program fails are ABORT_FILE and DEBUG_OUTPUT.

You can specify the ABORT_FILE or DEBUG_OUTPUT parameter on the SET_PROGRAM_ATTRIBUTES, CREATE_PROGRAM_DESCRIPTION, OR EXECUTE_TASK command (discussed under Accessing Debug During Program Development and described in the Object Code Management Usage manual).

The default abort file is $NULL, the special system file with no data in it. DEBUG_MODE must be off and ABORT_FILE must be a file other than $NULL for Debug to be activated when a program fails; if ABORT_FILE=$NULL, Debug does not gain control.

Activating Debug when a program fails should facilitate program maintenance. It allows you to associate a specific set of Debug commands with a program so that if the program ever fails, those commands are executed.

Because Debug is a command utility, the file specified in the ABORT_FILE parameter can contain any SCL command, not just Debug commands. This enables you to develop very sophisticated abort procedures.

Debug Concepts

These paragraphs contain miscellaneous information that applies to Debug usage, including status variables, breaks, addressing, debugging optimized code, interrupt processing while debugging, and Debug rings.

STATUS Variable

All Debug commands have an optional parameter called STATUS. When you specify this parameter, a previously declared SCL variable of kind STATUS must be supplied as its value. See the SCL Language Definition manual for a discussion of SCL variables. This variable contains the completion status of the command.

A status variable is a record that contains the following fields:

NORMAL

A boolean that has a value of FALSE if the command could not be processed correctly and a value of TRUE if the command was processed correctly.

IDENTIFIER

A string with a length of 2 that contains the product identifier of the processor in which the error was detected. The product identifier for Debug is DB. This field is undefined when the command is processed correctly.

CONDITION

An integer code that identifies the detected error. The leftmost digits for messages from Debug are 64. This field is undefined when the command is processed correctly.

TEXT

A string with a length of 256 that contains the error message text. This field is undefined when the command is processed correctly.

Usually, the presence of the STATUS parameter on a command causes the next command to be processed even if an error is detected. Debug commands that do not scan for the STATUS parameter when an error is detected are the DELETE_BREAK, SET_BREAK, CHANGE_MEMORY, CHANGE_REGISTER commands. After checking the contents of the status variable, you can use succeeding commands to alter the flow of control based upon the occurrence of error conditions.

Breaks

The primary mechanism for Debug to gain control from an executing program is the user-defined break.

A user-defined break specifies one or more events and an address range such that when a specified event occurs within the address range, program execution is interrupted and Debug takes control.

The SET_BREAK, DELETE_BREAK, and DISPLAY_BREAK commands are used to set, delete, and display break definitions.

The events that can be specified are when execution reaches a specific place, before a branch to a specific location occurs, or before a write into memory.

Address ranges also can be specified in many forms. You cannot set a break for the same event at the same location or overlapping locations.

Once set, a break stays set until it is either explicitly deleted or implicitly deleted with the DELETE_ALL command.

A maximum of 64 breaks can be set at any one time. Because Debug uses some of these breaks itself, the actual number of breaks that you can set varies. A message informs you when a break cannot be set.

Addressing

Debug uses program addresses when addresses are reported in Debug command output, such as when DISPLAY_CALLS or DISPLAY_BREAKS is executed and when Debug gains control. Debug also uses program addresses when addresses are referenced in Debug commands, such as SET_BREAK and DISPLAY_MEMORY.

An important concept that relates to addressing is how the definition of the terms module and procedure differ when talking about the FORTRAN, COBOL, and CYBIL languages.

In FORTRAN, the definitions of module and procedure are identical. Both terms refer to a program, subprogram (subroutine or function), subprogram name, or block data.

In COBOL, the definitions of module and procedure are also identical; both terms refer to a program-name.

In CYBIL, the term module does not mean the same thing as the term procedure. A module is a unit of text that is accepted as input by the loader, linker, or object library generator. A procedure is subprogram.

Reported Addresses

The level of reported addresses is determined by the information available. For FORTRAN programs, the following levels are available by default:

- Module address tables indicating where modules are located.
- Line address tables indicating where code for each line is located.
- Symbol tables indicating where the value of each program name is located.

If you specify DEBUG=(NT) on the FORTRAN command, however, symbol and line number tables are not generated by the compiler. Without symbol or line number addresses, only module and machine level addressing are possible.

Module level addressing may be limited if the module was bound.

Addresses in the message issued when Debug gains control (the break report message) are formatted as follows depending on the information available.

When line and module tables are available (symbolic addressing):

If the address corresponds to the beginning of a line, then

M=module_name L=line_number

otherwise, if the address is somewhere within the line, then

M=module_name L=line_number BO=byte_offset_from_start_of_procedure

When only the module table is available (module addressing):

If the module is not bound, then

M=source_module_name P=procedure_name
BO=byte_offset_from_start_of_procedure

otherwise, if the module is bound, then

M=bound_module_name BO=byte_offset_from_start_of_procedure

When line and module tables are not available (machine addressing):

A=machine_address

within the address formats:

- Module_name and procedure_name correspond to the subroutine or function name, and are, therefore, the same name.
- Line_number corresponds to a line number on the source listing.
- Byte_offset is a decimal number corresponding to the number of bytes beyond the beginning of a line or a hexadecimal number corresponding to the number of bytes beyond the beginning of a procedure or bound module.
- Machine_address is a set of three hexadecimal numbers representing the ring number, segment number, and segment offset of a machine address.

Addresses reported in command output also provide the highest address level possible, but they are not always formatted the same as in break report messages. Addresses shown in DISPLAY_BREAK output are very similar, but addresses shown in DISPLAY_CALLS output contain both the procedure name and line number. Typical DISPLAY_CALLS output might look like the following:

```
-- Traceback from procedure PROC2 module MOD2 at line 34
-- Called from procedure PROC1 module MOD2 at line 55 byte offset 4(16)
-- Called from procedure BEGIN_PROCESS module MOD1 byte offset 1A3(16)
```

Addresses shown in DISPLAY_REGISTERS output are formatted only as hexadecimal addresses in the form

r sss oooooooo

where r is the ring number, sss is the segment number, and oooooooo is the offset from the beginning of the segment. Pointer addresses displayed by DISPLAY_PROGRAM_VALUE are also formatted as hexadecimal machine addresses except for pointers to procedures.

Referenced Addresses

Many Debug commands reference program code and data addresses. For example, SET_BREAK designates an address or address range for break events,
DISPLAY_MEMORY specifies the addresses in memory to be displayed, and
DISPLAY_PROGRAM_VALUE names a program identifier whose value is to be displayed.

Just as for reporting addresses, the capabilities available when referencing program addresses depend on the information available:

- Symbolic addressing is available if line and symbol tables exist. (The tables exist unless DEBUG=NT is specified on the FORTRAN command.)
- Module/procedure offset addressing is available if module tables exist (they always do for user programs).
- Machine level addressing is always available.

Symbolic level addresses can be referenced in different forms from how they will be reported. For example, entry point names, section names, and program names can be referenced, but addresses are never reported in these terms.

Machine level addresses can be referenced only as a single integer (a 12-digit hexadecimal value); they are reported, however, either as a 12-digit hexadecimal integer or as three separate integers corresponding to ring number, segment number, and byte offset from the beginning of the segment.

Not all address forms, however, are used by all commands. For example, the DISPLAY_PROGRAM_VALUE command allows a program name to be referenced by name, including all of the subscripting and qualification syntax. But, the DISPLAY_PROGRAM_VALUE command does not allow machine level addressing. The DISPLAY_MEMORY command, on the other hand, allows machine, module, and almost no symbolic level addressing. The SET_BREAK command allows all forms except names defined in a source program.

The different forms of addresses are specified by different parameters. LINE, MODULE, PROCEDURE, NAME, ENTRY_POINT, SECTION, and ADDRESS are typical address parameter names. Many of these address parameters can be used in combination to specify an address. For example, LINE and MODULE together specify a particular line of a particular module. NAME, MODULE, and PROCEDURE together specify a particular identifier of a particular module in a particular procedure. Similarly, SECTION can be used in conjunction with MODULE. ENTRY_POINT and ADDRESS, however, cannot be used in conjunction with MODULE or with each other because they specify addresses independent of any module. Debug issues an error message if an invalid combination of address parameters is used.

The BYTE_OFFSET parameter can be used to modify the address parameters. For example, the MODULE parameter specifies the first byte of the module; the MODULE parameter modified with BYTE_OFFSET=4, specifies the fifth byte of the module.

Another parameter, BYTE_COUNT, can be used to establish the block size (address range) associated with a referenced address. The BYTE_COUNT parameter indicates how many memory bytes are to be included in the block. For example,

```
SECTION=TEST, BYTE_COUNT=3
```

identifies a three-byte block that begins at section TEST. BYTE_COUNT and BYTE_OFFSET can be used to modify any referenced address except a program name (NAME parameter).

Addressing Bound Modules

Individual modules can be bound (combined) to form a new load module that loads and executes faster than the original separate modules. (For more information, see the CREATE_OBJECT_LIBRARY command in the Object Code Management manual.) Binding modules together has no effect on address reporting or address referencing at the symbolic level; you can debug bound modules in terms of their component module names, line numbers, and identifier names.

Binding does, however, have an effect on module/procedure and module/section offset addressing. After binding, original module and procedure names are not available when the tables that support symbolic addressing are not available; addresses are reported and must be referenced in terms of the new bound module name and byte offsets from the beginning of the module. Code from all original component modules is combined into one code section, static data from all original modules are combined into one static data memory section, and so forth, such that the original component portions of each section cannot be distinguished by Debug. You can deduce where each component portion is by inspecting the section map produced by the GENERATE_LIBRARY subcommand (described in the Object Code Management Usage manual).

Debugging Optimized Code

Most compilers can generate more than one level of object code. The OPTIMIZATION parameter on the compiler call controls the level of object code optimization. The default value is OPTIMIZATION=DEBUG, which generates the most

debuggable object code possible. This level of object code contains a separate packet of machine instructions for each executable source statement, carries no variable values across statement boundaries in registers, notifies Debug each time a new line or procedure is reached, and ensures that Debug can always find actual parameter lists.

If some higher level of optimization is selected, Debug can still function, but with restricted capabilities. For example, you cannot display program identifier values that are permanently allocated to machine registers. When values are temporarily carried in registers between statements, or when code for several source statements is mixed together, displayed values may not be the most recent values. Break report locations may not be as precise either.

Interrupt Processing While Debugging

Three external events can interrupt an executing user program or the Debug utility. These events are pause break, terminate break, and nearly exhausted resource. The following shows the effects of these interrupts.

Pause Break Interrupt

User Program Executing

Debug gains control, reports where the program was executing when the interrupt occurred, and prompts for commands.

Debug Executing

Default system action occurs.

Terminate Break Interrupt

User Program Executing

Debug gains control, reports where the program was executing when the interrupt occurred, and prompts for commands.

Debug Executing

If a Debug command is where the program was executing, that command is terminated and you are prompted for a new command. If Debug is already waiting for a command, the terminate break is ignored.

Nearly Exhausted Resource

User Program Executing

Debug does not get control; default system action occurs.

Debug Executing

Debug does not get control; default system action occurs.

Debug Ring

Debug normally runs in the same ring as the program being debugged. You can, however, control the ring in which Debug executes. The SCL command SET_DEBUG_RING specifies the ring in which Debug executes. The Debug ring cannot be set to a ring more privileged than the lowest ring for which you are validated. If your program runs in one ring, you need not be concerned with the Debug ring except to understand deferred breaks and multiple breaks.

Deferred Breaks

Breaks that occur below the Debug ring are deferred, or delayed, until execution again reaches the Debug ring. The break is deferred so that you do not get control in a ring more privileged than your own. If you were able to get control at a lower ring, you could view or change data that you normally do not have access to, thereby compromising system security.

Deferred breaks can occur even when your program runs in a single ring. Many of the operating system services used by the program execute in more privileged rings. For example, if you set a read or write break on a status variable used in some NOS/VE request and that variable is accessed in a lower ring, the break is delayed until NOS/VE returns control to your program.

When a break is deferred, Debug issues a special break report message. The break is reported as having happened at the line that made the call, and a second line indicating the actual address of the event is output. The second line is formatted as follows:

Trap deferred from <address>

where <address> is where the event actually occurred.

Multiple Breaks

Because breaks below the Debug ring are deferred until control returns to the Debug ring, several breaks can be stacked up before Debug gains control. When this happens, Debug must process multiple breaks.

If there are several unprocessed breaks outstanding when Debug gains control, Debug reports each one in the usual way but honors only the first one to occur. No commands are processed for the most recent breaks, not even commands associated with the break definition, since execution of the commands could destroy the environment that existed when the first break occurred.

Multiple breaks can also occur when execution is not below the Debug ring. For example, two terminal breaks or an execution break and a terminal break could occur before Debug gets control. If this ever occurs, Debug honors only the first break.

Multi-Ring Environment

The ability of Debug to function in a multi-ring environment is limited. If a break event occurs in a lower ring than the Debug ring, Debug gains control, but your options are limited. You can only resume execution of the interrupted procedure or terminate the Debug session.

Debug Commands

This topic describes the Debug commands. The Debug commands follow the syntax and conventions for SCL commands, as described in the SCL Language Definition Usage manual.

The language elements used as parameters are standard SCL elements as defined in that manual, except for source program names used in the DISPLAY_PROGRAM_VALUE command, the CHANGE_PROGRAM_VALUE command, and the value parameters specified on the CHANGE_PROGRAM_VALUE.

Debug Command Summary

Command	Short Form	Description
CHANGE_DEFAULT	CHAD	Changes the default DEBUG input/output files and procedure and module names.
CHANBE_MEMORY	CHAM	Changes the contents of memory.
CHANGE_PROGRAM_VALUE	CHAPV	Changes the value of a program variable.
CHANGE_REGISTER	CHAR	Changes the contents of the P, A, or X registers.
DELETE_BREAK	DELB	Deletes one or more break definitions.

(Continued on next page)

— (Continued from previous page)

Command	Short Form	Description
DISPLAY_BREAK	DISB	Displays specified break definitions.
DISPLAY_CALL	DISC	Displays information about the dynamic call chain.
DISPLAY_DEBUGGING_ENVIRONMENT	DISDE	Displays the environment of your session.
DISPLAY_PROGRAM_VALUE	DISPV	Displays the value of a program identifier.
DISPLAY_REGISTER	DISR	Displays the contents of the P, A, or X registers.
DISPLAY_STACK_FRAME	DISSF	Displays the contents of one or more stack frames.
QUIT	QUI	Terminates the Debug session.
RUN	RUN	Initiates or resumes program execution.
SET_BREAK	SETB	Defines a named break.
SET_STEP_MODE	SETSM	Defines units of a task to be executed in one step.

CHANGE_DEFAULT

The CHANGE_DEFAULT command allows you to change the default module, default procedure, default debug_input file, and default debug_output file. The change remains in effect until altered by another CHANGE_DEFAULT command.

The CHANGE_DEFAULT command has the following format:

CHANGE_DEFAULT or **CHANGE_DEFAULTS** or **CHAD**
MODULE = name or $CURRENT
PROCEDURE = name or $CURRENT
DEBUG_INPUT = file reference
DEBUG_OUTPUT = file reference
STATUS = status variable

Parameters

The parameters for the CHAD command have the following options.

MODULE or *M*

Specifies the program name that is used by default when the MODULE parameter is not specified in Debug commands that must refer to a module. Options are:

Omitted

Causes the current default module to remain unchanged.

MODULE = name

Causes the specified module to be used as the default module.

MODULE = $CURRENT

Causes the default module to be reset to the module that was executing when Debug gains control.

Debug commands that must refer to a module are:

CHANGE_PROGRAM_VALUE
DISPLAY_PROGRAM_VALUE
SET_BREAK
SET_STEP_MODE

The terms module and procedure mean the same thing in FORTRAN. Both terms refer to a program, subprogram (subroutine or function), subprogram name, or block data. Because the MODULE and PROCEDURE parameters name the same program unit, either both parameters should be assigned the same name or both parameters should not be specified.

The terms mean different things in COBOL and CYBIL. Module and procedure are defined for COBOL and CYBIL under Addressing.

PROCEDURE or *P*

Specifies the program name that is used by default when the PROCEDURE parameter is not specified in Debug commands that must refer to a procedure. Options are:

Omitted

Causes the current default procedure to remain unchanged.

PROCEDURE = name

Causes the specified procedure to be used as the default procedure.

PROCEDURE = $CURRENT

Causes the default procedure to be reset to the procedure that was executing when Debug gains control.

Debug commands that must refer to a procedure are:

CHANGE_PROGRAM_VALUE
DISPLAY_PROGRAM_VALUE
SET_STEP_MODE

The terms module and procedure mean the same thing in FORTRAN. Both terms refer to a program, subprogram (subroutine or function), subprogram name, or block data. Because the MODULE and PROCEDURE parameters name the same program unit, either both parameters should be assigned the same name or both parameters should not be specified.

The terms mean different things in COBOL and CYBIL. Module and procedure are defined for COBOL and CYBIL under Addressing.

DEBUG_INPUT or *DI*

Specifies the default file from which Debug commands are read when Debug next gains control. Options are:

Omitted

Causes the current DEBUG_INPUT file to remain unchanged. Unless otherwise specified, the initial DEBUG_INPUT file is COMMAND.

DEBUG_INPUT = file reference

Causes the specified file to be used as the DEBUG_INPUT file.

Unless a file position is specified in the file reference, the file is positioned at the beginning-of-information the first time it is used. The file is not repositioned the next time it is used. Commands are read from the file sequentially. If an end-of-partition or an end-of-file is reached on the input file, program execution resumes.

DEBUG_OUTPUT or *DO*

Specifies the default file on which Debug output is written. The change takes effect immediately. Both break report messages and command output are written to this file. Options are:

Omitted

Causes the current DEBUG_OUTPUT file to remain unchanged. Unless otherwise specified, the initial DEBUG_OUTPUT file is $OUTPUT.

DEBUG_OUTPUT = file reference

Causes the specified file to be used as the DEBUG_OUTPUT file.

Unless a file position is specified in the file reference, the file is positioned at the beginning-of-information the first time it is used. The file is not repositioned the next time it is used.

STATUS

Specifies the return status of the command. Options are:

Omitted

Causes Debug to process the next command if an error does not occur. Causes the return status to be output to $RESPONSE (and to the DEBUG_OUTPUT file if $RESPONSE is connected to that file) if an error does occur. $RESPONSE is normally connected during interactive debugging.

STATUS = status variable

Causes the specified variable to receive the return status.

Examples

Read commands from the file DBIN the next time Debug gains control:

```
DB/change_default debug_input=dbin
```

Write output to file $LIST:

```
DB/change_default debug_output=$list
```

Specify the default module name:

```
DB/chad module=main
```

CHANGE_MEMORY

The CHANGE_MEMORY command changes the contents of memory starting at a specific address. You can change the value of any memory location for which you have write permission.

The CHANGE_MEMORY command has the following format:

CHANGE_MEMORY or **CHAM**
ADDRESS = rsssooooooo(16)
VALUE = integer or string
TYPE = keyword value
REPEAT_COUNT = positive integer > 0 or ALL
STATUS = status variable

Parameters

The parameters for the CHAM command have the following options.

ADDRESS or **A**

Specifies the address of the first byte of memory to be changed. The form is:

ADDRESS = rsssoooooooo(16)

where r is the ring number, sss is the segment number, and oooooooo is the offset from the beginning of the segment. You can obtain machine addresses by using the cross-reference and load maps for your program.

VALUE or **V**

Specifies the new memory value. The form is:

VALUE = string or integer

A string value can be interpreted as a hexadecimal or ASCII string, depending on the TYPE parameter.

A hexadecimal string consists of the hexadecimal digits 0 through 9, A through F, and blanks. Blanks are ignored, but they can be used to improve legibility. Each hexadecimal digit corresponds to 4 bits of memory. The first two digits replace the first byte of memory at the specified address, the second two digits replace the second byte, and so on. If there is an odd number of hexadecimal digits, only the first half of the last byte is changed.

An ASCII string consists of a string of ASCII characters. Each ASCII character corresponds to one byte of memory. The first character replaces the first byte of memory at the specified address, the second character replaces the second byte, and so on.

An integer value completely replaces the contents of eight bytes. A diagnostic message is issued if the integer does not fit into eight bytes.

This parameter is required.

TYPE or *T*

Specifies the type of data defined by the VALUE parameter. Options are:

Omitted

Specifies HEX to be used for string values and INTEGER for numeric values.

TYPE = ASCII or A

Specifies ASCII string values.

TYPE = HEX or H

Specifies hexadecimal string values.

TYPE = INTEGER or I

Specifies integer values.

REPEAT_COUNT or *RC*

Specifies the number of times VALUE is repeated in memory. Options are:

Omitted

Causes a value of 1 to be used.

REPEAT_COUNT = positive integer > 0

Causes the address to be incremented by the value size each time the value is repeated. The memory change is limited to the end of the data segment containing the specified address. Specifying a value that is too large changes all the memory that can be changed.

REPEAT_COUNT = ALL

Causes all the memory that can be changed to be changed.

STATUS

Specifies the return status of the command. Options are:

Omitted

Causes Debug to process the next command if an error does not occur. Causes the return status to be output to $RESPONSE (and to the DEBUG_OUTPUT file if $RESPONSE is connected to that file) if an error does occur. $RESPONSE is normally connected during interactive debugging.

STATUS = status variable

Causes the specified variable to receive the return status.

Examples

Replace four bytes of memory beginning at location 0b02200001112(16) with the hexadecimal string '1010aaab':

```
DB/change_memory address=0b02200001112(16) value='1010aaab'
```

Replace six bytes of memory beginning at location 0b02200000055(16) with the ASCII string 'STRING':

```
DB/change_memory address=0b02200000055(16) ..
DB../value='string' type=ascii
```

Replace eight bytes of memory beginning at location 0b02300000223(16) with the integer value 44:

```
DB/change_memory address=0b02300000223(16) value=44
```

CHANGE_PROGRAM_VALUE

The CHANGE_PROGRAM_VALUE command changes the value of named program variables. Replacement values are entered in the same format as defined in your program, not as they are represented in memory.

The CHANGE_PROGRAM_VALUE has the following format:

CHANGE_PROGRAM_VALUE or **CHAPV**
NAME = variable
VALUE = variable
MODULE = name
PROCEDURE = name
RECURSION_LEVEL = positive integer > 0
RECURSION_DIRECTION = keyword value
STATUS = status variable

Parameters

The parameters for the CHAPV command have the following options.

NAME or **N**

Specifies the name of the program variable in the source program whose value is to be changed. The form is:

NAME = variable

Variable can be any variable defined in your program. You can change simple variables, arrays, array elements, and substrings. Subscripts and substrings references can be variables of type BOOLEAN, COMPLEX, DOUBLE PRECISION, INTEGER, or REAL or constants of type COMPLEX, DOUBLE PRECISION, INTEGER, or REAL. Subscripts or substring references that are constants of type COMPLEX, DOUBLE PRECISION, or REAL must be specified as SCL string variables as discussed below. Subscripts or substring references cannot contain expressions.

The variable must be used in your program.

Named constants (created by the PARAMETER statement) cannot be changed.

SCL string variables can be used to name long program names. To do this, assign the SCL variable to a string containing the identifier. Then use the SCL variable preceded by a question mark as the value of the NAME parameter.

VALUE or **V**

Specifies the new value for the NAME parameter variable. The form is:

VALUE = variable

Usually, the VALUE parameter variable must be the same type as the NAME parameter variable.

The value of any variable can be changed to a constant of the same type, to a variable reference of the same type, or to a named constant (created by the PARAMETER statement) of the same type.

Boolean constants must be entered as hexadecimal integers.

If the value is a string that is shorter than the NAME variable, then the value will be blank-filled on the right to equal the NAME variable length. If the value is longer than the NAME variable, then the value is truncated on the right to equal the NAME variable length.

Unsubscripted arrays can only be changed to a variable reference of the same type and size.

MODULE or *M*

Specifies the name of the module that contains the NAME parameter variable.

Omitted

Causes the default module (the module executing when Debug gained control or the module specified by the CHANGE_DEFAULTS command) to be used.

MODULE = name

Causes the specified name to be used as the module name.

The terms module and procedure mean the same thing in FORTRAN. Both terms refer to a program, subprogram (subroutine or function), subprogram name, or block data. Because the MODULE and PROCEDURE parameters name the same program unit, either both parameters should be assigned the same name or both parameters should not be specified.

The terms mean different things in COBOL and CYBIL. Module and procedure are defined for COBOL and CYBIL under Addressing.

PROCEDURE or *P*

Specifies the name of the procedure that contains the NAME parameter variable.

Omitted

Causes the default procedure (the procedure executing when Debug gained control or the procedure specified by the CHANGE_DEFAULTS command) to be used.

PROCEDURE = name

Causes the specified name to be used as the procedure name.

The terms module and procedure mean the same thing in FORTRAN. Both terms refer to a program, subprogram (subroutine or function), subprogram name, or block data. Because the MODULE and PROCEDURE parameters name the same program unit, either both parameters should be assigned the same name or both parameters should not be specified. If either parameter is not specified, enter a comma to mark the position of the omitted parameter if you plan to specify the STATUS parameter.

The terms mean different things in COBOL and CYBIL. Module and procedure are defined for COBOL and CYBIL under Addressing.

RECURSION_LEVEL or *RL*

Specifies which invocation of a named recursive procedure is to be used. Options are:

Omitted

Causes a value of 1 to be used.

RECURSION_LEVEL = positive integer > 0

Causes the specified invocation to be used. If RECURSION_DIRECTION=FORWARD, then 1 is the first invocation, 2 is the one it invoked, and so on. If RECURSION_DIRECTION=BACKWARD, then 1 is the most recent invocation, 2 is the predecessor, and so on.

Recursion only applies to program variables stored on the stack. Recursion cannot not apply to variables stored in either a common block or the $STATIC section.

RECURSION_DIRECTION or *RD*

Controls how the value of the RECURSION_LEVEL parameter is interpreted. Options are:

Omitted

Causes BACKWARD to be used.

RECURSION_DIRECTION = FORWARD

Causes the value 1 to be used as the first invocation, 2 as the one it invoked, and so on.

RECURSION_DIRECTION = BACKWARD

Causes the value 1 to be used as the most recent invocation, 2 as the predecessor, and so on.

Recursion only applies to program variables stored on the stack. Recursion cannot not apply to variables stored in either a common block or the $STATIC section.

STATUS

Specifies the return status of the command. Options are:

Omitted

Causes Debug to process the next command if an error does not occur. Causes the return status to be output to $RESPONSE (and to the DEBUG_OUTPUT file if $RESPONSE is connected to that file) if an error does occur. $RESPONSE is normally connected during interactive debugging.

STATUS = status variable

Causes the specified variable to receive the return status.

Examples

The first example refers to the following definition:

```
COMMON /BLK/ DVAL, RVAL, IVAL, ZVAL
DATA DVAL, RVAL, IVAL, ZVAL /20.0D+0, 3.45E+01, 30, (+20.0,20.3)/
```

Display initial value of variable DVAL:

```
DB/display_program_value name=dval
dval = 20.
```

Change the value of variable DVAL to 30.0:

```
DB/change_program_value name=dval value=+30.0d+0
```

Display the new value of DVAL:

```
DB/dispv dval
dval = 30
```

Change the value of variable INDEX:

```
DB/chapv name=index value=63 module=ff_pp
```

Change the value of logical variable VAR:

```
DB/change_program_value var value=true
```

CHANGE_REGISTER

The CHANGE_REGISTER command changes the value of the P, A, or X registers that are associated with the program executing when Debug gained control.

The CHANGE_REGISTER command has the following format:

CHANGE_REGISTER or **CHANGE_REGISTERS** or **CHAR**
KIND = keyword value
NUMBER = integer or range of integers or ALL
VALUE = integer or string
TYPE = keyword value
STATUS = status variable

Parameters

The parameters of the CHAR command have the following options.

KIND or *K*

Specifies the register to change. Options are:

Omitted

Specifies the P register to change.

KIND = P

Specifies the P register to change.

KIND = A

Specifies the A registers to change.

KIND = X

Specifies the X registers to change.

NUMBER or *N*

Indicates which A or X registers specified by the KIND parameter will change. If the KIND parameter is not specified, the P register is assumed and this parameter is ignored.

The A and X registers can be saved and changed when Debug gains control. However, some registers are not always saved; a message is issued for each register that cannot be changed because it was not saved.

Options are:

Omitted

Specifies the zero register.

NUMBER = integer or range of integers

Specifies a set of registers. Integer can be 0 through 15.

NUMBER = ALL

Specifies all A and X registers.

VALUE or **V**

Specifies the new value of the register. Options are:

Omitted

Same as integer or string allowed for KIND=P.

If KIND = P or A, then VALUE = integer or string

Integer can be an integer in the range 0 through 0FFFFFFFFFFF(16).

String can be a hexadecimal string containing a maximum of 12 hexadecimal digits (blanks are ignored); each hexadecimal digit corresponds to 4 bits.

The upper 4 bits are ignored when changing the P register because the ring number in P cannot be changed.

If KIND = X, then VALUE = integer or string

Integer can be an integer in the range -7FFFFFFFFFFFFFFF(16) through 7FFFFFFFFFFFFFFF(16).

String can be a hexadecimal string containing a maximum of 16 hexadecimal digits (blanks are ignored); each hexadecimal digit corresponds to 4 bits.

An ASCII string containing a maximum of eight ASCII characters; each character corresponds to one byte.

The upper bits of the register are set to zero if an integer is negative or to 1 if an integer is positive when the value does not fill the register. A string value is left-justified with remaining bytes unchanged.

TYPE or *T*

Specifies the type of data specified by the VALUE parameter. Options are:

Omitted

Specifies HEX to be used for string values and INTEGER to be used for numeric values.

TYPE = ASCII or A

Specifies an ASCII string.

TYPE = HEX or H

Specifies a hexadecimal string.

TYPE = INTEGER or I

Specifies an integer string.

STATUS

Specifies the return status of the command. Options are:

Omitted

Causes Debug to process the next command if an error does not occur. Causes the return status to be output to $RESPONSE (and to the DEBUG_OUTPUT file if $RESPONSE is connected to that file) if an error does occur. $RESPONSE is normally connected during interactive debugging.

STATUS = status variable

Causes the specified variable to receive the return status.

Examples

Change the current value of the P register to 0A02200004500(16). The upper 4 bits for the ring number are ignored.

```
DB/change_register kind=p value=0a02200004500(16)
```

Change the current value of the X7 register to 'ABCDEFGH':

```
DB/char kind=x number=7 value='abcdefgh' type=ascii
```

DELETE_BREAK

The DELETE_BREAK command deletes one or more break definitions.

The DELETE_BREAK command has the following format:

DELETE_BREAK or **DELETE_BREAKS** or **DELB**
BREAK = list of names or ALL
STATUS = status variable

Parameters

The parameters of the DELB command have the following options.

BREAK or **BREAKS** or **B**

Specifies the break definitions to be deleted. Options are:

BREAK or BREAKS = list of names

The break associated with each name is deleted. If the keyword ALL appears in the list of names, all breaks are deleted. An informative message is issued if a specified break name does not exist, however, all subsequent breaks in the list will be processed.

BREAK or BREAKS = ALL

All breaks are deleted.

STATUS

Specifies the return status of the command. Options are:

Omitted

Causes Debug to process the next command if an error does not occur. Causes the return status to be output to $RESPONSE (and to the DEBUG_OUTPUT file if $RESPONSE is connected to that file) if an error does occur. $RESPONSE is normally connected during interactive debugging.

STATUS = status variable

Causes the specified variable to receive the return status.

Examples

Delete break definitions B1, B2, and B3:

```
DB/delete_breaks breaks=(b1,b2,b3)
```

Delete all break definitions:

```
DB/delete_breaks all
```

Delete break definition B4:

```
DB/delete_break b4
```

DISPLAY_BREAK

The DISPLAY_BREAK command displays specified break definitions. The break name, events, address, and any commands associated with the break are displayed.

The DISPLAY_BREAK command has the following format:

DISPLAY_BREAK or **DISPLAY_BREAKS** or **DISB**
BREAK = list of names or ALL
OUTPUT = file reference
STATUS = status variable

Parameters

The parameters of the DELB command have the following options.

BREAK or *BREAKS* or *B*

Specifies the break definitions to be deleted. Options are:

BREAK or BREAKS = list of names

The break associated with each name is deleted. If the keyword ALL appears in the list of names, all breaks are deleted. An informative message is issued if a specified break name does not exist, however, all subsequent breaks in the list will be processed.

BREAK or BREAKS = ALL

All breaks are deleted.

OUTPUT or *O*

Specifies the file on which the break definitions are written. Options are:

Omitted

Causes the current default Debug output file to be written to.

OUTPUT = file reference

Causes the specified file to be written to.

STATUS

Specifies the return status of the command. Options are:

Omitted

Causes Debug to process the next command if an error does not occur. Causes the return status to be output to $RESPONSE (and to the DEBUG_OUTPUT file if $RESPONSE is connected to that file) if an error does occur. $RESPONSE is normally connected during interactive debugging.

STATUS = status variable

Causes the specified variable to receive the return status.

Examples

Display break definitions B1, B2, and B5:

```
DB/display_breaks breaks=(b1,b2,b5)
```

Display all break definitions:

```
DB/display_breaks
```

Debug outputs the following:

```
-- Break B1
-- event(s) = execution
-- location: M=module_main L=26

-- Break B2
-- event(s) = execution
-- location: M=m L=13 B0=16

-- Break B3
-- event(s) = execution
-- location: M=m L=16

-- Break B4
-- event(s) = execution
-- location: M=perform_integer_multiplications L=7
```

DISPLAY_CALL

The DISPLAY_CALL command displays information about the dynamic call chain. Usually the procedure name, module name, and line number of each call are shown. Only the procedure or module name and byte offset from the beginning of the procedure or module are shown if you inhibit Debug tables when compiling your program. Only machine addresses are shown for internal NOS/VE calls.

The DISPLAY_CALL command has the following format:

DISPLAY_CALL or **DISPLAY_CALLS** or **DISC**
COUNT = positive integer > 0 or ALL
START = integer
OUTPUT = file reference
STATUS = status variable

Parameters

The parameters of the DISC command have the following options.

COUNT or *C*

Specifies the number of calls to be displayed. Options are:

Omitted

Causes all calls to be displayed.

COUNT = positive integer > 0

Causes the specified number of calls to be displayed; all calls are displayed if the integer is greater than the number of existing calls.

COUNT = ALL

Causes all calls to be displayed.

START or *S*

Specifies the call on the chain to be displayed first. Thus, it is possible to skip the most recent calls. Options are:

Omitted

Causes the most recent call to be displayed.

START = positive integer > 0

Causes the specified call to be displayed. One represents the most recent call, two represents the predecessor of the most recent call, and so forth.

An informative message is issued if the specified number of calls is greater than the actual number of calls.

OUTPUT or *O*

Specifies the file on which the call information is written. Options are:

Omitted

Causes the current DEBUG_OUTPUT file to be written to.

OUTPUT = file reference

Causes the specified file to be written to.

STATUS

Specifies the return status of the command. Options are:

Omitted

Causes Debug to process the next command if an error does not occur. Causes the return status to be output to $RESPONSE (and to the DEBUG_OUTPUT file if $RESPONSE is connected to that file) if an error does occur. $RESPONSE is normally connected during interactive debugging.

STATUS = status variable

Causes the specified variable to receive the return status.

Examples

Display the first four calls on the call chain:

```
DB/display_calls count = 4
```

Display all the calls on the call chain beginning with the second most recent call:

```
DB/display_calls start = 2
```

DISPLAY_DEBUGGING_ENVIRONMENT

The DISPLAY_DEBUGGING_ENVIRONMENT command can display the following: current defaults for module, procedure, DEBUG_INPUT, DEBUG_OUTPUT, total number of breaks you have set and Debug has set, information about STEP_MODE, and the location in your program where execution has stopped.

The DISPLAY_DEBUGGING_ENVIRONMENT has the following format:

DISPLAY_DEBUGGING_ENVIRONMENT or **DISDE**
DISPLAY_OPTIONS = list of keyword values
OUTPUT = file reference
STATUS = status variable

Parameters

The parameters of the DISDE command have the following options.

DISPLAY_OPTIONS or *DO*

Specifies the type of information to be displayed. Options are one or more of the following:

Omitted

Causes ALL to be used.

DISPLAY_OPTIONS = DEFAULTS or D

Specifies that the current default values for module, procedure, DEBUG_INPUT, and DEBUG_OUTPUT are to be listed.

Unless the CHANGE_DEFAULTS command has been specified, the default module and procedure is where execution has stopped in your task. The text $CURRENT is output if module or procedure has not been initialized.

DISPLAY_OPTIONS = BREAKS or B

Specifies that the number of breaks you have set, the number of breaks currently in use by Debug, and the maximum number of allowed breaks are to be listed.

DISPLAY_OPTIONS = STEP_MODE or SM

Specifies that the current STEP_MODE attributes are to be listed.

DISPLAY_OPTIONS = USER_ADDRESS or UA

Specifies that the location where execution has stopped in your program is to be listed.

DISPLAY_OPTIONS = ALL

Specifies that defaults, breaks, STEP_MODE attributes, and user_addresses are to be listed.

OUTPUT or *O*

Specifies the file where the debugging environment display is written. options are:

Omitted

Causes the current default Debug output file to be written to.

OUTPUT = file reference

Causes the specified file to be written to.

STATUS

Specifies the return status of the command. Options are:

Omitted

Causes Debug to process the next command if an error does not occur. Causes the return status to be output to $RESPONSE (and to the DEBUG_OUTPUT file if $RESPONSE is connected to that file) if an error does occur. $RESPONSE is normally connected during interactive debugging.

STATUS = status variable

Causes the specified variable to receive the return status.

Examples

Display the number of breaks set, the number of breaks in use by Debug, the maximum number of allowed breaks, and the location where execution has stopped:

```
DB/display_debugging_environment do=(b,ua)
```

Write defaults, breaks, STEP_MODE attributes, and location where execution has stopped to file FILE1 and returns the status to variable SS:

```
DB/disde do=all output=file1 status=ss
```

Write defaults, breaks, STEP_MODE attributes, and location where execution has stopped to the current default Debug output file:

```
DB/disde
```

DISPLAY_MEMORY

The DISPLAY_MEMORY command allows you to display information located at any address to which you have read access. This command allows you to debug your program even when compiler-generated symbol tables are not available, and to display memory areas that do not correspond to program identifiers. Each display line shows the memory contents in hexadecimal and ASCII formats; the relative byte offset from the initial address is also shown.

The compiler-generated attributes list shows the section name and offset for all variables. You can reference static variables by specifying section name and byte offset. You can reference variables on the stack by specifying the machine address of the stack frame and byte offset. You can obtain the address of the stack frame of the procedure executing when Debug got control by displaying register A1 (refer to the DISPLAY_REGISTER command description in this chapter). You can obtain the address of other stack frames by displaying the save area of the wanted stack frame using the DISPLAY_STACK_FRAME command and obtaining the value of register A1 from that stack frame. You can also use the DISPLAY_PROGRAM_VALUE command to display program variables when symbol tables are available.

The DISPLAY_MEMORY command has the following format:

DISPLAY_MEMORY or **DISM**
  address
    address can be one or more of the following:
      SECTION = name or keyword value
      MODULE = name
      ADDRESS = rsssooooooo(16)
  BYTE_OFFSET = positive integer
  BYTE_COUNT = positive integer > 0
  REPEAT_COUNT = positive integer > 0 or ALL
  OUTPUT = file reference
  STATUS = status variable

Parameters

The parameters of the DISM command have the following options.

address

Specifies the memory location to be displayed. The memory location is specified by one or more of the following address parameters:

SECTION = name or keyword value
MODULE = name
ADDRESS = rsssooooooo(16)

SECTION or *SEC*

Identifies the memory section that contains the data to be displayed. Options are:

Omitted

Causes the memory address specified by the ADDRESS parameter to be displayed.

SECTION = $BLANK

Causes the memory section that contains unnamed common to be displayed.

SECTION = $BINDING

Causes the memory section that contains the links to external procedures and the data of the module to be displayed.

SECTION = $LITERAL

Causes the memory section that containing the literal data (for example, long constants) of the module to be displayed.

SECTION = $STATIC

Causes the memory section containing the static (not on the run-time stack) variables not explicitly allocated to a named section of the module to be displayed.

When you use SECTION to specify an address, you must qualify it with the MODULE parameter. You can use the BYTE_OFFSET parameter to modify the starting address of memory to be displayed.

MODULE or M

Specifies the module that contains the data to be displayed. The MODULE parameter cannot be specified unless the SECTION parameter is specified. Options are:

Omitted

Causes the memory address specified by the ADDRESS parameter to be displayed.

MODULE = name

Causes the specified name as the module with the data to be displayed.

The term module refers to a program, subprogram (subroutine or function), subprogram name, or block data. The meaning of the term module is different for COBOL and CYBIL. Module is defined for COBOL and CYBIL under Addressing.

ADDRESS or A

Specifies the address of the first byte of memory to be displayed. Options are:

Omitted

Causes the address specified by the SECTION and MODULE parameters to be used.

ADDRESS = rsssooooooooo(16)

where r is the ring number, sss is the segment number, and oooooooo is the offset from the beginning of the segment. You can use the BYTE_OFFSET parameter to modify the starting address of memory to be displayed.

BYTE_OFFSET or *BO*

Specifies the offset to the base address established by one of the address parameters. Options are:

Omitted

Causes zero to be used.

BYTE_OFFSET = integer

Causes the integer to be added to the base address to form a new address.

The address generated by adding BYTE_OFFSET to the base address must be within the memory block implied by the base address. The block size is the length of the section when the SECTION parameter is specified, and the length of the segment containing the machine address when the ADDRESS parameter is specified.

BYTE_COUNT or *BC*

Specifies the number of bytes in the item to be displayed. Options are:

Omitted

Causes one byte to be displayed.

BYTE_COUNT = positive integer > 0

Causes the specified number of bytes to be displayed.

REPEAT_COUNT or *RC*

Specifies the number of memory area (items) of length BYTE_COUNT to be displayed. Options are:

Omitted

Causes one item to be displayed.

REPEAT_COUNT = positive integer > 0

Causes the specified number of items to be displayed. The maximum amount of memory that can be displayed is limited to the block size implied by address (section length for SECTION and segment length for ADDRESS). A large integer causes all memory from the specified address to the end of the memory block to be displayed.

REPEAT_COUNT = ALL

Causes all items to be displayed. The keyword ALL causes all memory from the specified address to the end of the memory block to be displayed.

OUTPUT or *O*

Specifies the file on which the displayed information is written. Options are:

Omitted

Causes the current Debug output file to be written to.

OUTPUT = file reference

Causes the specified file to be written to.

STATUS

Specifies the return status of the command. Options are:

Omitted

Causes Debug to process the next command if an error does not occur. Causes the return status to be output to $RESPONSE (and to the DEBUG_OUTPUT file if $RESPONSE is connected to that file) if an error does occur. $RESPONSE is normally connected during interactive debugging.

STATUS = status variable

Causes the specified variable to receive the return status.

Examples

Display the first three bytes of the literal memory section for module MOD1:

```
DB/display_memory section=$literal module=mod1 byte_count=3
```

Display the first four bytes of the memory section DATA1 for module MOD2 as separate items:

```
DB/display_memory sec=data1 module=mod2 rc=4
```

Display the first 25 bytes of memory starting from the specified address:

```
DB/dism a=0b02400000224(16) bo=8 rc=25
```

DISPLAY_PROGRAM_VALUE

Displays program values (except boolean values) as they appear in the source program, not in hexadecimal format.

The DISPLAY_PROGRAM_VALUE command has the following format:

DISPLAY_PROGRAM_VALUE or **DISPV**
NAME = name
MODULE = name
PROCEDURE = name
RECURSION_LEVEL = positive integer > 0
RECURSION_DIRECTION = keyword value
OUTPUT = file reference
STATUS = status variable

Parameters

The parameters of the DISPV command have the following options.

NAME or **N**

Specifies the name of the program variable in the source program whose value is to be changed. The form is:

NAME = variable

Variable can be any variable defined in your program. You can change simple variables, arrays, array elements, named constants, labels, subprogram names, and substrings. Subscripts and substrings references can be variables of type BOOLEAN, COMPLEX, DOUBLE PRECISION, INTEGER, or REAL or constants of type COMPLEX, DOUBLE PRECISION, INTEGER, or REAL. Subscripts or substring references that are constants of type COMPLEX, DOUBLE PRECISION, or REAL must be specified as SCL string variables as discussed below. Subscripts or substring references cannot contain expressions.

The variable must be used in your program.

SCL string variables can be used to name long program names. To do this, assign the SCL variable to a string containing the identifier. Then use the SCL variable preceded by a question mark as the value of the NAME parameter.

MODULE or *M*

Specifies the name of the module that contains the NAME parameter variable.

Omitted

Causes the default module (the module executing when Debug gained control or the module specified by the CHANGE_DEFAULTS command) as the name of the nodule.

MODULE = name

Causes the specified name to be used as the name of the module.

The terms module and procedure mean the same thing in FORTRAN. Both terms refer to a program, subprogram (subroutine or function), subprogram name, or block data. Because the MODULE and PROCEDURE parameters name the same program unit, either both parameters should be assigned the same name or both parameters should not be specified.

The terms mean different things in COBOL and CYBIL. Module and procedure are defined for COBOL and CYBIL under Addressing.

PROCEDURE or *P*

Specifies the name of the procedure that contains the NAME parameter variable.

Omitted

Causes the default procedure (the procedure executing when Debug gained control or the procedure specified by the CHANGE_DEFAULTS command) to be used as the name of the procedure.

PROCEDURE = name

Causes the specified name to be used as the name of the procedure.

The terms module and procedure mean the same thing in FORTRAN. Both terms refer to a program, subprogram (subroutine or function), subprogram name, or block data. Because the MODULE and PROCEDURE parameters name the same program unit, either both parameters should be assigned the same name or both parameters should not be specified.

The terms mean different things in COBOL and CYBIL. Module and procedure are defined for COBOL and CYBIL under Addressing.

RECURSION_LEVEL or *RL*

Specifies which invocation of a named recursive procedure is to be used. Options are:

Omitted

Causes a value of 1 to be used.

RECURSION_LEVEL = positive integer > 0

Causes the specified invocation to be used. If RECURSION_DIRECTION=FORWARD, then 1 is the first invocation, 2 is the one it invoked, and so on. If RECURSION_DIRECTION=BACKWARD, then 1 is the most recent invocation, 2 is the predecessor, and so on.

Recursion only applies to program variables stored on the stack. Recursion cannot not apply to variables stored in either a common block or the $STATIC section.

RECURSION_DIRECTION or *RD*

Controls how the value of the RECURSION_LEVEL parameter is interpreted. Options are:

Omitted

Causes BACKWARD to be used.

RECURSION_DIRECTION = FORWARD

Causes 1 to be used as the first invocation, 2 as the one it invoked, and so on.

RECURSION_DIRECTION = BACKWARD

Causes 1 to be used as the most recent invocation, 2 as its predecessor, and so on.

Recursion only applies to program variables stored on the stack. Recursion cannot not apply to variables stored in either a common block or the $STATIC section.

OUTPUT or *O*

Specifies the file where the display information is written.

Omitted

Causes the current Debug output file to be written to.

OUTPUT = file reference

Causes the specified file to be written to.

STATUS

Specifies the return status of the command. Options are:

Omitted

Causes Debug to process the next command if an error does not occur. Causes the return status to be output to $RESPONSE (and to the DEBUG_OUTPUT file if $RESPONSE is connected to that file) if an error does occur. $RESPONSE is normally connected during interactive debugging.

STATUS = status variable

Causes the specified variable to receive the return status.

Examples

The examples refer to the following definitions:

```
COMMON /BLK/ DVAL, RVAL, IVAL, ZVAL
DATA DVAL, RVAL, IVAL, ZVAL/20.0D+0, 3.45E+01, 30, (+20.0,20.3)/
```

Display the value of DVAL:

```
DB/display_program_value name=dval
dval = 20.
DB/
```

Display the value of RVAL:

```
DB/dispv name=rval
rval = 34.5
DB/
```

Display the value of IVAL:

```
DB/display_program_value ival
ival = 30
DB/
```

Display the value of ZVAL:

```
DB/dispv name=zval
zval = (20., 20.3)
DB/
```

DISPLAY_REGISTER

Displays the contents of the P, A, or X registers that are associated with the procedure executing when Debug gained control.

The DISPLAY_REGISTER command has the following format:

DISPLAY_REGISTER or **DISPLAY_REGISTERS** or **DISR**
KIND = list of keyword values
NUMBER = integer or list of integers or ALL
TYPE = keyword value
OUTPUT = file reference
STATUS = status variable

Parameters

KIND or *K*

Specifies the register to change. Options are:

Omitted

Specifies the P register to change.

KIND = P

Specifies the P register to change.

KIND = A

Specifies the A registers to change.

KIND = X

Specifies the X registers to change.

NUMBER or *N*

Indicates which A or X registers specified by the KIND parameter will change. If the KIND parameter is not specified, the P register is assumed and this parameter is ignored.

The A and X registers can be saved and changed when Debug gains control. However, some registers are not always saved; a message is issued for each register that cannot be changed because it was not saved.

Options are:

Omitted

Specifies the zero register to change.

NUMBER = integer or range of integers

Specifies a set of registers to change. Integer can be 0 through 15.

NUMBER = ALL

Specifies that all A or X registers will change.

TYPE or *T*

Specifies the type of data specified by the VALUE parameter. Options are:

Omitted

Specifies HEX for string values and INTEGER for numeric values.

TYPE = ASCII or A

Specifies ASCII strings.

TYPE = HEX or H

Specifies hexadecimal strings.

TYPE = INTEGER or I

Specifies integer strings.

OUTPUT or *O*

Specifies the file where the display information is written.

Omitted

Causes the current Debug output file to be written to.

OUTPUT = file reference

Causes the specified file to be written to.

STATUS

Specifies the return status of the command. Options are:

Omitted

Causes Debug to process the next command if an error does not occur. Causes the return status to be output to $RESPONSE (and to the DEBUG_OUTPUT file if $RESPONSE is connected to that file) if an error does occur. $RESPONSE is normally connected during interactive debugging.

STATUS = status variable

Causes the specified variable to receive the return status.

H3 Examples

Display the contents of the P register in hexadecimal:

```
DB/display_register p

Debug outputs P=B 031 00000040
```

Display the contents of the A8 register in hexadecimal:

```
DB/display_register kind=a number=8 type=hex
```

Display the contents of the X4, X5, X6, X7, X8, X9, and X10 registers in hexadecimal:

```
DB/disr kind=x number=4..10
```

DISPLAY_STACK_FRAME

Displays the contents of one or more stack frames. Values are displayed in hexadecimal.

The DISPLAY_STACK_FRAME command has the following format:

DISPLAY_STACK_FRAME or **DISPLAY_STACK_FRAMES** or **DISSF**
COUNT = integer or ALL
START = positive integer > 0
DISPLAY_OPTIONS = list of keyword values
OUTPUT = file reference
STATUS = status variable

Parameters

The parameters for the DISPLAY_STACK_FRAME command have the following options.

COUNT or *C*

Specifies the number of calls to be displayed. Options are:

Omitted

Causes all calls to be displayed.

COUNT = positive integer > 0

Causes the specified number of calls to be displayed; if the integer is greater than the number of existing calls, all calls are displayed.

COUNT = ALL

Causes all calls to be displayed.

START or *S*

Specifies the call on the chain to be displayed first. Thus, it is possible to skip the most recent calls. Options are:

Omitted

Causes the most recent call to be displayed.

START = positive integer > 0

Causes the specified call to be displayed. One represents the most recent call, two represents the predecessor of the most recent call, and so forth.

An informative message is issued if the specified number of calls is greater than the actual number of calls.

DISPLAY_OPTIONS or *DO*

Specifies the area of the stack frames to be displayed. Options are:

Omitted

Specifies both the automatic and save areas.

DISPLAY_OPTIONS = AUTO or A

Specifies the area that contains the automatic (dynamically allocated) variables of the procedure.

DISPLAY_OPTIONS = SAVE or S

Specifies the area that contains a copy of the registers of the procedure as they existed at the time of a call or trap.

DISPLAY_OPTIONS = ALL

Specifies both the automatic and save areas.

OUTPUT or *O*

Specifies the file on which the call information is written. Options are:

Omitted

Causes the current DEBUG_OUTPUT file to be written to.

OUTPUT = file reference

Causes the specified file to be written to.

STATUS

Specifies the return status of the command. Options are:

Omitted

Causes Debug to process the next command if an error does not occur. Causes the return status to be output to $RESPONSE (and to the DEBUG_OUTPUT file if $RESPONSE is connected to that file) if an error does occur. $RESPONSE is normally connected during interactive debugging.

STATUS = status variable

Causes the specified variable to receive the return status.

Examples

Display the save area of the most recent stack frame:

```
DB/display_stack_frame display_option=save
```

Display the automatic and save areas of three stack frames beginning with the second most recent one:

```
DB/display_stack_frames count=3 start=2
```

Display the automatic and save areas of the most recent stack frame:

```
DB/dissf  count=1

STACK FRAME 001         SEGMENT=032
00000000    00000000  00000000
00000008    00000000  00000000
00000010    30300000  00C0FFFF    00
00000018    80000000  00000000
00000020    B032B031  00000000     2 1
00000028    0000B01D  0009B346           F
00000030    0000B032  00000430       2   0
00000038    0040B032  00000400      @ 2
00000040    FF77B032  000003C0      w 2
00000048    FFFCB01B  00020F78           x
00000050    0000B032  00000390       2

SAVE AREA

P=B 035 00000026         VMID=0
UM=FFF7    UCR=0040    MCR=0000
```

```
=B 032 00000460      A1=B 032 00000408
A2=B 032 000003C0      A3=B 030 00000000
A4=B 032 00000390      A5=B 02F 00000020
A6=B 02E 00000000      A7=B 02F 00000000
A8=B 00F 00000018      A9=B 032 00000630
AA=B 032 00000A30      AB=F FFF 80000000
AC=F FFF 80000000      AD=B 032 00001058
AE=F FFF 80000000      AF=B 00B 000557F8

X0=0000B01D 00020060      X1=00000000 00000000
X2=0000FFFF 80000000      X3=000007FF FFFFFFFF

X4=00000000 10000000      X5=00000000 00000008
X6=00000000 0000000D      X7=00000000 0000001D
X8=00000000 00000000      X9=00000000 00000008
XA=00000000 00000300      XB=00000000 00000000
XC=00000000 00000001      XD=00000000 00000022
XE=00000000 00010040      XF=00000000 0000004E
```

QUIT

Terminates the Debug session and returns control to the NOS/VE operating system. The session is terminated immediately; the program is not executed to completion.

The Quit command has the following format:

QUIT or **QUI**
STATUS = *status variable*

Parameter

The parameter for the QUIT command has the following options.

STATUS

Specifies the return status of the command. Options are:

Omitted

Causes Debug to process the next command if an error does not occur. Causes the return status to be output to $RESPONSE (and to the DEBUG_OUTPUT file if $RESPONSE is connected to that file) if an error does occur. $RESPONSE is normally connected during interactive debugging.

STATUS = status variable

Causes the specified variable to receive the return status.

RUN

Initiates or resumes program execution once Debug has gained control. Execution continues until Debug again gains control. If the program has run to completion, entering the RUN command terminates the program and returns control to the NOS/VE operating system.

Execution begins at the instruction whose address is contained in the P register of the program for the event that caused Debug to gain control. If the P register points to the instruction that caused the event (such as division by zero), the same event will occur immediately after entering the RUN command. In this case, you must change the value in the P register with the CHANGE_REGISTER command or change the value of one of the operands with the CHANGE_PROGRAM_VALUE command before entering the RUN command.

When Debug processes the RUN command, all previousely created SCL blocks (except SET_BREAK command information and the name of the current DEBUG_INPUT file) are lost. This means that all information about SCL commands, such as IF-THEN blocks or WHILE-FOR loops that span RUN commands are lost. Global variables must be recreated with XREF. For example, the following sequence recreates the variable COUNT:

```
DB/crev count kind=integer scope=job v=0
DB/setb b=one l=one ..
DB../run "BREAK ONE"
DB/crev count kind=integer scope=xref
```

The RUN command has the following format:

RUN
STATUS = status variable

Parameter

The parameter for the RUN command has the following options.

STATUS

Specifies the return status of the command. Options are:

Omitted

Causes Debug to process the next command if an error does not occur. Causes the return status to be output to $RESPONSE (and to the DEBUG_OUTPUT file if $RESPONSE is connected to that file) if an error does occur. $RESPONSE is normally connected during interactive debugging.

STATUS = status variable

Causes the specified variable to receive the return status.

SET_BREAK

Defines a break. You can specify one or more events and the location at which Debug is to take control. When the specified event occurs, program execution is suspended and a message informs you which break occurred. At this point, you can enter another Debug command that can be processed by the operating system command or other active command utility.

The SET_BREAK command has the following format:

SET_BREAK or **SET_BREAKS** or **SETB**
BREAK = name
EVENT = list of keyword values
address
address can be one or more of the following:
LINE = integer
SECTION = name or keyword value
MODULE = name
PROCEDURE = name
ENTRY_POINT = name
ADDRESS = rsssooooooo(16)
BYTE_OFFSET = integer
BYTE_COUNT = integer
COMMAND = string
STATUS = status variable

Parameters

The parameters for the SET_BREAK command have the following options.

BREAK or **B**

Specifies the name of the break. This name is used to reference the break definition in the DISPLAY_BREAK and DELETE_BREAK commands. This name is displayed in the break report message when the break occurs. A break cannot be named ALL. The form is:

BREAK = name

The break is called by the specified name.

EVENT or *EVENTS* or *E*

Specifies the events that must occur for the break to occur. If you specify more than one event, the break is honored if any of the events occur. Event options are:

Omitted

Causes EXECUTION to be used.

EVENT = one or more of the following events:

ARITHMETIC_OVERFLOW or AO

Breaks when an arithmetic overflow occurs on an instruction in the specified address range. The P register points to the instruction that caused the overflow.

ARITHMETIC_SIGNIFICANCE or AS

Breaks when arithmetic significance is lost on an instruction in the specified address range. The P register points to the instruction that caused the loss of significance.

BRANCH or B

Breaks before either a branch to or a return from any location in the specified address range occurs.

CALL or C

Breaks before a subprogram call occurs to any address in the specified address range.

DIVIDE_FAULT or DF

Breaks when division by zero occurs in an instruction in the specified address range. The P register points to the instruction that caused the division by zero.

EXECUTION or E

Breaks before the instruction in the specified address range is executed.

If the address is specified by the line number, not every line is usable. For example, breaks cannot be set at ENDIF statements because it is not obvious when control reaches them.

EXPONENT_OVERFLOW or EO

Breaks when an exponent overflow occurs in an instruction in the specified address range. The P register points to the instruction following the one that caused the overflow.

EXPONENT_UNDERFLOW or EU

Breaks when an exponent underflow occurs in an instruction in the specified address range. The P register points to the instruction following the one that caused the underflow.

FLOATING_POINT_INDEFINITE or FPI

Breaks when the result of a floating-point operation is indefinite in an instruction in the specified address range. The P register points to the instruction following the one that caused the results to be indefinite.

FLOATING_POINT_SIGNIFICANCE or FPS

Breaks when significance is lost during a floating-point operation in an instruction in the specified address range. The P register points to the instruction following the one that caused the loss of significance. This event will not occur unless your program sets the floating-point loss-of-significance bit in the user mask register.

INVALID_BDP_DATA or IBD

Breaks when a business data processing (BDP) instruction fault occurs in an instruction in the specified address range. The P register points to the instruction that caused the fault. The BDP instructions are described in the Virtual State Hardware reference manual.

READ or R

Breaks before a read occurs from the specified address range. The break occurs only if the first byte of the item to be read is within the address range.

READ_NEXT_INSTRUCTION or RNI

Breaks before the instruction in the specified address range is executed.

WRITE or W

Breaks before a write occurs into the specified address range. The break occurs only if the first byte of the item to be written is within the address range.

Debug gains control when the following events occur, even if you do not set a break for them:

ARITHMETIC_OVERFLOW
ARITHMETIC_SIGNIFICANCE
DIVIDE_FAULT
EXPONENT_OVERFLOW
EXPONENT_UNDERFLOW
FLOATING_POINT_INDEFINITE
FLOATING_POINT_SIGNIFICANCE
INVALID_BDP_DATA

Specific breaks can be set for these events, however, so that a predefined set of commands can be executed when Debug gains control.

address

Specifies the location at which the break occurs. For the break to occur, the specified event must occur within the range defined by the address parameters. All address parameters are interpreted as a single address. You can use the BYTE_COUNT and BYTE_OFFSET parameters to specify an address range. Options are:

Omitted

Indicates an address range of one byte.

One or more of the following parameters:

LINE = integer
SECTION = name or keyword value
MODULE = name
PROCEDURE = name
ENTRY_POINT = name
ADDRESS = rsssoooooooo(16)

LINE or *L*

Line at which Debug gains control. The line number must exist in the current default module (that was explicitly set with the CHANGE_DEFAULTS command or was executing when Debug gained control) unless the MODULE parameter is also specified.

Omitted

Indicates that the break address is specified by another parameter.

LINE = integer

Indicates the line number on which the break occurs in the specified address range.

You can use BYTE_OFFSET and BYTE_COUNT to modify this parameter.

Not all lines of a program can be referenced. Only executable statements that begin on a separate line can be referenced. A second or third statement on a line or a line that contains the continuation of a statement cannot be referenced. ENDIF lines cannot be referenced.

If the source code was compiled at a high optimization level, you will not be able to reference some lines because they have been moved or deleted.

SECTION or *SEC*

Identifies a memory section. Options are:

Omitted

Indicates that the break address is specified by another parameter.

SECTION = $BINDING

Identifies the memory section that contains the links to external procedures and the data of the module.

SECTION = $BLANK

Identifies the memory section that contains unnamed common.

SECTION = $LITERAL

Identifies the memory section that containing the literal data (for example, long constants) of the module.

SECTION = $STATIC

Identifies the memory section containing the static (not on the run-time stack) variables not explicitly allocated to a named section of the module.

You can use the BYTE_OFFSET and BYTE_COUNT parameters to modify this parameter.

The section must exist for the current default module (that was explicitly set with the CHANGE_DEFAULTS command or was executing when Debug gained control) unless the MODULE parameter is also specified.

The SECTION parameter cannot be specified for modules that are components of a bound module unless the section is a common block (read the discussion about Addressing Bound Modules in this chapter).

MODULE or *M*

Specifies an address or qualification of another address parameter. If used alone, the MODULE parameter specifies an address (the first byte of the first code section of the module). Options are:

Omitted

Causes the current default module (that was explicitly set with the CHANGE_DEFAULTS command or was executing when Debug gained control) to be used as the address.

MODULE = name

Causes the specified name to be used as the address.

If used to specify an address, the BYTE_OFFSET and BYTE_COUNT parameters can be used to modify this parameter.

If MODULE is used with the LINE, SECTION, or PROCEDURE address parameters, the MODULE parameter identifies the module containing the line, section, or procedure.

The term module refers to a program, subprogram (subroutine or function), subprogram name, or block data. Because the MODULE and PROCEDURE parameters name the same program unit, either both parameters should be assigned the same name or both parameters should not be specified.

The terms mean different things in COBOL and CYBIL. Module and procedure are defined for COBOL and CYBIL under Addressing.

PROCEDURE or *P*

Specifies an address (the first byte of the code section of the procedure). The procedure must exist in the current default module (that was explicitly set with the CHANGE_DEFAULTS command or was executing when Debug gained control) unless the MODULE parameter is specified.

You cannot specify the LINE or SECTION address parameters with PROCEDURE.

Omitted

Indicates that the break address is specified by another parameter.

PROCEDURE = name

Causes the specified name to be used as the break address.

You can use the BYTE_OFFSET and BYTE_COUNT parameters to modify this parameter.

The term module refers to a program, subprogram (subroutine or function), subprogram name, or block data. Because the MODULE and PROCEDURE parameters name the same program unit, either both parameters should be assigned the same name or both parameters should not be specified.

The terms mean different things in COBOL and CYBIL. Module and procedure are defined for COBOL and CYBIL under Addressing.

ENTRY_POINT or *EP*

Specifies an entry point expressed as a name known to the loader; cannot be an address range. ENTRY_POINT must be a program or subprogram name with an XDCL attribute subject to certain restrictions. See the Object Code Management manual for information on restrictions.

Omitted

Indicates that the break address is specified by another parameter.

ENTRY_POINT = name

Causes the specified name to be used as the entry point name.

You can use the BYTE_OFFSET and BYTE_COUNT parameters to modify the ENTRY_POINT parameter. You cannot use other address parameters with this parameter.

ADDRESS or *A*

Specifies the address of the first byte of memory to be changed. The form is:

ADDRESS = rsssooooooooo(16)

where r is the ring number, sss is the segment number, and ooooooo is the offset from the beginning of the segment. You can obtain machine addresses by using the cross-reference and load maps for your program.

BYTE_OFFSET or *BO*

Specifies the offset to the base address established by one of the address parameters. Options are:

Omitted

Causes zero to be used.

BYTE_OFFSET = positive integer

Causes the specified integer to be added to the base address to form a new address. The break is then set for this new address.

The address generated by adding BYTE_OFFSET to the base address must be within the memory block implied by the base address. The block size is the length of the section when the SECTION parameter is specified, and the length of the segment containing the machine address when the ADDRESS parameter is specified.

BYTE_COUNT or *BC*

Specifies the number of bytes in the item to be displayed. Options are:

Omitted

Causes one byte in the address range to be displayed.

BYTE_COUNT = positive integer > 0

Causes the specified number of bytes in the address range to be displayed.

COMMAND or *COMMANDS* or *C*

Specifies the string of commands to be executed by Debug, SCL, or any other active command processor when the break is honored. Commands are read from the Debug file. Options are:

Omitted

Indicates that no commands are associated with the break.

COMMAND = string

Specifies a string of commands to be used.

If a command in the string includes a quoted string, that string must be enclosed in two single quotes. After the commands in the string have been executed, commands are read from the current Debug input file unless the string contains a RUN command.

No break report message is issued before the commands in the string are executed. If you want a message to be displayed, include an SCL DISPLAY_VALUE command in the string. If an error is detected in one of the commands in the string, the break report message is issued, the error is reported, and commands are read from the Debug input file. The remaining commands in the string are not executed.

STATUS

Specifies the return status of the command. Options are:

Omitted

Causes Debug to process the next command if an error does not occur. Causes the return status to be output to $RESPONSE (and to the DEBUG_OUTPUT file if $RESPONSE is connected to that file) if an error does occur. $RESPONSE is normally connected during interactive debugging.

STATUS = status variable

Causes the specified variable to receive the return status.

Examples

Cause a break to occur when execution reaches line 10 of module PROG1:

```
DB/set_break break=b1 line=10 module=prog1
```

Cause a break when a branch or return to line 40 of the module executing when Debug gained control occurs:

```
DB/set_break break=b2 event=branch line=40
```

SET_STEP_MODE

The SET_STEP_MODE command enables you to execute a specified subset of a task and receive control.

If step mode is activated, a RUN command causes your program to execute for the specified unit. You are then prompted for further command input. A string of commands can be associated with the step and will be processed each time the step is completed. Stepping with a unit of line or procedure is only available if the source program was compiled with the stylized code option (OPT=DEBUG).

Activating step mode is an effective debugging aid, but is very expensive in terms of execution time.

The SET_STEP_MODE command has the following format:

SET_STEP_MODE or **SETSM**
MODE = keyword value
UNIT = keyword value
MODULE = keyword value
PROCEDURE = keyword value
COMMAND = string
STATUS = status variable

Parameters

The parameters of the SETSM command have the following options.

MODE

Specifies whether to activate or deactivate step mode. Options are:

MODE = ON

Causes step mode to be activated. When step mode is on, a RUN command causes one step to be executed. A step is defined by the UNIT parameter.

MODE = OFF

Causes step mode to be deactivated. When step mode is off, any remaining parameters are ignored.

If you specify MODE=ON and step mode is already on, all previous values will be replaced with the new values.

UNIT or *U*

Specifies the length of the step. Options are:

Omitted

Causes LINE to be used.

UNIT = LINE or L

Causes the step to be reported before the code is executed for each line, except for the procedure lines.

UNIT = PROCEDURE or P

Causes the step to be reported each time a new procedure begins and after any prolog code for the procedure has executed.

MODULE or *M*

Used with the UNIT parameter, specifies the modules reported. Options are:

Omitted

Causes the current default module to be used.

MODULE = $ALL

Causes a step to be reported that is in any module.

MODULE = $CURRENT

Causes a step to be reported only if the step occurs in the module where the program is executing when step mode is activated.

You cannot specify both the MODULE and PROCEDURE parameters in the same SET_STEP_MODE command.

The terms module and procedure mean the same thing in FORTRAN. Both terms refer to a program, subprogram (subroutine or function), subprogram name, or block data. The terms mean different things in COBOL and CYBIL. Module and procedure are defined for COBOL and CYBIL under Addressing.

PROCEDURE or *P*

Used with the UNIT parameter, specifies the procedure reported. Options are:

Omitted

Causes the current default procedure to be used.

PROCEDURE = $ALL

Causes a step to be reported that is in any procedure.

PROCEDURE = $CURRENT

Causes a step to be reported only if the step occurs in the procedure where the program is executing when step mode is activated.

You cannot specify both the MODULE and PROCEDURE parameters in the same SET_STEP_MODE command.

The terms module and procedure mean the same thing in FORTRAN. Both terms refer to a program, subprogram (subroutine or function), subprogram name, or block data. The terms mean different things in COBOL and CYBIL. Module and procedure are defined for COBOL and CYBIL under Addressing.

COMMAND or *COMMANDS* or *C*

Specifies a string of commands that will be executed when the step occurs. If the string of commands includes a RUN command, the task is resumed and the step is not reported. If the string does not include a RUN command, command input will be requested from the current DEBUG_INPUT file after the string of commands has been executed.

STATUS

Specifies the return status of the command. Options are:

Omitted

Causes Debug to process the next command if an error does not occur. Causes the return status to be output to $RESPONSE (and to the DEBUG_OUTPUT file if $RESPONSE is connected to that file) if an error does occur. $RESPONSE is normally connected during interactive debugging.

STATUS = status variable

Causes the specified variable to receive the return status.

Examples

The following commands activates step mode with a unit of line in the current module, executes the entire program and displays each line that is executed, then deactivates step mode:

```
DB/setsm, on, c='disde, us; run'
DB/run
DB/setsm, off
DB/quit
```

Sample Debug Session

Debug can be used in interactive or batch mode. There is virtually no difference between an interactive and a batch Debug session. All commands and features are available in both modes. NOS/VE makes some distinction with respect to device assignment for files such as OUTPUT and initial connections for standard files such as $RESPONSE.

The following session simulates an interactive session. In an interactive session, you can either enter Debug commands one by one or specify a stream of commands that will interact with your compiled source program.

To enter Debug commands one by one, compile your program for use with Debug and then begin execution of the source code under Debug control. For example,

```
/fortran i=test l=list b=lgo
/execute_task f=lgo debug_mode=on
```

Debug gains control immediately and outputs the following prompt:

```
DEBUG 1.1  84059
DB/
```

Now you interact with Debug by entering Debug commands one by one.

To enter a stream of Debug commands that will interact with your compiled source program, you compile your program for use with Debug and then begin execution of the source code under Debug control with the following sequence:

```
/fortran i=test l=list b=lgo
/execute_task f=lgo debug_mode=on debug_input=stream
```

Debug interacts with the file of commands named in the DEBUG_INPUT parameter and writes the output to your terminal.

The remaining pages in this chapter show the result of a Debug session using the compiled source program shown in figure 11-1 and the command stream shown in figure 11-2. Figure 11-2 also shows the Debug output from each command in the command stream.

```
 1       program test
 2       implicit double precision (d)
 3       implicit complex (z)
 4       logical log1, log2, log3
 5       character*10 ch1, ch2*10, ch3*10
 6       integer aray(20)
 7       real zero
 8       boolean pind
 9       common /blk/ dval, rval, ival, zval
10 c
11 c INITIALIZATION
12 c
13       data dval, rval, ival, zval /20.0d+0,3.45e+01,30,(+20.0,20.3)/
14       data base, xpwr /2.0, 4094.0/
15       data ch1 /'jackswife'/
16       pind = shift (z"7000", 48)
17       zero = 0.0
18 c
19 c THE FOLLOWING WILL PRODUCE EVENTS - EO, EU, FPI, DF
20 c
21 c EXPONENT OVERFLOW (EO)
22 c
23       reslt = 2.0 ** xpwr / (2.0 ** (-2))
24       print *,' EXPONENT OVERFLOW RESULT = ', reslt
25 c
26 c EXPONENT UNDERFLOW (EU)
27 c
28       PRINT *,' EXPONENT UNDERFLOW FOLLOWS '
29       reslt = (base**(-3000)) * ( base**(-1098))
30       print *,' EXPONENT UNDERFLOW RESULT = ', reslt
31 c
32 c FLOATING POINT INDEFINITE (FPI)
33 c
34       arg = ranf( ) * 2.0
35       print *,' FLOATING POINT INDEFINTE FOLLOWS '
36       reslt = pind / arg
37       print *,' FPI RESULT = ', reslt
38 c
39 c DIVIDE FAULT
40 c USE DISPLAY_REGISTER AND CHANGE_REGISTER TO GET AROUND DF.
41 c
42       print *,' DIVIDE FAULT FOLLOWS '
43       reslt = arg / zero
44       print *,' DIVIDE FAULT RESULT = ', reslt
45 c
46 c 2ND DIVIDE FAULT
47 c USE DISPLAY_PROGRAM_VALUE AND CHANGE_PROGRAM_VALUE TO FIX DF.
48       print *,' 2ND DIVIDE FAULT FOLLOWS '
49       reslt = arg / zero
50 c
51 c MAKE CALLS TO SUBROUTINE I01
52 c
53       x1 = y1 = 3.0
```

(Continued on next page)

Figure 11-1. Compiled Source Program used in Debug Session

(Continued from previous page)

```
      call io1(2.0, x1, y1, log1)
      print *,' RETURN FROM SUBROUTINE IO1 '
      if (log1) print*,' Y1 IS GREATER THAN X1 '
c
c MAKE CALLS TO SECOND-LEVEL SUBROUTINE
c
      call sub1(log1)
      print *,' log1 = ', log1
      log2 = .false.
c
c FORM LOGICAL EXPRESSION
c
      log3 = log1 .and. log2
      print *,' LOG3 = ', log3
c
c FILL INTEGER ARRAY (ARAY) WITH VALUES
c
      do 10 i = 1,20
      aray(i) = ranf( ) * float(i)
10    continue
      print *,' ARAY(1-20) = ', ARAY
      stop
      end
```

```
c
c SUBROUTINE IO1
c
      subroutine io1(rcon, x1, y1, logc)
      logical logc
      real rcon, xval, yval
      common /blk/ dval, rval, ival, zval
c
      print *,' HERE WE ARE IN IO1 '
      do 10 i = 1,5
      rval = float(i) + float(i)
      dval = float(i) * dval
      zval = cmplx(real(dval), float(i))
      ival = i
10    continue
      print *,' rval = ', rval
      print *,' dval = ', dval
      print *,' ival = ', ival
      print *,' zval = ', zval
      y1 = rval + dval
      x1 = ival + zval
      if (y1 .gt. x1) then
      logc = .true.
      else
      logc = .false.
      endif
      print *,' * RETURN TO MAIN PROGRAM '
      return
      end
```

(Continued on next page)

Figure 11-1. Compiled Source Program used in Debug Session

(Continued from previous page)

```
 1 c
 2 c SUBROUTINE SUB1
 3 c
 4       subroutine sub1(log)
 5       logical log
 6       common /blk/ dval, rval, ival, zval
 7 c
 8       print *,' HERE WE ARE IN SUB1 '
 9       reslt= dval + rval + ival + zval
10       print *,' RESULT = ', reslt
11       if (reslt .gt. 100.) then
12       log = .true.
13       print *,' RESULT IS GREATER THAN 100.0 '
14       else
15       log = .false.
16       print *,' RESULT IS LESS THAN 100.0 '
17       endif
18       print *,' CALL TO SUB2 FOLLOWS '
19       call sub2(log, reslt)
20       print *,' RETURN FROM SUB2 '
21       return
22       end
```

```
 1 c
 2 c SUBROUTINE SUB2
 3 c
 4       subroutine sub2(log, r1)
 5       logical log
 6       real r1
 7       common /blk/ dval, rval, ival, zval
 8 c
 9       print *,' HERE WE ARE IN SUB2 '
10       reslt = dval + rval + ival + zval
11       r1 = r1 - reslt
12       if (r1 .eq. 0.0) then
13       print *,' FINAL RESULT = 0 '
14       else
15       print *,' FINAL RESULT = ', r1
16       endif
17       print *,' RETURN FROM SUB2 NOW '
18       return
19       end
```

Figure 11-1. Compiled Source Program used in Debug Session

DEBUG_INPUT Command Stream	Debug Output
	DEBUG 1.1 84059
setb break=b16 line=16 module=test	
run "b16"	-- DEBUG: break B16, execution at M=TEST L=16
display_calls	-- Traceback from procedure TEST module TEST at line 16 -- Called from procedure FLP$INIT module FLM$INIT byte offset 406(16) -- Called from procedure TEST module TEST at line 1 byte offset 22
display_program_value name=dval	dval = 20.
dispv rval	rval = 34.5
dispv zval	zval = (20., 20.3)
dispv ival	ival = 30
change_program_value n=dval v=+20.0d+0	
chapv name=rval value=+300.0	
"chapv n=zval v=(+33.0,+2.0)"	
chapv name=ival v=123	
dispv rval	rval = 300.
dispv dval	dval = 20.
dispv zval	zval = (20., 20.3)
dispv ival	ival = 123
set_break break=eo1 .. event=exponent_overflow .. procedure=test .. bo=50(16) bc=40 .. c=' putl '' EXPONENT_OVERFLOW HIT ''; .. disc; run'	
setb b=eu1 e=eu p=test bo=98(16) bc=58 .. c=' putl '' EXPONENT_UNDERFLOW HIT '''	
setb b=fpi1 e=fpi p=test bo=130(16) .. bc=18 c=' putl '' FPI HIT '' '	

(Continued on next page)

Figure 11-2. Command Stream Used and Output from Debug Session

— (Continued from previous page)

DEBUG_INPUT Command Stream	Debug Output

```
setb b=b49 line=49 m=test ..
c=' PUTL '' BREAK BEFORE 2ND FPI '' '

setb b=divf1 e=df p=test ..
bo=162(16) bc=18..
c=' putl '' 1ST DIVIDE FAULT HIT ''; disc'

disb break=all                          -- Break B16
                                        -- event(s) = execution
                                        -- location: M=TEST L=16

                                        -- Break E01
                                        -- event(s) = exponent_overflow
                                        -- range: M=TEST L=23 to M=TEST L=23 BO=39
                                        -- command(s) =  putl ' EXPONENT_OVERFLOW HIT ';disc;run

                                        -- Break EU1
                                        -- event(s) = eu
                                        -- range: M=TEST L=29 to M=TEST L=29 BO=57
                                        -- command(s) =  putl ' EXPONENT_UNDERFLOW HIT '

                                        -- Break FPI1

                                        -- event(s) = fpi
                                        -- range: M=TEST L=36 to M=TEST L=36 BO=17
                                        -- command(s) =  putl ' FPI HIT '

                                        -- Break B49
                                        -- event(s) = execution
                                        -- location: M=TEST L=49
                                        -- command(s) =  PUTL ' BREAK BEFORE 2ND FPI '

                                        -- Break DIVF1
                                        -- event(s) = df
                                        -- range: M=TEST L=43 to M=TEST L=43 BO=17
                                        -- command(s) =  putl ' 1ST DIVIDE FAULT HIT ';disc

delete_break break=b16

display_breaks b=all                    -- Break E01
                                        -- event(s) = exponent_overflow
                                        -- range: M=TEST L=23 to M=TEST L=23 BO=39
                                        -- command(s) =  putl ' EXPONENT_OVERFLOW HIT ';disc;run

                                        -- Break EU1
                                        -- event(s) = eu
                                        -- range: M=TEST L=29 to M=TEST L=29 BO=57
                                        -- command(s) =  putl ' EXPONENT_UNDERFLOW HIT '
```

(Continued on next page)—

Figure 11-2. Command Stream Used and Output from Debug Session

(Continued from previous page)

DEBUG_INPUT Command Stream	Debug Output
	-- Break FPI1 -- event(s) = fpi -- range: M=TEST L=36 to M=TEST L=36 BO=17 -- command(s) = putl ' FPI HIT '
	-- Break B49 -- event(s) = execution -- location: M=TEST L=49 -- command(s) = PUTL ' BREAK BEFORE 2ND FPI '
	-- Break DIVF1 -- event(s) = df -- range: M=TEST L=43 to M=TEST L=43 BO=17 -- command(s) = putl ' 1ST DIVIDE FAULT HIT ';disc
run " E01 BREAK- RUN- EU1 BREAK "	EXPONENT_OVERFLOW HIT -- Traceback from procedure TEST module TEST at line 23 byte offset 36 -- Called from procedure FLP$INIT module FLM$INIT byte -- Called from procedure TEST module TEST at line 1 byte offset 22
RUN " EXPONENT_UNDERFLOW IN OUTPUT_FLOATING_NUMBER "	EXPONENT OVERFLOW RESULT = INFINITE EXPONENT UNDERFLOW FOLLOWS EXPONENT_UNDERFLOW HIT -- DEBUG: break EU1, eu at M=TEST L=29 BO=54 -- DEBUG: exponent_underflow at M=MLM$OUTPUT_FLOATING_NUMBER BO=586(16) EXPONENT UNDERFLOW RESULT = 0.
run " FPI1 BREAK - P REGISTER POINTS TO SAME INSTRUCTION "	FLOATING POINT INDEFINTE FOLLOWS FPI HIT -- DEBUG: break FPI1, fpi at M=TEST L=36 BO=12
display_calls	-- Traceback from procedure TEST module TEST at line 36 byte offset 12 -- Called from procedure FLP$INIT module FLM$INIT byte offset 406(16) -- Called from procedure TEST module TEST at line 1 byte offset 22

(Continued on next page)

Figure 11-2. Command Stream Used and Output from Debug Session

— (Continued from previous page)

DEBUG_INPUT Command Stream	Debug Output
display_debugging_environment do=all	-- Default module is $CURRENT. -- Default procedure is $CURRENT. -- Default debug_input file is DBIN. -- Default debug_output file is $OUTPUT. -- The number of breaks set by the user is 5. -- The number of breaks in use by DEBUG is 0. -- The number of available breaks is 59. -- Step_mode is OFF. -- Execution is currently stopped at B 02E 0000013C which, in higher symbolic terms is M=TEST L=36 B0=12
disr k=x n=4	X4=70000000 00000000
char k=x n=4 v=4001800000000000(16)	
disr k=x n=4	X4=40018000 00000000
run " DIVF1 BREAK- P REGISTER POINTS TO SAME INSTRUCTION "	FPI RESULT = .8619000797934 DIVIDE FAULT FOLLOWS 1ST DIVIDE FAULT HIT -- Traceback from procedure TEST module TEST at line 43 byte offset 12 -- Called from procedure FLP$INIT module FLM$INIT byte offset 406(16) -- Called from procedure TEST module TEST at line-1 byte offset 22 -- DEBUG: break DIVF1, df at M=TEST L=43 B0=12
display_register kind=x number=4	X4=00000000 00000000
" display, change register X4, which " " contains ZERO to 1 in HEX "	
change_register kind=x .. number=4 v=4001800000000000(16)	
disr k=x n=4	X4=40018000 00000000
run " BREAK ON LINE 49 TO CHANGE VALUE OF ZERO "	DIVIDE FAULT RESULT = 1.160227297159 2ND DIVIDE FAULT FOLLOWS BREAK BEFORE 2ND FPI -- DEBUG: break B49, execution at M=TEST L=49
dispv zero	zero = 0.
chapv n=zero v=+1.0	
dispv zero	zero = 1.

(Continued on next page)—

Figure 11-2. Command Stream Used and Output from Debug Session

(Continued from previous page)

DEBUG_INPUT Command Stream	Debug Output
delb b=(eo1,eu1,fpi1,b49,divf1)	
setb b=wrval m=io1 e=write p=io1 .. section=blk bo=8 bc=8.. c=' putl '' WRITE BREAK RVAL '' '	
setb b=wival m=io1 e=w p=io1 .. section=blk bo=16 bc=8.. c=' putl '' WRITE BREAK IVAL '' '	
setb b=rrval m=io1 e=read .. p=io1 section=blk bo=8 bc=8.. c=' putl '' READ BREAK RVAL '' '	
setb b=rival m=io1 e=r p=io1 .. section=blk bo=16 bc=8.. c=' putl '' READ BREAK IVAL '' '	
disb	-- Break WRVAL -- event(s) = write -- range: section=BLK BO=8 (16) to section=BLK BO=F (16) -- command(s) = putl ' WRITE BREAK RVAL ' -- Break WIVAL -- event(s) = w -- range: section=BLK BO=10 (16) to section=BLK BO=17 (16) -- command(s) = putl ' WRITE BREAK IVAL ' -- Break RRVAL -- event(s) = read -- range: section=BLK BO=8 (16) to section=BLK BO=F (16) -- command(s) = putl ' READ BREAK RVAL ' -- Break RIVAL -- event(s) = r -- range: section=BLK BO=10 (16) to section=BLK BO=17 (16) -- command(s) = putl ' READ BREAK IVAL '
run '' WRVAL BREAK ''	HERE WE ARE IN IO1 WRITE BREAK RVAL -- DEBUG: break WRVAL, write at M=IO1 L=11 BO=18
run '' WIVAL BREAK ''	WRITE BREAK IVAL -- DEBUG: break WIVAL, w at M=IO1 L=14 BO=4
chapv n=ival v=1	
dispv ival	ival = 1

(Continued on next page)

Figure 11-2. Command Stream Used and Output from Debug Session

— (Continued from previous page)

DEBUG_INPUT Command Stream	Debug Output
run " WRVAL BREAK "	WRITE BREAK RVAL -- DEBUG: break WRVAL, write at M=I01 L=11 B0=18
dispv rval	rval = 2.
dispv i	i = 2
run " WIVAL BREAK "	WRITE BREAK IVAL -- DEBUG: break WIVAL, w at M=I01 L=14 B0=4
dispv ival	ival = 1
delete_breaks breaks=(wrval,wival)	
disb	-- Break RRVAL -- event(s) = read -- range: section=BLK B0=8 (16) to section=BLK B0=F (16) -- command(s) = putl ' READ BREAK RVAL ' -- Break RIVAL -- event(s) = r -- range: section=BLK B0=10 (16) to section=BLK B0=17 (16) -- command(s) = putl ' READ BREAK IVAL '
run " RRVAL BREAK "	READ BREAK RVAL -- DEBUG: break RRVAL, read at M=MLM$OUTPUT_FLOATING_NUMBER B0=40E(16)
disc	-- Traceback from module MLM$OUTPUT_FLOATING_NUMBER byte offset 40E(16) -- Called from procedure FLP$SEQ_ACC_LST_OUT module FLM$LIST_DIRECTED_IO byte offset 61E(16) -- Called from procedure I01 module I01 at line 16 byte offset 12 -- Called from procedure TEST module TEST at line 54 byte offset 12 -- Called from procedure FLP$INIT module FLM$INIT byte offset 406(16) -- Called from procedure TEST module TEST at line 1 byte offset 22
run " RRVAL BREAK "	READ BREAK RVAL -- DEBUG: break RRVAL, read at M=MLM$OUTPUT_FLOATING_NUMBER B0=536(16)
disc s=3 c=2	-- Traceback from procedure I01 module I01 at line 17 -- Called from procedure TEST module TEST at line 54 byte offset 12

(Continued on next page)—

Figure 11-2. Command Stream Used and Output from Debug Session

— (Continued from previous page)

DEBUG_INPUT Command Stream	Debug Output
run " RRVAL"	READ BREAK RVAL -- DEBUG: break RRVAL, read at M=MLM$OUTPUT_FLOATING_NUMBER B0=CC(16)
disc s=3 c=2	-- Traceback from procedure I01 module I01 at line 17 -- Called from procedure TEST module TEST at line 54 byte offset 12
run " RRVAL"	READ BREAK RVAL -- DEBUG: break RRVAL, read at M=MLM$OUTPUT_FLOATING_NUMBER B0=11C(16)
disc s=3 c=2	-- Traceback from procedure I01 module I01 at line 17 -- Called from procedure TEST module TEST at line 54 byte offset 12
dispv rval m=io1	rval = 10.
run " RRVAL "	READ BREAK RVAL -- DEBUG: break RRVAL, read at M=MLM$OUTPUT_FLOATING_NUMBER B0=14C(16)
DISC C=2	-- Traceback from module MLM$OUTPUT_FLOATING_NUMBER byte offset 14C(16) -- Called from procedure FLP$SEQ_ACC_LST_OUT module FLM$LIST_DIRECTED_IO byte offset 61E(16)
dispv ival m=io1	ival = 5
dispv x1 m=io1	x1 = 3.
dispv y1 m=io1	y1 = 3.
disc	-- Traceback from module MLM$OUTPUT_FLOATING_NUMBER byte offset 14C(16) -- Called from procedure FLP$SEQ_ACC_LST_OUT module FLM$LIST_DIRECTED_IO byte offset 61E(16) -- Called from procedure I01 module I01 at line 16 byte offset 12 -- Called from procedure TEST module TEST at line 54 byte offset 12 -- Called from procedure FLP$INIT module FLM$INIT byte offset 406(16) -- Called from procedure TEST module TEST at line 1 byte offset 22
run " RIVAL"	rval = 10. dval = 2400. READ BREAK IVAL -- DEBUG: break RIVAL, r at M=MLM$OUTPUT_INTEGER B0=3A(16)

(Continued on next page)—

Figure 11-2. Command Stream Used and Output from Debug Session

— (Continued from previous page)

DEBUG_INPUT Command Stream	Debug Output
run " RRVAL"	ival = 5 zval = 2400. READ BREAK RVAL -- DEBUG: break RRVAL, read at M=I01 L=20 B0=8
run " RIVAL- SUB1 "	READ BREAK IVAL -- DEBUG: break RIVAL, r at M=I01 L=21 B0=4 * RETURN TO MAIN PROGRAM RETURN FROM SUBROUTINE I01 Y1 IS GREATER THAN X1
run " RRVAL SUB1 "	HERE WE ARE IN SUB1 READ BREAK RVAL -- DEBUG: break RRVAL, read at M=SUB1 L=8 B0=12
delete_breaks breaks=(rrval,rival)	
disb	-- no breaks set
disc	-- Traceback from procedure SUB1 module SUB1 at line 8 byte offset 12 -- Called from procedure TEST module TEST at line 60 byte offset 12 -- Called from procedure FLP$INIT module FLM$INIT byte offset 406(16) -- Called from procedure TEST module TEST at line 1 byte offset 22
set_break break=sub1a line=19 .. " WERE IN SUB1A- DEFAULT MODULE=SUB1 "	
run "SUB1A"	RESULT = 4815. RESULT IS GREATER THAN 100.0 CALL TO SUB2 FOLLOWS -- DEBUG: break SUB1A, execution at M=SUB1 L=19
setb b=sub2a l=9 m=sub2 " WERE STILL IN SUB1A- THEREFORE HAVE TO " " SPECIFY M=SUB2 IN BREAK DESCRIPTION "	
dispv log	log = TRUE
dispv reslt	reslt = 4815.
disc c=2	-- Traceback from procedure SUB1 module SUB1 at line 19 -- Called from procedure TEST module TEST at line 60 byte offset 12
dispv rval m=io1	rval = 10.
dispv rval	rval = 10.
dispv dval m=io1	dval = 2400.

(Continued on next page)—

Figure 11-2. Command Stream Used and Output from Debug Session

— (Continued from previous page)

DEBUG_INPUT Command Stream	Debug Output
dispv rval	dval = 2400.
dispv zval m=io1	zval = 2400.
dispv zval	zval = 2400.
dispv ival m=io1	ival = 5
dispv ival	ival = 5
run " SUB2A BREAK "	-- DEBUG: break SUB2A, execution at M=SUB2 L=9
setsm on	
run " STEP AT LINE 9 "	-- DEBUG: step at M=SUB2 L=9 HERE WE ARE IN SUB2
setsm on c='dispv r1; disc'	
run " STEP AT LINE 9 "	r1 = 4815. -- Traceback from procedure SUB2 module SUB2 at line 9 -- Called from procedure SUB1 module SUB1 at line 19 byte offset 20 -- Called from procedure TEST module TEST at line 60 byte offset 12 -- Called from procedure FLP$INIT module FLM$INIT byte offset 406(16) -- Called from procedure TEST module TEST at line 1 byte offset 22 -- DEBUG: step at M=SUB2 L=9 HERE WE ARE IN SUB2
run " STEP AT LINE 10 "	r1 = 4815. -- Traceback from procedure SUB2 module SUB2 at line 10 -- Called from procedure SUB1 module SUB1 at line 19 byte offset 20 -- Called from procedure TEST module TEST at line 60 byte offset 12 -- Called from procedure FLP$INIT module FLM$INIT byte offset 406(16) -- Called from procedure TEST module TEST at line 1 byte offset 22 -- DEBUG: step at M=SUB2 L=10
run " STEP AT LINE 11 "	r1 = 4815. -- Traceback from procedure SUB2 module SUB2 at line 11 -- Called from procedure SUB1 module SUB1 at line 19 byte offset 20 -- Called from procedure TEST module TEST at line 60 byte offset 12 -- Called from procedure FLP$INIT module FLM$INIT byte offset 406(16) -- Called from procedure TEST module TEST at line 1 byte offset 22 -- DEBUG: step at M=SUB2 L=11
disde do=ua	-- Execution is currently stopped at B 02E 000005CC which, in higher symbolic terms is M=SUB2 L=11

(Continued on next page)—

Figure 11-2. Command Stream Used and Output from Debug Session

— (Continued from previous page)

DEBUG_INPUT Command Stream	Debug Output

```
setsm off c='disc; run'        --  Default module is $CURRENT.
                               --  Default procedure is $CURRENT.
                               --  Default debug_input file is TESTIN.
                               --  Default debug_output file is OUTPUT.
                               --  The number of breaks set by the user is 3.
                               --  The number of breaks in use by DEBUG is 0.
                               --  The number of available breaks is 61.
                               --  Step_mode is OFF.
                               --  Execution is currently stopped at B 02E 000005CC which, in higher
                               symbolic terms is M=SUB2 L=11
                               FINAL RESULT = 0
                               RETURN FROM SUB2 NOW
                               RETURN FROM SUB2
                               log1 = T
                               LOG3= F

setb b=74 l=74 m=test ..
c='dispv aray; dispv log3; run'    aray =    0   1   0   1   0   1   2   5   3   1   9   6   1   3   9   1
                                   16   17   13   18
                                    log3 =  FALSE
                                    ARAY(1-20) = 0 1 0 1 0 1 2 5 3 1 9 6 1 3 9 1 16 17 13 18

" THE FOLLOWING WILL TERMINATE "
" PROGRAM TEST "
RUN                            --  DEBUG: program terminated by calling exit at M=FLM$TERMINATE
                                  _JOB
                                  P=FLP$TERMINATE_JOB B0=20E(16)

disde                          --  Default module is $CURRENT.
                               --  Default procedure is $CURRENT.
                               --  Default debug_input file is DBIN.
                               --  Default debug_output file is OUTPUT.
                               --  The number of breaks set by the user is 3.
                               --  The number of breaks in use by DEBUG is 0.
                               --  The number of available breaks is 61.
                               --  Step_mode is OFF.
                               --  Execution is currently stopped at B 02C 000204DE which, in higher
                                   symbolic terms is M=FLM$TERMINATE_JOB P=FLP$TERMINATE_JOB B0=20E(16)

quit                           --  DEBUG: QUIT terminated task
```

Figure 11-2. Command Stream Used and Output from Debug Session

Examples 12

This chapter shows complete executable programs along with examples of input, output, and terminal dialog where appropriate. Note that for examples showing actual terminal dialog, the dialog displayed at your terminal might vary slightly depending on the characteristics of the terminal.

Program PASCAL

Program PASCAL, shown in figure 12-1, generates a pascal triangle. The program illustrates the use of DO loops, including nested DO loops and a loop with a negative index parameter.

```
      PROGRAM PASCAL
C
C THIS PROGRAM PRODUCES A PASCAL TRIANGLE.
C
      INTEGER LROW(15)
      DO 10 I=1,15
         LROW(I) = 1
10    CONTINUE
      PRINT '(" PASCAL TRIANGLE"//1X,I5,/1X,2I5)',
     +LROW(15), LROW(14), LROW(15)
      DO 20 J=14,2,-1
         DO 15 K=J,14
            LROW(K) = LROW(K) + LROW(K+1)
15       CONTINUE
         PRINT '(1X,15I5)', (LROW(M), M=J-1,15)
20    CONTINUE
      END
```

Figure 12-1. Program PASCAL

The INTEGER statement declares a 15-word array to be used to contain the elements of a row of the triangle. The first DO loop initializes all elements of the array to 1. The first PRINT statement prints the first two rows of the triangle; the format specification uses the slash descriptor to print multiple lines.

The nested DO loops calculate and print the remaining rows of the triangle. The value of the first and last element of each row is one. Each remaining element is calculated by adding the corresponding element in the preceding row to its preceding element. (For example, the third element of row three is calculated by adding the second and third elements of row 2, and so forth.)

The triangle produced by program PASCAL is shown in figure 12-2.

```
PASCAL TRIANGLE

      1
      1    1
      1    2    1
      1    3    3    1
      1    4    6    4    1
      1    5   10   10    5    1
      1    6   15   20   15    6    1
      1    7   21   35   35   21    7    1
      1    8   28   56   70   56   28    8    1
      1    9   36   84  126  126   84   36    9    1
      1   10   45  120  210  252  210  120   45   10    1
      1   11   55  165  330  462  462  330  165   55   11    1
      1   12   66  220  495  792  924  792  495  220   66   12    1
      1   13   78  286  715 1287 1716 1716 1287  715  286   78   13    1
      1   14   91  364 1001 2002 3003 3432 3003 2002 1001  364   91   14    1
```

Figure 12-2. Program PASCAL Output

Program CORR

Program CORR, shown in figure 12-3, reads two sets of integers from the terminal and calculates a correlation coefficient. Program CORR illustrates the following features:

PARAMETER statement

Interactive input and output

The correlation coefficient measures the correlation between two sets of numbers. A coefficient with a value close to 1 indicates close correlation.

```
      PROGRAM CORR
C
C  ASSIGN SYMBOLIC NAME N TO CONSTANT 10.
C
      PARAMETER (N=10)
C
      INTEGER SUMJ, SUMK, SUMJK, SUMJSQ, SUMKSQ, J(N), K(N)
      REAL NUM
C
C  READ NUMBERS TO BE CORRELATED.
C
10    CONTINUE
      PRINT*, 'ENTER FIRST SET OF ', N, ' NUMBERS'
      READ*, J
      IF (J(1) .EQ. 9999) STOP
      PRINT*, 'ENTER SECOND SET OF ', N, ' NUMBERS'
      READ*, K
C
C  INITIALIZATION.
C
      SUMJ = 0
      SUMK = 0
      SUMJSQ = 0
      SUMKSQ = 0
      SUMJK = 0
C
C  CALCULATE CORRELATION COEFFICIENT.
C
      DO 20 I = 1,N
         SUMJ = SUMJ + J(I)
         SUMK = SUMK + K(I)
         SUMJSQ = SUMJSQ + J(I)**2
         SUMKSQ = SUMKSQ + K(I)**2
         SUMJK = SUMJK + J(I) * K(I)
20    CONTINUE
C
      NUM = REAL(N * SUMJK - SUMJ * SUMK)
      A = REAL(N * SUMJSQ - SUMJ**2)
      B = REAL(N * SUMKSQ - SUMK**2)
      DENOM = SQRT(A) * SQRT(B)
      R = NUM/DENOM
      PRINT 100, R
100   FORMAT (' CORRELATION COEFFICIENT = ',F6.2,//)
      GO TO 10
      END
```

Figure 12-3. Program CORR

Program CORR reads two sets of numbers from the terminal. If the first number of the first set is 9999, the program immediately stops; otherwise, the program performs the calculation and branches to the beginning to request another set of input values. The program is written to process sets containing 10 numbers each. The statement PARAMETER (N=10) assigns the name N to the constant 10. This symbolic constant is used in the DIMENSION statement, the DO statement, and in the statements that calculate the intermediate values NUM, A, and B. The program can be modified to calculate a result for a different number of values simply by changing the PARAMETER statement.

The two PRINT statements at the beginning of the program provide an informative prompt for input. Because the UNIT specifier is omitted from the succeeding READ statements, unit INPUT is implied. When the READ statements are executed, the system prints a question mark, and execution stops until the user types a set of input values.

An example of terminal dialog for program CORR, showing input and output, is shown in figure 12-4.

```
/lgo
ENTER FIRST SET OF 10 NUMBERS
? 1 2 3 4 5 6 7 8 9 10
ENTER SECOND SET OF 10 NUMBERS
? 1 2 3 4 5 6 7 8 9 10
CORRELATION COEFFICIENT =   1.00


ENTER FIRST SET OF 10 NUMBERS
? 5 96 127 0 3 25 84 16 22 50
ENTER SECOND SET OF 10 NUMBERS
? 0 0 4 18 9 56 32 0 0 10
CORRELATION COEFFICIENT =   -.05


ENTER FIRST SET OF 10 NUMBERS
? 3 4 5 3 4 5 3 4 5 3
ENTER SECOND SET OF 10 NUMBERS
? 3 4 5 3 4 5 3 2 1 0
CORRELATION COEFFICIENT =    .39


ENTER FIRST SET
? 9999 0 0 0 0 0 0 0 0 0
```

Figure 12-4. Program CORR Output

Program COMPSAL

Program COMPSAL, shown in figure 12-5, calculates salaries from data input at the terminal. This program illustrates interactive input and output, and the use of block IF structures.

```
      PROGRAM COMPSAL
      CHARACTER NAME*20
      INTEGER AGE
10    CONTINUE
C
C  PROMPT FOR INPUT.
C
      PRINT*, 'TYPE NAME, AGE, AND WAGES'
C
C  READ INPUT VALUES.  UNIT=* READS FROM FILE INPUT.
C  FMT=* SPECIFIES LIST DIRECTED INPUT.
C
      READ (UNIT=*,FMT=*) NAME, AGE, WAGES
C
C  CALCULATE SALARY, DEPENDING ON VALUE OF AGE.
C
      IF (AGE .GT. 65) THEN
         SALARY = WAGES*0.7
      ELSE IF (AGE .GT. 60) THEN
         SALARY = WAGES*0.6
      ELSE IF (AGE .LT. 18) THEN
         SALARY = WAGES*0.52
      ELSE
         SALARY = WAGES
      ENDIF
C
      PRINT 100, SALARY
100   FORMAT (/, ' SALARY IS $', F8.2, //)
C
C  TEST FOR LAST INPUT NAME.
C
      IF (NAME(1:1) .EQ. '/') STOP
C
C  BRANCH BACK TO READ ANOTHER LINE.
C
      GO TO 10
      END
```

Figure 12-5. Program COMPSAL

Program COMPSAL reads a line from the terminal, containing values for NAME, AGE, and WAGES. The UNIT=* specifier in the READ statement causes the program to read from unit INPUT, which implies the terminal in interactive usage.

The program uses a block IF structure to test the value of WAGES and calculate a value for SALARY depending on the result of the test. The program then tests for a slash in the first position of the input line. If a slash is found, execution stops; if no slash is found, control transfers to the beginning to read another input line.

A sample terminal dialog for program COMPSAL is shown in figure 12-6.

```
/lgo
TYPE NAME, AGE, AND WAGES
? 'smith' 70 12000

SALARY IS $ 8400.00


TYPE NAME, AGE, AND WAGES
? 'jones' 33 8000

SALARY IS $ 8000.00


TYPE NAME, AGE, AND WAGES
? '/hansen' 16 1000

SALARY IS $  520.00
```

Figure 12-6. Sample Terminal Dialog for Program COMPSAL

Subroutine COUNTC

Subroutine COUNTC, shown in figure 12-7, counts the number of occurrences of a specified character in the input line. This program illustrates the use of character substrings and a variable length CHARACTER specification.

```
      PROGRAM MAIN
      CHARACTER LINE*80, CHAR*1
      PRINT*, ' TYPE A LINE'
      READ*, LINE
      PRINT*, ' TYPE A CHARACTER'
      READ*, CHAR
      CALL COUNTC (LINE, CHAR, NCHAR)
      PRINT 111, CHAR, NCHAR
111   FORMAT (/, ' CHARACTER ', A1, ' OCCURRED ', I1, ' TIMES ', //)
      END

      SUBROUTINE COUNTC (A, CH, N)
C*
C* DECLARE A TO HAVE THE LENGTH USED IN THE CALL.
C*
      CHARACTER A*(*), CH*1
      N = 0
C*
C* TEST EACH CHARACTER IN INPUT LINE.  IF MATCH IS SUCCESSFUL,
C* INCREMENT COUNTER. IF PERIOD, RETURN.
C*
      DO 10 I = 1,LEN(A)
      IF (A(I:I) .EQ. CH) THEN
          N = N + 1
      ELSE IF (A(I:I) .EQ. '.') THEN
          RETURN
      ENDIF
10    CONTINUE
      END
```

Figure 12-7. Subroutine COUNTC

Subroutine COUNTC has three dummy arguments: Argument A receives the input character string, CH receives the character to be tested, and N returns the number of occurrences of the character passed in CH. The argument A is declared to have length (*). This means that when COUNTC is called, A will have the length of the string passed through A. Thus, a string of any length can be passed (although the subroutine processes no more than 80 characters).

The DO loop contains a block IF structure that tests each character of the input line for the occurrence of the input character. If the input character is detected, a counter is incremented. If the input character is not detected, the ELSE IF statement tests for a period. If a period is found, control returns to the calling program; otherwise, the next character in the line is tested. Each character is tested until either a period is found or 80 characters have been tested; control then returns to the calling program.

The main program in figure 12-7 reads a character string from the terminal, reads a single character, calls COUNTC, and prints the results. Figure 12-8 shows an example of terminal dialog and resulting output.

```
/lgo
 TYPE A LINE
? 'this is the first line.'
 TYPE A CHARACTER
? 't'

CHARACTER t OCCURRED  3 TIMES.
```

Figure 12-8. Sample Terminal Dialog for Subroutine COUNTC.

Program SCLCALL

Program SCLCALL, shown in figure 12-9, illustrates the use of SCL interface calls to reference parameters specified on the execution call command.

```
      PROGRAM SCLCALL
C
C  DEFINE A STRING PARAMETER AND AN INTEGER VARIABLE PARAMETER.
C
C$    PARAM ('P1:STRING; P2:VAR OF INTEGER')
      CHARACTER KIND*8, SVAL*20, VREF*7, VARKND*7
      LOGICAL TSTPARM
      INTEGER VAL, RAD
C
C  TEST FOR PRESENCE OF P1 ON EXECUTION COMMAND.
C
      IF (TSTPARM('P1')) THEN
C
C  GET LENGTH AND VALUE OF P1.  ARGUMENTS setnum AND valnum ARE
C  NOT USED AND ARE SET TO 1.  ARGUMENT lhi IS NOT USED AND IS
C  SET TO 'LOW'.
C
          CALL GETCVAL ('P1', 1, 1, 'LOW', LEN, SVAL)
          PRINT 100, SVAL, LEN
100       FORMAT (' PARAMETER NAME IS P1', /, ' VALUE IS ', A20,
     +/, ' LENGTH IS ', I1, /)
      ELSE
          PRINT*, ' PARAMETER P1 NOT SPECIFIED.'
      ENDIF
C
C  TEST FOR PRESENCE OF P2 ON EXECUTION COMMAND.
C
      IF (TSTPARM('P2')) THEN
C
C  GET NAME AND KIND OF VARIABLE REFERENCE.  ARGUMENTS setnum
C  AND valnum ARE NOT USED AND ARE SET TO 1.  ARGUMENT lhi IS NOT
C  USED AND IS SET TO 'LOW'.
C
          CALL GETVREF ('P2', 1, 1, 'LOW', VREF, VARKND, J, K, L)
C
C  GET VALUE AND BASE OF INTEGER VARIABLE.
C
          CALL REDIVAR (VREF, 1, VAL, RAD)
C
C  PRINT NAME, KIND, AND BASE OF INTEGER VARIABLE.
C
          PRINT 200, VREF, VARKND, RAD
200       FORMAT ( VARIABLE NAME IS ', A7, /, ' KIND IS ', A7, /,
     +'   BASE IS ', I1)

C
C  DETERMINE BASE OF VALUE, THEN PRINT USING PROPER FORMAT.
C
          IF (RAD .EQ. 2 .OR. RAD .EQ. 8) PRINT 201, VAL
201       FORMAT (' VALUE IS ', O20)
          IF (RAD .EQ. 10) PRINT 202, VAL
202       FORMAT (' VALUE IS ', I10)
          IF (RAD .EQ. 16) PRINT 203, VAL
203       FORMAT (' VALUE IS ', Z16)
      ELSE
          PRINT*, ' PARAMETER P2 NOT SPECIFIED.'
      ENDIF
      END
```

Figure 12-9. Program SCLCALL

The C$ PARAM directive defines a string parameter P1, and an integer variable parameter P2. The succeeding SCL calls test for the presence of the parameters on the execution command and, if the parameters are present, return information about the parameters.

Note that in the GETCVAL and GETVREF calls, the values 1, 1, and 'LOW' are supplied for the value number, value set number, and range position arguments, respectively. Even though value set, value list, and range attributes are not defined for P1 and P2, arguments corresponding to those attributes must be specified in the SCL calls.

Figure 12-10 shows examples of two execution commands for program SCLCALL. (The object code is assumed to be on file LGO.) In the first example, no parameters are specified. In the second example, a CREATE_VARIABLE command is entered to define an SCL INTEGER variable named VAR, and the string parameter P1 and variable parameter P2 are specified on the LGO command.

Example 1

```
/lgo
 PARAMETER P1 NOT SPECIFIED.
 PARAMETER P2 NOT SPECIFIED.
```

Example 2

```
/create_variable var kind=integer value=3A(16)
/lgo p1='abcde' p2=var
PARAMETER NAME IS P1
VALUE IS abcde
LENGTH IS  5

VARIABLE NAME IS VAR
KIND IS INTEGER
BASE IS 16
VALUE IS 000000000000003A
```

Figure 12-10. Sample Terminal Dialog for Program SCLCALL

Glossary A

This glossary does not include terms defined in the ANSI standard for FORTRAN, X3.9-1978.

B

Beginning-of-Information (BOI)

File boundary that works the beginning of the file.

Blank Common Block

An unnamed common block. No data can be stored into a blank common block at load time. Contrast with Named Common Block.

Block

A logical or physical grouping of data.

Boolean Constant

A constant that represents a boolean string, octal, or hexadecimal value.

Boolean String Constant

A constant that represents a string of one through eight characters.

Buffer Statement

One of the input/output statements BUFFER IN or BUFFER OUT.

Byte

A group of bits. When used for encoding character data, a byte represents a single character. On NOS/VE, the term implies an 8-bit group.

Byte Addressable File

A file organization in which records are accessed by their byte address.

C

Calling Sequence

A set of instructions used to transfer control to a subprogram.

Character

A letter, digit, or symbol forming part of one or more of the standard character sets. Also, a unit of measure used to specify block length, record length, and so forth. Can be a nonprinting symbol or overpunch representation.

Character Constant

A string of one or more characters delimited by apostrophes.

Close Operation

A set of terminating operations performed on a file when input and output operations are complete.

Collated Key

A key consisting of 1 to 255 8-bit characters. These keys are sorted according to the sequence indicated by a collation table. Contrast with Uncollated Key.

Collating Sequence

The sequence in which characters are ordered for purposes of sorting, mergine, and comparing. The default collating sequence is ASCII. Other collating sequences that can be selected are ASCII6, COBOL6, and DISPLAY. The collating sequence is selected by the routines CSOWN, COLSEQ, and WTSET.

Common Block

An area of memory that can be declared in a COMMON statement by more than one program and used for storage of shared data. See Blank Common Block and Named Common Block.

Compilation Time

The time at which a source program is translated by the compiler to an object program that can be loaded and executed. Contrast with Execution Time.

D

Data Block

A logical or physical grouping of records in which user records are stored in an indexed sequential file.

Default

Assumed value for a parameter when the parameter is not specified by the user.

Default Data Type

The data type assumed by a variable in the absence of any type declarations for the variable. Variables whose names begin with one of the letters A through H or O through Z have a default type of real. Variables whose names begin with one of the letters I through N have a default type of integer. The default typing can be changed by using an IMPLICIT statement.

Direct Access Input/Output

A method of input/output in which records can be read or written in any order.

E

Embedded Key

A key that is contained within a record, as opposed to a key that is defined outside of a record.

End-of-File (EOF)

A particular kind of boundary on a sequential file, recognized by the END= specifier and the functions EOF and UNIT. Either of the following boundaries is recognized as end-of-file:

End-of-partition

End-of-information (EOI)

The ENDFILE statement writes an end-of-partition boundary.

End-of-Information (EOI)

The point at which data in a file ends.

End-of-Partition

A special delimiter in a file with variable record type.

Entry Point

A location within a program unit that can be branched to from other program units. Each entry point has a unique name.

Equivalence Class

A group of variables or arrays whose position relative to each other is defined as a result of an EQUIVALENCE statement.

Execution Time

The time at which a compiled program is executed. Also known as run time.

Extended Hollerith Constant

A constant that represents a string of nine or more characters. Extended Hollerith constants are used only as actual arguments.

External File

A file residing on an external storage device. See File.

External Reference

A reference in one program unit to an entry point in another program unit.

External Storage Device

Disk or magnetic tape.

F

Field

A subdivision of a record that consists of one or more contiguous characters.

File

A collection of information referenced by a name. Files read and written by FORTRAN programs can be classified according to their residence (external and internal files) or their method of access (sequential and direct access files).

File Attribute

A characteristic of a file. Each file has a set of attributes that completely define file structure and processing limitations.

File Information Table

An internal table through which FORTRAN communicates with the internal file interface routines.

File Organization

Defines the way records are stored in a file. The available file organizations are sequential, byte addressable, and indexed sequential.

File Reference

An SCL element that identifies a file and optionally the file position to be established prior to use.

Floating-Point Number

A method of internal binary representation for numbers written with a decimal point; used for FORTRAN types REAL and DOUBLE PRECISION. In floating-point representation, the leftmost bit indicates the sign of the fraction (0 for positive, 1 for negative), the next 15 bits represent the exponent, and the rightmost 48 bits represent the fraction.

Flushing

The process of writing to disk any parts of a file whose images in real memory have been altered or expanded, if the alteration or expansion has not yet been made on disk. Flushing does not alter the logical status or position of a file.

F-Type Record

Fixed-length records, as defined by the ANSI standard.

G

Generic Function Name

The name of an intrinsic function that accepts arguments of any data type. Except for data type conversion generic functions, the data type of the result is the same as the data type of the arguments.

Graphic Character

A character that can be printed or displayed.

H

Hollerith Constant

See Boolean String Constant and Extended Hollerith Constant.

I

Implicit Type

The type of a variable as declared in an IMPLICIT statement.

Indefinite Value

A value that results from a mathematical operation that cannot be resolved, such as a division where the dividend and divisor are both zero.

Index Block

A block in an indexed sequential file that contains ordered keys and pointers to the data blocks or other index blocks.

Index Record

An internal record in an index block that guides the system in locating data records by primary key value. An index record consists of a primary key value and a pointer to a block. The primary key value in the index record matches the primary key value of the first record in either a data block or a lower-level index block.

Indexed Sequential (IS) File

A file organization in which records are accessed using a primary key. Indexed sequential files contain index blocks and data blocks.

Infinite Value

A value that results from a computation whose result exceeds the largest value that can be represented by the computer. The representation of an infinite value in a computer word does not correspond to the representation of a number.

Integer Key

A signed binary key used with indexed sequential files. Integer keys are sorted by arithmetic value.

Internal File

A character variable, array, or substring on which input/output operations are performed by formatted READ and WRITE statements. Internal files provide a method of transferring and converting data from one area of memory to another.

Intrinsic Function

A predefined function that can be called from a FORTRAN program. An intrinsic function calculates a single result and returns it to the calling program.

J

Job

A sequence of tasks executed for a user name.

Job Log

A chronological listing of all operations associated with a terminal from login to logout.

K

Key

A string of characters or a number the user defines to uniquely identify a record.

L

Library

See Source Library and Object Library.

Load time

The time at which an object program is loaded into memory and prepared for execution.

Local File Name

The name used by an executing job to reference a file while the file is assigned to the job's $LOCAL catalog. Only one file can be associated with a given name in a job; however, in one job, a file can have more than one instance of open by that name.

M

Major Key

A high-order portion of a primary key in an indexed sequential file. A major key is used to position a file to a specific record.

Mass Storage

Disk storage.

Mass Storage File

A particular kind of randomly accessible file, accesssed by the mass storage input/output routines.

Mass Storage Input/Output

A type of input/output used for random access to files; it involves the subroutines OPENMS, READMS, WRITMS, CLOSMS, and STINDX.

N

Named Common Block

A common block that has a name. Data can be stored into a named common block at load time. The first program unit declaring a named common block determines the amount of memory allocated. Contrast with Blank Common Block.

Nonembedded Key

A primary key that is not physically contained within the record. A nonembedded key appears before a record that is stored in a data block.

O

Object Code

Executable code produced by the compiler.

Object Library

A file containing one or more modules and a directory to each module.

Open

A set of preparatory operations performed on a file before input and output can occur.

Optimization

The manipulation of object code to reduce execution time. The level of optimization performed by the compiler can be selected through the OPTIMIZATION parameter on the FORTRAN command.

Owncode Routine

A user-written routine that processes certain conditions, such as errors, end of data, and record insertion, substitution, deletion, or modification. Control passes automatically to user owncode routines.

P

Padding

Space deliberately left unused in each block during the creation of a file. The empty space makes later expansion of the file easier.

Partition

A unit of data on a sequential or byte addressable file delimited by end-of-partition separators or the beginning-of-information or the end-of-information.

Pass by Reference

A method of referencing a subprogram in which the addresses of the actual arguments are passed.

Pass by Value

A method of referencing a subprogram in which only the values of the actual arguments are passed.

Permanent File

A file preserved by NOS/VE across job executions and system deadstarts. A permanent file has an entry in a permanent catalog. See file.

Primary Key

The key of a record in an indexed sequential file. The primary key must be defined for a file when the file is first created, an each record in the file must have a unique value for the key.

Procedure

A FORTRAN function subprogram, subroutine subprogram, or statement function.

Program Unit

A sequence of FORTRAN statements terminated by an END statement. The FORTRAN program units are main programs, subroutines, functions, and block data subprograms.

R

Random Access

The process of reading or writing a record in a file without having to read or write the preceding records; applies only to mass storage files. Contrast with Sequential Access.

Random File Organization

A file organization in which records can be accessed by the value of their keys. Random files are processed by direct access READ and WRITE statements, file interface subprograms, and the mass storage subroutines.

Record

A unit of data that can be read or written by a single I/O request.

Record Length

The length of a record measured in words for unformatted input/output and in characters for formatted input/output.

Reference Listing

A part of the listing produced by a FORTRAN compilation, which displays some or all of the entities used by the program, and provides other information such as attributes and location of those entities.

Rewind

Positions a file at the beginning-of-information.

Run Time

See Execution Time.

S

Sequential Access

An access mode in which records are processed in the order (physical or logical) in which they occur on a storage device. Contrast with Random Access.

Sequential File Organization

A file with records stored and retrieved in the order in which they were written. No logical order exists other than the relative physical position.

Source Code

Code written by the programmer in a language (for example, FORTRAN) and input to a compiler.

Source Library

A collection of text units on a file, generated and manipulated by the Source Code Utility (SCU).

Source Listing

A compiler-produced listing of the user's original source program.

Specific Function Name

The name of an intrinsic function that accepts arguments of a particular data type, and returns a result of a particular data type. Contrast with Generic Function Name.

System Command Language

The block-structured interpretive language that provides the interface to the features and capabilities of NOS/VE. All commands and statements are interpreted by SCL before being processed by the system.

T

Task

The instance of execution of a program.

Traceback

A list of subprogram names within a program, beginning with the currently executing subprogram, proceeding backward through the sequence of called subprograms, and ending with the main program.

U

Uncollated Key

A key consisting of 1 to 255 8-bit characters. These keys are sorted by the magnitude of their binary ASCII code values. See Collated Key.

Unit Identifier

An integer constant, or an integer variable with a value of either 0 through 999, or an L format unit name. In input/output statements, it indicates on which unit the operation is to be performed. It can be linked with the actual file name by an OPEN statement. If no OPEN statement is specified, a default file name is used.

U-Type Record

Records for which the record structure is undefined.

V

V-Type Record

Variable-sized record; system default record type. Each V-type record has a record header. The header contains the record length and the length of the preceding record.

W

Working Storage Area

An area allocated by the task to hold data copied by get or put calls to a file.

ASCII Character Set and Collating Weight Tables B

Tables B-1 through B-12 give the ASCII character set, the hexcadecimal character code for each ASCII character, and the weight tables for the following collating sequences:

ASCII: FORTRAN default collating sequence

OSV$ASCII6_FOLDED and OSV$ASCII6_STRICT: NOS FORTRAN 5 default collating sequence.

OSV$COBOL6_FOLDED and OSV$COBOL6_STRICT: NOS COBOL 5 default collating sequence.

OSV$DISPLAY63_FOLDED and OSV$DISPLAY63_STRICT: NOS 63-character display code collating sequence.

OSV$DISPLAY64_FOLDED and OSV$DISPLAY64_STRICT: NOS 64-character display code collating sequence.

OSV$EBCDIC: Full EBCDIC collating sequence.

OSV$EBCDIC6_FOLDED and OSV$EBCDIC6_STRICT: EBCDIC 6-bit subset collating sequence supported by NOS COBOL 5 and SORT 5.

The collation table variants FOLDED and STRICT indicate different mapping of the characters not in the 63 or 64 characters of the original NOS collating sequence. A strict mapping maps all characters not in the original 64 or 63-character set to the ordinal for the space character. A folded mapping maps some characters into ordinals of the original characters and the others into the ordinal value for the space character as shown in the listing of the collating sequence.

Table B-1. ASCII Character Set and Collating Sequence

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
0	00	NULL	Null
1	01	SOH	Start of heading
2	02	STX	Start of text
3	03	ETX	End of text
4	04	EOT	End of transmission
5	05	ENQ	Enquiry
6	06	ACK	Acknowledge
7	07	BEL	Bell
8	08	BS	Backspace
9	09	HT	Horizontal tabulation
10	0A	LF	Line feed
11	0B	VT	Vertical tabulation
12	0C	FF	Form feed
13	0D	CR	Carriage return
14	0E	SO	Shift out
15	0F	SI	Shift in
16	10	DLE	Data link escape
17	11	DC1	Device control 1
18	12	DC2	Device control 2
19	13	DC3	Device control 3
20	14	DC4	Device control 4
21	15	NAK	Negative acknowledge
22	16	SYN	Synchronous idle
23	17	ETB	End of transmission block
24	18	CAN	Cancel
25	19	EM	End of medium
26	1A	SUB	Substitute
27	1B	ESC	Escape
28	1C	FS	File separator
29	1D	GS	Group separator
30	1E	RS	Record separator
31	1F	US	Unit separator
32	20	SP	Space
33	21	!	Exclamation point
34	22	"	Quotation marks
35	23	#	Number sign
36	24	$	Dollar sign
37	25	%	Percent sign
38	26	&	Ampersand
39	27	'	Apostrophe
40	28	(	Opening parenthesis
41	29	)	Closing parenthesis
42	2A	*	Asterisk
43	2B	+	Plus
44	2C	,	Comma
45	2D	-	Hyphen
46	2E	.	Period
47	2F	/	Slant
48	30	0	Zero
49	31	1	One

(Continued on next page)

Table B-1. ASCII Character Set and Collating Sequence

— (Continued from previous page)

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
50	32	2	Two
51	33	3	Three
52	34	4	Four
53	35	5	Five
54	36	6	Six
55	37	7	Seven
56	38	8	Eight
57	39	9	Nine
58	3A	:	Colon
59	3B	;	Semicolon
60	3C	<	Less than
61	3D	=	Equal to
62	3E	>	Greater than
63	3F	?	Question mark
64	40	@	Commercial at
65	41	A	Uppercase A
66	42	B	Uppercase B
67	43	C	Uppercase C
68	44	D	Uppercase D
69	45	E	Uppercase E
70	46	F	Uppercase F
71	47	G	Uppercase G
72	48	H	Uppercase H
73	49	I	Uppercase I
74	4A	J	Uppercase J
75	4B	K	Uppercase K
76	4C	L	Uppercase L
77	4D	M	Uppercase M
78	4E	N	Uppercase N
79	4F	O	Uppercase O
80	50	P	Uppercase P
81	51	Q	Uppercase Q
82	52	R	Uppercase R
83	53	S	Uppercase S
84	54	T	Uppercase T
85	55	U	Uppercase U
86	56	V	Uppercase V
87	57	W	Uppercase W
88	58	X	Uppercase X
89	59	Y	Uppercase Y
90	5A	Z	Uppercase Z
91	5B	[	Opening bracket
92	5C	\	Reverse slant
93	5D	]	Closing bracket
94	5E	^	Circumflex
95	5F	_	Underline
96	60	`	Grave accent
97	61	a	Lowercase a
98	62	b	Lowercase b
99	63	c	Lowercase c

— (Continued on next page)—

Table B-1. ASCII Character Set and Collating Sequence

— (Continued from previous page)

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
100	64	d	Lowercase d
101	65	e	Lowercase e
102	66	f	Lowercase f
103	67	g	Lowercase g
104	68	h	Lowercase h
105	69	i	Lowercase i
106	6A	j	Lowercase j
107	6B	k	Lowercase k
108	6C	l	Lowercase l
109	6D	m	Lowercase m
110	6E	n	Lowercase n
111	6F	o	Lowercase o
112	70	p	Lowercase p
113	71	q	Lowercase q
114	72	r	Lowercase r
115	73	s	Lowercase s
116	74	t	Lowercase t
117	75	u	Lowercase u
118	76	v	Lowercase v
119	77	w	Lowercase w
120	78	x	Lowercase x
121	79	y	Lowercase y
122	7A	z	Lowercase z
123	7B	{	Opening brace
124	7C	|	Vertical line
125	7D	}	Closing brace
126	7E	~	Tilde
127	7F	DEL	Delete

Table B-2. OSV$ASCII6_FOLDED Collating Sequence

Any ASCII codes not listed in this table (ASCII codes 0 through 1F and 7F through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
00	20	SP	Space
01	21	!	Exclamation point
02	22	"	Quotation marks
03	23	#	Number sign
04	24	$	Dollar sign
05	25	%	Percent sign
06	26	&	Ampersand
07	27	'	Apostrophe
08	28	(	Opening parenthesis
09	29	)	Closing parenthesis
10	2A	*	Asterisk
11	2B	+	Plus
12	2C	,	Comma
13	2D	-	Hyphen
14	2E	.	Period
15	2F	/	Slant
16	30	0	Zero
17	31	1	One
18	32	2	Two
19	33	3	Three
20	34	4	Four
21	35	5	Five
22	36	6	Six
23	37	7	Seven
24	38	8	Eight
25	39	9	Nine
26	3A	:	Colon
27	3B	;	Semicolon
28	3C	<	Less than
29	3D	=	Equals
30	3E	>	Greater than
31	3F	?	Question mark
32	40,60	@,`	Commercial at, grave accent
33	41,61	A,a	Uppercase A, lowercase a
34	42,62	B,b	Uppercase B, lowercase b
35	43,63	C,c	Uppercase C, lowercase c
36	44,64	D,d	Uppercase D, lowercase d
37	45,65	E,e	Uppercase E, lowercase e
38	46,66	F,f	Uppercase F, lowercase f
39	47,67	G,g	Uppercase G, lowercase g
40	48,68	H,h	Uppercase H, lowercase h
41	49,69	I,i	Uppercase I, lowercase i
42	4A,6A	J,j	Uppercase J, lowercase j
43	4B,6B	K,k	Uppercase K, lowercase k
44	4C,6C	L,l	Uppercase L, lowercase l
45	4D,6D	M,m	Uppercase M, lowercase m
46	4E,6E	N,n	Uppercase N, lowercase n

(Continued on next page)

Table B-2. OSV$ASCII6_FOLDED Collating Sequence

— (Continued from previous page)

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
47	4F,6F	O,o	Uppercase O, lowercase o
48	50,70	P,p	Uppercase P, lowercase p
49	51,71	Q,q	Uppercase Q, lowercase q
50	52,72	R,r	Uppercase R, lowercase r
51	53,73	S,s	Uppercase S, lowercase s
52	54,74	T,t	Uppercase T, lowercase t
53	55,75	U,u	Uppercase U, lowercase u
54	56,76	V,v	Uppercase V, lowercase v
55	57,77	W,w	Uppercase W, lowercase w
56	58,78	X,x	Uppercase X, lowercase x
57	59,79	Y,y	Uppercase Y, lowercase y
58	5A,7A	Z,z	Uppercase Z, lowercase z
59	5B,7B	[,{,	Opening bracket, opening brace
60	5C,7C	\,|	Reverse slant, vertical line
61	5D,7D	],}	Closing bracket, closing brace
62	5E,7E	^,~	Circumflex, tilde
63	5F	_	Underline

Table B-3. OSV$ASCII6_STRICT Collating Sequence

Any ASCII codes not listed in this table (ASCII codes 0 through 1F and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
00	20	SP	Space
01	21	!	Exclamation point
02	22	"	Quotation marks
03	23	#	Number sign
04	24	$	Dollar sign
05	25	%	Percent sign
06	26	&	Ampersand
07	27	'	Apostrophe
08	28	(	Opening parenthesis
09	29	)	Closing parenthesis
10	2A	*	Asterisk
11	2B	+	Plus
12	2C	,	Comma
13	2D	-	Hyphen
14	2E	.	Period
15	2F	/	Slant
16	30	0	Zero
17	31	1	One
18	32	2	Two
19	33	3	Three
20	34	4	Four
21	35	5	Five
22	36	6	Six
23	37	7	Seven
24	38	8	Eight
25	39	9	Nine
26	3A	:	Colon
27	3B	;	Semicolon
28	3C	<	Less than
29	3D	=	Equals
30	3E	>	Greater than
31	3F	?	Question mark
32	40	@	Commercial at
33	41	A	Uppercase A
34	42	B	Uppercase B
35	43	C	Uppercase C
36	44	D	Uppercase D
37	45	E	Uppercase E
38	46	F	Uppercase F
39	47	G	Uppercase G
40	48	H	Uppercase H
41	49	I	Uppercase I
42	4A	J	Uppercase J
43	4B	K	Uppercase K
44	4C	L	Uppercase L

(Continued on next page)

Table B-3. OSV$ASCII6_STRICT Collating Sequence

— (Continued from previous page)

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
45	4D	M	Uppercase M
46	4E	N	Uppercase N
47	4F	O	Uppercase O
48	50	P	Uppercase P
49	51	Q	Uppercase Q
50	52	R	Uppercase R
51	53	S	Uppercase S
52	54	T	Uppercase T
53	55	U	Uppercase U
54	56	V	Uppercase V
55	57	W	Uppercase W
56	58	X	Uppercase X
57	59	Y	Uppercase Y
58	5A	Z	Uppercase Z
59	5B	[	Opening bracket
60	5C	\	Reverse slant
61	5D	]	Closing bracket
62	5E	^	Circumflex
63	5F	_	Underline

Table B-4. OSV$COBOL6_FOLDED Collating Sequence

Any ASCII codes not listed in this table (ASCII codes 0 through 1F and 7F through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
00	20	SP	Space
01	40,60	@,`	Commercial at, grave accent
02	25	%	Percent sign
03	5B,7B	[,{	Opening bracket, opening brace
04	5F	_	Underline
05	23	#	Number sign
06	26	&	Ampersand
07	27	'	Apostrophe
08	3F	?	Question mark
09	3E	>	Greater than
10	5C,7C	\ ,|	Reverse slant, vertical line
11	5E,7E	^ , ~	Circumflex, tilde
12	2E	.	Period
13	29	)	Closing parenthesis
14	3B	;	Semicolon
15	2B	+	Plus
16	24	$	Dollar sign
17	2A	*	Asterisk
18	2D	-	Hyphen
19	2F	/	Slant
20	2C	,	Comma
21	28	(	Opening parenthesis
22	3D	=	Equals
23	22	"	Quotation marks
24	3C	<	Less than
25	41,61	A,a	Uppercase A, lowercase a
26	42,62	B,b	Uppercase B, lowercase b
27	43,63	C,c	Uppercase C, lowercase c
28	44,64	D,d	Uppercase D, lowercase d
29	45,65	E,e	Uppercase E, lowercase e
30	46,66	F,f	Uppercase F, lowercase f
31	47,67	G,g	Uppercase G, lowercase g
32	48,68	H,h	Uppercase H, lowercase h
33	49,69	I,i	Uppercase I, lowercase i
34	21	!	Exclamation point
35	4A,6A	J,j	Uppercase J, lowercase j
36	4B,6B	K,k	Uppercase K, lowercase k
37	4C,6C	L,l	Uppercase L, lowercase l
38	4D,6D	M,m	Uppercase M, lowercase m
39	4E,6E	N,n	Uppercase N, lowercase n
40	4F,6F	O,o	Uppercase O, lowercase o
41	50,70	P,p	Uppercase P, lowercase p
42	51,71	Q,q	Uppercase Q, lowercase q
43	52,72	R,r	Uppercase R, lowercase r
44	5D,7D	], }	Closing bracket, closing brace
45	53,73	S,s	Uppercase S, lowercase s

(Continued on next page)

Table B-4. OSV$COBOL6_FOLDED Collating Sequence

— (Continued from previous page)

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
46	54,74	T,t	Uppercase T, lowercase t
47	55,75	U,u	Uppercase U, lowercase u
48	56,76	V,v	Uppercase V, lowercase v
49	57,77	W,w	Uppercase W, lowercase w
50	58,78	X,x	Uppercase X, lowercase x
51	59,79	Y,y	Uppercase Y, lowercase y
52	5A,7A	Z,z	Uppercase Z, lowercase z
53	3A	:	Colon
54	30	0	Zero
55	31	1	One
56	32	2	Two
57	33	3	Three
58	34	4	Four
59	35	5	Five
60	36	6	Six
61	37	7	Seven
62	38	8	Eight
63	39	9	Nine

Table B-5. OSV$COBOL6_STRICT Collating Sequence

Any ASCII codes not listed in this table (ASCII codes 0 through 1F and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Collating Position	Sequence (Hexadecimal)	ASCII Code Mnemonic	Graphic or Name or Meaning
00	20	SP	Space
01	40	@	Commercial at
02	25	%	Percent sign
03	5B	[	Opening bracket
04	5F	_	Underline
05	23	#	Number sign
06	26	&	Ampersand
07	27	'	Apostrophe
08	3F	?	Question mark
09	3E	>	Greater than
10	5C	\	Reverse slant
11	5E	^	Circumflex
12	2E	.	Period
13	29	)	Closing parenthesis
14	3B	;	Semicolon
15	2B	+	Plus
16	24	$	Dollar sign
17	2A	*	Asterisk
18	2D	-	Hyphen
19	2F	/	Slant
20	2C	,	Comma
21	28	(	Opening parenthesis
22	3D	=	Equals
23	22	"	Quotation marks
24	3C	<	Less than
25	41	A	Uppercase A
26	42	B	Uppercase B
27	43	C	Uppercase C
28	44	D	Uppercase D
29	45	E	Uppercase E
30	46	F	Uppercase F
31	47	G	Uppercase G
32	48	H	Uppercase H
33	49	I	Uppercase I
34	21	!	Exclamation point
35	4A	J	Uppercase J
36	4B	K	Uppercase K
37	4C	L	Uppercase L
38	4D	M	Uppercase M
39	4E	N	Uppercase N
40	4F	O	Uppercase O
41	50	P	Uppercase P
42	51	Q	Uppercase Q
43	52	R	Uppercase R
44	5D	]	Closing bracket
45	53	S	Uppercase S
46	54	T	Uppercase T
47	55	U	Uppercase U

(Continued on next page)

Table B-5. OSV$COBOL6_STRICT Collating Sequence

(Continued from previous page)

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
48	56	V	Uppercase V
49	57	W	Uppercase W
50	58	X	Uppercase X
51	59	Y	Uppercase Y
52	5A	Z	Uppercase Z
53	3A	:	Colon
54	30	0	Zero
55	31	1	One
56	32	2	Two
57	33	3	Three
58	34	4	Four
59	35	5	Five
60	36	6	Six
61	37	7	Seven
62	38	8	Eight
63	39	9	Nine

Table B-6. OSV$DISPLAY63_FOLDED Collating Sequence

Any ASCII codes not listed in this table (ASCII codes 0 through 1F, 25, and 7F through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
00	41,61	A,a	Uppercase A, lowercase a
01	42,62	B,b	Uppercase B, lowercase b
02	43,63	C,c	Uppercase C, lowercase c
03	44,64	D,d	Uppercase D, lowercase d
04	45,65	E,e	Uppercase E, lowercase e
05	46,66	F,f	Uppercase F, lowercase f
06	47,67	G,g	Uppercase G, lowercase g
07	48,68	H,h	Uppercase H, lowercase h
08	49,69	I,i	Uppercase I, lowercase i
09	4A,6A	J,j	Uppercase J, lowercase j
10	4B,6B	K,k	Uppercase K, lowercase k
11	4C,6C	L,l	Uppercase L, lowercase l
12	4D,6D	M,m	Uppercase M, lowercase m
13	4E,6E	N,n	Uppercase N, lowercase n
14	4F,6F	O,o	Uppercase O, lowercase o
15	50,70	P,p	Uppercase P, lowercase p
16	51,71	Q,q	Uppercase Q, lowercase q
17	52,72	R,r	Uppercase R, lowercase r
18	53,73	S,s	Uppercase S, lowercase s
19	54,74	T,t	Uppercase T, lowercase t
20	55,75	U,u	Uppercase U, lowercase u
21	56,76	V,v	Uppercase V, lowercase v
22	57,77	W,w	Uppercase W, lowercase w
23	58,78	X,x	Uppercase X, lowercase x
24	59,79	Y,y	Uppercase Y, lowercase y
25	5A,7A	Z,z	Uppercase Z, lowercase z
26	30	0	Zero
27	31	1	One
28	32	2	Two
29	33	3	Three
30	34	4	Four
31	35	5	Five
32	36	6	Six
33	37	7	Seven
34	38	8	Eight
35	39	9	Nine
36	2B	+	Plus
37	2D	-	Hyphen
38	2A	*	Asterisk
39	2F	/	Slant
40	28	(	Opening parenthesis
41	29	)	Closing parenthesis
42	24	$	Dollar sign
43	3D	=	Equals
44	20	SP	Space
45	2C	,	Comma
46	2E	.	Period

(Continued on next page)

Table B-6. OSV$DISPLAY6_FOLDED Collating Sequence

— (Continued from previous page)

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
47	23	#	Number sign
48	5B,7B	[,{	Opening bracket, opening brace
49	5D,7D	],}	Closing bracket, closing brace
50	3A	:	Colon
51	22	"	Quotation marks
52	5F	_	Underline
53	21	!	Exclamation point
54	26	&	Ampersand
55	27	'	Apostrophe
56	3F	?	Question mark
57	3C	<	Less than
58	3E	>	Greater than
59	40,60	@,`	Commercial at, grave accent
60	5C,7C	\,|	Reverse slant, vertical line
61	5E,7E	^,~	Circumflex, tilde
62	3B	;	Semicolon

Table B-7. OSV$DISPLAY63_STRICT Collating Sequence

Any ASCII codes not listed in this table (ASCII codes 0 through 1F, 25, and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
00	41	A	Uppercase A
01	42	B	Uppercase B
02	43	C	Uppercase C
03	44	D	Uppercase D
04	45	E	Uppercase E
05	46	F	Uppercase F
06	47	G	Uppercase G
07	48	H	Uppercase H
08	49	I	Uppercase I
09	4A	J	Uppercase J
10	4B	K	Uppercase K
11	4C	L	Uppercase L
12	4D	M	Uppercase M
13	4E	N	Uppercase N
14	4F	O	Uppercase O
15	50	P	Uppercase P
16	51	Q	Uppercase Q
17	52	R	Uppercase R
18	53	S	Uppercase S
19	54	T	Uppercase T
20	55	U	Uppercase U
21	56	V	Uppercase V
22	57	W	Uppercase W
23	58	X	Uppercase X
24	59	Y	Uppercase Y
25	5A	Z	Uppercase Z
26	30	0	Zero
27	31	1	One
28	32	2	Two
29	33	3	Three
30	34	4	Four
31	35	5	Five
32	36	6	Six
33	37	7	Seven
34	38	8	Eight
35	39	9	Nine
36	2B	+	Plus
37	2D	-	Hyphen
38	2A	*	Asterisk
39	2F	/	Slant
40	28	(	Opening parenthesis
41	29	)	Closing parenthesis
42	24	$	Dollar sign
43	3D	=	Equals
44	20	SP	Space
45	2C	,	Comma
46	2E	.	Period
47	23	#	Number sign

(Continued on next page)

Table B-7. OSV$DISPLAY63_STRICT Collating Sequence

— (Continued from previous page)

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
48	5B	[	Opening bracket
49	5D	]	Closing bracket
50	3A	:	Colon
51	22	"	Quotation marks
52	5F	_	Underline
53	21	!	Exclamation point
54	26	&	Ampersand
55	27	'	Apostrophe
56	3F	?	Question mark
57	3C	<	Less than
58	3E	>	Greater than
59	40	@	Commercial at
60	5C	\	Reverse slant
61	5E	^	Circumflex
62	3B	;	Semicolon

Table B-8. OSV$DISPLAY64_FOLDED Collating Sequence

Any ASCII codes not listed in this table (ASCII codes 0 through 1F and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
00	3A	:	Colon
01	41,61	A,a	Uppercase A, lowercase a
02	42,62	B,b	Uppercase B, lowercase b
03	43,63	C,c	Uppercase C, lowercase c
04	44,64	D,d	Uppercase D, lowercase d
05	45,65	E,e	Uppercase E, lowercase e
06	46,66	F,f	Uppercase F, lowercase f
07	47,67	G,g	Uppercase G, lowercase g
08	48,68	H,h	Uppercase H, lowercase h
09	49,69	I,i	Uppercase I, lowercase i
10	4A,6A	J,j	Uppercase J, lowercase j
11	4B,6B	K,k	Uppercase K, lowercase k
12	4C,6C	L,l	Uppercase L, lowercase l
13	4D,6D	M,m	Uppercase M, lowercase m
14	4E,6E	N,n	Uppercase N, lowercase n
15	4F,6F	O,o	Uppercase O, lowercase o
16	50,70	P,p	Uppercase P, lowercase p
17	51,71	Q,q	Uppercase Q, lowercase q
18	52,72	R,r	Uppercase R, lowercase r
19	53,73	S,s	Uppercase S, lowercase s
20	54,74	T,t	Uppercase T, lowercase t
21	55,75	U,u	Uppercase U, lowercase u
22	56,76	V,v	Uppercase V, lowercase v
23	57,77	W,w	Uppercase W, lowercase w
24	58,78	X,x	Uppercase X, lowercase x
25	59,79	Y,y	Uppercase Y, lowercase y
26	5A,7A	Z,z	Uppercase Z, lowercase z
27	30	0	Zero
28	31	1	One
29	32	2	Two
30	33	3	Three
31	34	4	Four
32	35	5	Five
33	36	6	Six
34	37	7	Seven
35	38	8	Eight
36	39	9	Nine
37	2B	+	Plus
38	2D	-	Hyphen
39	2A	*	Asterisk
40	2F	/	Slant
41	28	(	Opening parenthesis
42	29	)	Closing parenthesis
43	24	$	Dollar sign
44	3D	=	Equals
45	20	SP	Space
46	2C	,	Comma
47	2E	.	Period

(Continued on next page)

Table B-8. OSV$DISPLAY64_FOLDED Collating Sequence

— (Continued from previous page)

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
48	23	#	Number sign
49	5B,7B	[,{	Opening bracket, opening brace
50	5D,7D	],}	Closing bracket, closing brace
51	25	%	Percent sign
52	22	"	Quotation marks
53	5F	_	Underline
54	21	!	Exclamation point
55	26	&	Ampersand
56	27	'	Apostrophe
57	3F	?	Question mark
58	3C	<	Less than
59	3E	>	Greater than
60	40,60	@,`	Commercial at, grave accent
61	5C,7C	\,|	Reverse slant, vertical line
62	5E,7E	^,~	Circumflex, tilde
63	3B	;	Semicolon

Table B-9. OSV$DISPLAY64_STRICT Collating Sequence

Any ASCII codes not listed in this table (ASCII codes 0 through 1F and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
00	3A	:	Colon
01	41	A	Uppercase A
02	42	B	Uppercase B
03	43	C	Uppercase C
04	44	D	Uppercase D
05	45	E	Uppercase E
06	46	F	Uppercase F
07	47	G	Uppercase G
08	48	H	Uppercase H
09	49	I	Uppercase I
10	4A	J	Uppercase J
11	4B	K	Uppercase K
12	4C	L	Uppercase L
13	4D	M	Uppercase M
14	4E	N	Uppercase N
15	4F	O	Uppercase O
16	50	P	Uppercase P
17	51	Q	Uppercase Q
18	52	R	Uppercase R
19	53	S	Uppercase S
20	54	T	Uppercase T
21	55	U	Uppercase U
22	56	V	Uppercase V
23	57	W	Uppercase W
24	58	X	Uppercase X
25	59	Y	Uppercase Y
26	5A	Z	Uppercase Z
27	30	0	Zero
28	31	1	One
29	32	2	Two
30	33	3	Three
31	34	4	Four
32	35	5	Five
33	36	6	Six
34	37	7	Seven
35	38	8	Eight
36	39	9	Nine
37	2B	+	Plus
38	2D	-	Hyphen
39	2A	*	Asterisk
40	2F	/	Slant
41	28	(	Opening parenthesis
42	29	)	Closing parenthesis
43	24	$	Dollar sign
44	3D	=	Equals
45	20	SP	Space
46	2C	,	Comma
47	2E	.	Period

(Continued on next page)

Table B-9. OSV$DISPLAY64_STRICT Collating Sequence

— (Continued from previous page)

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
48	23	#	Number sign
49	5B	[	Opening bracket
50	5D	]	Closing bracket
51	25	%	Percent sign
52	22	"	Quotation marks
53	5F	_	Underline
54	21	!	Exclamation point
55	26	&	Ampersand
56	27	'	Apostrophe
57	3F	?	Question mark
58	3C	<	Less than
59	3E	>	Greater than
60	40	@	Commercial at
61	5C	\	Reverse slant
62	5E	^	Circumflex
63	3B	;	Semicolon

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
000	00	NUL	Null
001	01	SOH	Start of heading
002	02	STX	Start of text
003	03	ETX	End of text
004	9C	_	Unassigned
005	09	HT	Horizontal tabulation
006	86	_	Unassigned
007	7F	DEL	Delete
008	97	_	Unassigned
009	8D	_	Unassigned
010	8E	_	Unassigned
011	0B	VT	Vertical tabulation
012	0C	FF	Form feed
013	0D	CR	Carriage return
014	0E	SO	Shift out
015	0F	SI	Shift in
016	10	DLE	Data link escape
017	11	DC1	Device control 1
018	12	DC2	Device control 2
019	13	DC3	Device control 3
020	9D	_	Unassigned
021	85	_	Unassigned
022	08	BS	Backspace
023	87	_	Unassigned
024	18	CAN	Cancel
025	19	EM	End of medium
026	92	_	Unassigned
027	8F	_	Unassigned
028	1C	FS	File separator
029	1D	GS	Group separator
030	1E	RS	Record separator
031	1F	US	Unit separator
032	80	_	Unassigned
033	81	_	Unassigned
034	82	_	Unassigned
035	83	_	Unassigned
036	84	_	Unassigned
037	0A	LF	Line feed
038	17	ETB	End of transmission block
039	1B	ESC	Escape
040	88	_	Unassigned
041	89	_	Unassigned
042	8A	_	Unassigned
043	8B	_	Unassigned
044	8C	_	Unassigned
045	05	ENQ	Enquiry
046	06	ACK	Acknowledge
047	07	BEL	Bell
048	90	_	Unassigned
049	91	_	Unassigned
050	16	SYN	Synchronous idle

(Continued on next page)

Table B-10. OSV$EBCDIC Collating Sequence

— (Continued from previous page)

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
051	93	_	Unassigned
052	94	_	Unassigned
053	95	_	Unassigned
054	96	_	Unassigned
055	04	EOT	End of transmission
056	98	_	Unassigned
057	99	_	Unassigned
058	9A	_	Unassigned
059	9B	_	Unassigned
060	14	DC4	Device control 4
061	15	NAK	Negative acknowledge
062	9E	_	Unassigned
063	1A	SUB	Substitute
064	20	SP	Space
065	A0	_	Unassigned
066	A1	_	Unassigned
067	A2	_	Unassigned
068	A3	_	Unassigned
069	A4	_	Unassigned
070	A5	_	Unassigned
071	A6	_	Unassigned
072	A7	_	Unassigned
073	A8	_	Unassigned
074	5B	[	Opening bracket
075	2E	.	Period
076	3C	<	Less than
077	28	(	Opening parenthesis
078	2B	+	Plus
079	21	!	Exclamation point
080	26	&	Ampersand
081	A9	_	Unassigned
082	AA	_	Unassigned
083	AB	_	Unassigned
084	AC	_	Unassigned
085	AD	_	Unassigned
086	AE	_	Unassigned
087	AF	_	Unassigned
088	B0	_	Unassigned
089	B1	_	Unassigned
090	5D	]	Closing bracket
091	24	$	Dollar sign
092	2A	*	Asterisk
093	29	)	Closing parenthesis
094	3B	;	Semicolon
095	5E	^	Circumflex
096	2D	-	Hyphen
097	2F	/	Slant
098	B2	_	Unassigned
099	B3	_	Unassigned
100	B4	_	Unassigned

(Continued on next page)—

Table B-10. OSV$EBCDIC Collating Sequence

— (Continued from previous page)

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
101	B5	_	Unassigned
102	B6	_	Unassigned
103	B7	_	Unassigned
104	B8	_	Unassigned
105	B9	_	Unassigned
106	7C	|	Vertical line
107	2C	,	Comma
108	25	%	Percent sign
109	5F	_	Underline
110	3E	>	Greater than
111	3F	?	Question mark
112	BA	_	Unassigned
113	BB	_	Unassigned
114	BC	_	Unassigned
115	BD	_	Unassigned
116	BE	_	Unassigned
117	BF	_	Unassigned
118	C0	_	Unassigned
119	C1	_	Unassigned
120	C2	_	Unassigned
121	60	`	Grave accent
122	3A	:	Colon
123	23	#	Number sign
124	40	@	Commercial at
125	27	'	Apostrophe
126	3D	=	Equals
127	22	"	Quotation marks
128	C3	_	Unassigned
129	61	a	Lowercase a
130	62	b	Lowercase b
131	63	c	Lowercase c
132	64	d	Lowercase d
133	65	e	Lowercase e
134	66	f	Lowercase f
135	67	g	Lowercase g
136	68	h	Lowercase h
137	69	i	Lowercase i
138	C4	_	Unassigned
139	C5	_	Unassigned
140	C6	_	Unassigned
141	C7	_	Unassigned
142	C8	_	Unassigned
143	C9	_	Unassigned
144	CA	_	Unassigned
145	6A	j	Lowercase j
146	6B	k	Lowercase k
147	6C	l	Lowercase l
148	6D	m	Lowercase m
149	6E	n	Lowercase n
150	6F	o	Lowercase o
151	70	p	Lowercase p

— (Continued on next page)

Table B-10. OSV$EBCDIC Collating Sequence

— (Continued from previous page)

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
152	71	q	Lowercase q
153	72	r	Lowercase r
154	CB	_	Unassigned
155	CC	_	Unassigned
156	CD	_	Unassigned
157	CE	_	Unassigned
158	CF	_	Unassigned
159	D0	_	Unassigned
160	D1	_	Unassigned
161	7E	_	Unassigned
162	73	s	Lowercase s
163	74	t	Lowercase t
164	75	u	Lowercase u
165	76	v	Lowercase v
166	77	w	Lowercase w
167	78	x	Lowercase x
168	79	y	Lowercase y
169	7A	z	Lowercase z
170	D2	_	Unassigned
171	D3	_	Unassigned
172	D4	_	Unassigned
173	D5	_	Unassigned
174	D6	_	Unassigned
175	D7	_	Unassigned
176	D8	_	Unassigned
177	D9	_	Unassigned
178	DA	_	Unassigned
179	DB	_	Unassigned
180	DC	_	Unassigned
181	DD	_	Unassigned
182	DE	_	Unassigned
183	DF	_	Unassigned
184	E0	_	Unassigned
185	E1	_	Unassigned
186	E2	_	Unassigned
187	E3	_	Unassigned
188	E4	_	Unassigned
189	E5	_	Unassigned
190	E6	_	Unassigned
191	E7	_	Unassigned
192	7B	{	Opening brace
193	41	A	Uppercase A
194	42	B	Uppercase B
195	43	C	Uppercase C
196	44	D	Uppercase D
197	45	E	Uppercase E
198	46	F	Uppercase F
199	47	G	Uppercase G
200	48	H	Uppercase H
201	49	I	Uppercase I

(Continued on next page)—

Table B-10. OSV$EBCDIC Collating Sequence

(Continued from previous page)

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
202	E8	_	Unassigned
203	E9	_	Unassigned
204	EA	_	Unassigned
205	EB	_	Unassigned
206	EC	_	Unassigned
207	ED	_	Unassigned
208	7D	}	Closing brace
209	4A	J	Uppercase J
210	4B	K	Uppercase K
211	4C	L	Uppercase L
212	4D	M	Uppercase M
213	4E	N	Uppercase N
214	4F	O	Uppercase O
215	50	P	Uppercase P
216	51	Q	Uppercase Q
217	52	R	Uppercase R
218	EE	_	Unassigned
219	EF	_	Unassigned
220	F0	_	Unassigned
221	F1	_	Unassigned
222	F2	_	Unassigned
223	F3	_	Unassigned
224	5C	\	Reverse slant
225	9F	_	Unassigned
226	53	S	Uppercase S
227	54	T	Uppercase T
228	55	U	Uppercase U
229	56	V	Uppercase V
230	57	W	Uppercase W
231	58	X	Uppercase X
232	59	Y	Uppercase Y
233	5A	Z	Uppercase Z
234	F4	_	Unassigned
235	F5	_	Unassigned
236	F6	_	Unassigned
237	F7	_	Unassigned
238	F8	_	Unassigned
239	F9	_	Unassigned
240	30	0	Zero
241	31	1	One
242	32	2	Two
243	33	3	Three
244	34	4	Four
245	35	5	Five
246	36	6	Six
247	37	7	Seven
248	38	8	Eight
249	39	9	Nine
250	FA	_	Unassigned
251	FB	_	Unassigned
252	FC	_	Unassigned
253	FD	_	Unassigned
254	FE	_	Unassigned
255	FF	_	Unassigned

Table B-11. OSV$EBCDIC6_FOLDED Collating Sequence

Any ASCII codes not listed in this table (ASCII codes 0 through 1F and 7F through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
00	20	SP	Space
01	2E	.	Period
02	3C	<	Less than
03	28	(	Opening parenthesis
04	2B	+	Plus
05	21	!	Exclamation point
06	26	&	Ampersand
07	24	$	Dollar sign
08	2A	*	Asterisk
09	29	)	Closing parenthesis
10	3B	;	Semicolon
11	5E,7E	^,~	Circumflex, tilde
12	2D	-	Hyphen
13	2F	/	Slant
14	2C	,	Comma
15	25	%	Percent sign
16	5F	_	Underline
17	3E	>	Greater than
18	3F	?	Question mark
19	3A	:	Colon
20	23	#	Number sign
21	40,60	@,`	Commercial at, grave accent
22	27	'	Apostrophe
23	3D	=	Equals
24	22	"	Quotation marks
25	5B,7B	[,{	Opening bracket, opening brace
26	41,61	A,a	Uppercase A, lowercase a
27	42,62	B,b	Uppercase B, lowercase b
28	43,63	C,c	Uppercase C, lowercase c
29	44,64	D,d	Uppercase D, lowercase d
30	45,65	E,e	Uppercase E, lowercase e
31	46,66	F,f	Uppercase F, lowercase f
32	47,67	G,g	Uppercase G, lowercase g
33	48,68	H,h	Uppercase H, lowercase h
34	49,69	I,i	Uppercase I, lowercase i
35	5D,7D	],}	Closing bracket, closing brace
36	4A,6A	J,j	Uppercase J, lowercase j
37	4B,6B	K,k	Uppercase K, lowercase k
38	4C,6C	L,l	Uppercase L, lowercase l
39	4D,6D	M,m	Uppercase M, lowercase m
40	4E,6E	N,n	Uppercase N, lowercase n
41	4F,6F	O,o	Uppercase O, lowercase o
42	50,70	P,p	Uppercase P, lowercase p
43	51,71	Q,q	Uppercase Q, lowercase q
44	52,72	R,r	Uppercase R, lowercase r
45	5C,7C	\,|	Reverse slant, vertical line
46	53,73	S,s	Uppercase S, lowercase s
47	54,74	T,t	Uppercase T, lowercase t

(Continued on next page)

Table B-11. OSV$EBCDIC6_FOLDED Collating Sequence

(Continued from previous page)

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
48	55,75	U,u	Uppercase U, lowercase u
49	56,76	V,v	Uppercase V, lowercase v
50	57,77	W,w	Uppercase W, lowercase w
51	58,78	X,x	Uppercase X, lowercase x
52	59,79	Y,y	Uppercase Y, lowercase y
53	5A,7A	Z,z	Uppercase Z, lowercase z
54	30	0	Zero
55	31	1	One
56	32	2	Two
57	33	3	Three
58	34	4	Four
59	35	5	Five
60	36	6	Six
61	37	7	Seven
62	38	8	Eight
63	39	9	Nine

Table B-12. OSV$EBCDIC6_STRICT Collating Sequence

Any ASCII codes not listed in this table (ASCII codes 0 through 1F and 60 through FF hexadecimal) are ordered as equal to the space (ASCII code 20 hexadecimal).

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
00	20	SP	Space
01	2E	.	Period
02	3C	<	Less than
03	28	(	Opening parenthesis
04	2B	+	Plus
05	21	!	Exclamation point
06	26	&	Ampersand
07	24	$	Dollar sign
08	2A	*	Asterisk
09	29	)	Closing parenthesis
10	3B	;	Semicolon
11	5E	^	Circumflex
12	2D	-	Hyphen
13	2F	/	Slant
14	2C	,	Comma
15	25	%	Percent sign
16	5F	_	Underline
17	3E	>	Greater than
18	3F	?	Question mark
19	3A	:	Colon
20	23	#	Number sign
21	40	@	Commercial at
22	27	'	Apostrophe
23	3D	=	Equals
24	22	"	Quotation marks
25	5B	[	Opening bracket
26	41	A	Uppercase A
27	42	B	Uppercase B
28	43	C	Uppercase C
29	44	D	Uppercase D
30	45	E	Uppercase E
31	46	F	Uppercase F
32	47	G	Uppercase G
33	48	H	Uppercase H
34	49	I	Uppercase I
35	5D	]	Closing bracket
36	4A	J	Uppercase J
37	4B	K	Uppercase K
38	4C	L	Uppercase L
39	4D	M	Uppercase M
40	4E	N	Uppercase N
41	4F	O	Uppercase O
42	50	P	Uppercase P
43	51	Q	Uppercase Q
44	52	R	Uppercase R
45	5C	\	Reverse slant
46	53	S	Uppercase S
47	54	T	Uppercase T

(Continued on next page)

Table B-12. OSV$EBCDIC6_STRICT Collating Sequence

(Continued from previous page)

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
48	55	U	Uppercase U
49	56	V	Uppercase V
50	57	W	Uppercase W
51	58	X	Uppercase X
52	59	Y	Uppercase Y
53	5A	Z	Uppercase Z
54	30	0	Zero
55	31	1	One
56	32	2	Two
57	33	3	Three
58	34	4	Four
59	35	5	Five
60	36	6	Six
61	37	7	Seven
62	38	8	Eight
63	39	9	Nine

Differences Between NOS/VE FORTRAN and NOS FORTRAN 5 C

This appendix presents the differences between FORTRAN 5 and the first released version of NOS/VE FORTRAN, and is intended as an aid to converting programs from FORTRAN 5 to NOS/VE FORTRAN.

NOS/VE FORTRAN is designed to be compatible with FORTRAN 5. However, the new operating system and hardware have resulted in several areas of incompatibility. Other incompatibilities are the result of FORTRAN 5 features which are not currently supported under NOS/VE FORTRAN but for which future support is anticipated.

In some cases, language incompatibilities may necessitate program modification; in other cases, statements using incompatible features can remain in the program but will not be processed.

Two forms of differences are given. The general guidelines describe common programming practices which are not compatible between the two versions of FORTRAN. These practices are dependent on the specific characteristics of the hardware systems used by FORTRAN, such as word length and number of characters per word. The feature differences describe specific features for which incompatibilities exist.

General Guidelines

The following programming practices have different results in NOS/VE FORTRAN and FORTRAN 5. FORTRAN 5 programs that use these practices will probably require modification before they can be successfully processed under NOS/VE FORTRAN.

Coding that depends on the internal representation of data (floating-point layout, number of characters per word, and so forth) should be checked. Because of differences in word size and internal representations, these uses nearly always require modification.

Data manipulations based on the binary representation of the data should be checked. FORTRAN 5 programs that manipulate characters as octal display-coded values or as 6-bit binary digits must be modified before being compiled and executed under NOS/VE FORTRAN.

File structure and naming conventions differ significantly under NOS/VE, and default file positioning has changed. All usages that depend on any of these properties should be checked.

Code that identifies or classifies information based on the location of a specific value within a specific set of central memory word bits must be modified.

Intermixed COMPASS subprograms are not supported under NOS/VE FORTRAN. COMPASS subprograms must be replaced by equivalent FORTRAN routines before compilation and execution under NOS/VE FORTRAN.

Feature Differences

The following paragraphs describe specific differences between FORTRAN 5 and NOS/VE FORTRAN.

CYBER Record Manager (AAM) Subprograms

All uses of the CYBER Record Manager subprogram calls should be checked. Under FORTRAN 5, CRM offered several features within which choices could be made. However, the file interface calls under NOS/VE offers only a subset of these features. The following notes are applicable to any user of CYBER Record Manager Advanced Access Methods Version 2 (NOS and NOS/BE) converting FORTRAN 5 programs programs to NOS/VE FORTRAN programs.

Alternate Keys

Alternate keys are not available in the file interface to FORTRAN.

File Organization

Indexed sequential file organization is the only file organization available in the file interface to FORTRAN.

Record Type

The record types available are fixed-length, variable-length, and undefined-length. Advanced Access Methods Version 2 record types D, R, S, T, and Z type records are not available in the file interface to FORTRAN.

File Information Table

User programs do not need to reserve 35 words for the file information table. All that is needed is room for a one-word pointer. If the program does reserve 35 words, only the first word will be used.

Values can be stored or fetched from the file information table in standard ways, i.e., CALL FILEIS, CALL STOREF, CALL IFETCH, and IFETCH. Values in the file information table can only be modified through the file processing calls because the file information table is an internal table which cannot be accessed directly by a program.

Any attempt to read from the table without using IFETCH will return an undefined value. If the field being read has been stored in an unconventional manner, then the value will not be returned.

Keywords must be enclosed in apostrophes; for example, 'WSA'. Boolean forms like 3LWSA (FTN4) or L"WSA" (FTN5) will not work.

The following CYBER Record Manager file information fields do not have equivalents in file interface to FORTRAN:

BAL	HMB	PC
BBH	HRL	PEF
BFF	IBL	PKA
BFS	IRS	PM
BN	KNE	PNO
BT	KR	POS
BZF	LA	PTL
B8F	LAC	RC
CDT	LBL	RDR
CL	LCR	RMK
CM	LGX	SB
CNF	LL	SBF
CP	LNG	SDS
CPA	LOP5	SES
C1	LP	SOL
DCA	LT	SPR
DFLG	LVL	TL
DKI	LX	TRC
EFC	MFN	ULP
EO	MNB	VF
EOFWA	MUL	VNO
EXD	NDX	WA
FPB	NOFCP	WPN
FWB	ORG	XBS
HB	OVF	XN
HL		

Field FL, although not applicable to file interface to FORTRAN, will be recognized as a synonym of field MRL.

Other keywords from Advanced Access Methods Version 2 and their meanings for the file interface to FORTRAN:

DX Data exit. There is no mechanism for data exit in NOS/VE system requests, but the FORTRAN interface saves the subroutine address, and calls the subroutine when the appropriate file position is returned from an access operation.

OC Open/closed flag. Although NOS/VE system requests tell whether the file is opened or closed, the file information table will also contain this information so you can read it by a IFETCH operation.

ON Old/new flag. The file information table maintains a value of ON which can be set to OLD (default) or NEW by a FILEIS call. When a CALL OPENM statement is issued, the FORTRAN interface first finds out from the system whether the file already exists. If the answer to this question conflicts with the setting of ON, a fatal error occurs.

FNF Fatal error flag. To allow you to read the information with a fetch request, the FORTRAN interface maintains this information in the file information table.

KP Key position. This keyword, although it has no meaning in AAM NOS/VE, will be accepted by the FORTRAN interface as a keyword in the CALL FILEIS statement or as a parameter in the CALL STARTM, CALL STOREF, and CALL GET statements. KP will be added to KA to determine the position of the key.

RKW Relative key word. If RKW is specified in a CALL FILEIS statement, the interface will multiply the value by 10 and add it to RKP. This may be a problem because NOS/VE has a word size of 8 bytes and not 10 bytes (NOS and NOS/BE). Users should visually inspect the program to ensure that the correct value is specified.

Reserving Space For WSA

A problem may occur if your FORTRAN 5 program uses either an INTEGER or a REAL array for WSA because NOS and NOS/BE have word size of 10 bytes (NOS/VE has word size of 8 bytes). For example, in NOS/VE FORTRAN, coding a statement like RECORD (8) reserves only 64 characters of space (as opposed to 80 characters in FORTRAN 5), and the first time a record is read into the area, the record overwrites the next item in memory. For this reason, your NOS/VE FORTRAN program should always declare WSA as character data type.

Optimization

FORTRAN optimization (OPT=HIGH) can cause unpredictable results when WSA and/or KA are not in common. If OPT=HIGH is to be used, WSA and KA should be declared as COMMON.

Embedded Keys

The default for EMK in Advanced Access Methods Version 2 was NO (nonembedded keys). The default for the file interface to FORTRAN is YES (embedded keys).

CALL GETNR Statement

For purposes of compatibility, CALL GETNR statement is allowed. CALL GETNR is treated as a CALL GETN.

CALL SEEK Statement

The SEEK function does not exist in the file interface to FORTRAN. If a CALL SEEKF is encountered, the FORTRAN interface copies parameters to the file information table, sets the FILE_POSITION field to end-of-information (EOR), and returns control to the program.

Post Mortem Dump

NOS/VE FORTRAN does not support the Post Mortem Dump debugging facility. The calls to PMDARRY, PMDDUMP, PMDLOAD, and PMDSTOP are provided but are ignored during compilation and execution.

Overlays and OVCAPs

Overlays and OVCAPs are not meaningful in the NOS/VE FORTRAN environment and are not supported. All OVERLAY and OVCAP directives should be removed from programs being converted. PROGRAM statements in primary and secondary overlays should be changed to SUBROUTINE statements. Calls to OVERLAY, LOVCAP, and XOVCAP should be replaced by appropriate subroutine calls. UOVCAP calls should be removed.

Static Memory Management

The static memory management routines are not supported by NOS/VE FORTRAN.

Subroutine LABEL

Subroutine LABEL is not supported by NOS/VE FORTRAN. A LABEL subroutine is provided but it performs no operation.

Subroutine GETPARM

Subroutine GETPARM is replaced by the SCL interface capability under NOS/VE FORTRAN. A GETPARM subroutine is provided but it performs no operation.

Subroutines CHEKPTX and RECOVR

Subroutines CHEKPTX and RECOVR are not supported by NOS/VE FORTRAN. CHEKPTX and RECOVR subroutines are provided but perform no operations.

DATE, TIME, and CLOCK Functions

The values returned by the DATE, TIME, and CLOCK functions have different formats. The format provided by NOS/VE FORTRAN is described in chapter 8. Note that the length declared for TIME and CLOCK by the CHARACTER statement must be changed to 8. (The length of DATE is still 10.)

LOCF Function

The LOCF function is not supported under NOS/VE FORTRAN.

SECOND Function

Under NOS/VE FORTRAN, the SECOND function is supplied as a utility subprogram rather than an intrinsic function. Thus, any FORTRAN 5 programs that declare the SECOND function in an INTRINSIC statement should be changed to declare the function in an EXTERNAL statement.

ENCODE/DECODE

Most usages of ENCODE/DECODE involve packing and unpacking of characters within a word and are dependent on the number of characters per word. All usages should be checked. Conversion of ENCODE/DECODE to FORTRAN standard internal READ and WRITE is recommended.

Hollerith Constants

Under NOS/VE FORTRAN, Hollerith constants are replaced by boolean string constants, which are limited to 8 characters. Constants of the form nHs, L"s", R"s", or "s" that exceed 8 characters can be passed as actual arguments to external procedures. These constants are called extended Hollerith constants.

Buffer I/O

Some uses of buffer input/output, such as the ubc value returned by LENGTH/LENGTHX and the size of the storage area to receive incoming data, are dependent on the number of characters per word. The parity indicator (p parameter) is ignored by NOS/VE FORTRAN. The BUFFER statements are included in NOS/VE FORTRAN for compatibility only. Because buffers are not used in NOS/VE FORTRAN in the same way as in FORTRAN 5, BUFFER statements are generally not advantageous; unformatted READ and WRITE statements should be used instead.

O and Z Editing

Under FORTRAN 5, reading a blank field with the Ow or Zw descriptor gives a minus zero. Under NOS/VE FORTRAN, no minus zero exists (a positive zero is stored).

All list items used with the O and Z descriptors should be declared type boolean.

Boolean Data Type

The boolean data types and operations (SHIFT, MASK, and so forth) are provided specifically for machine-dependent uses. Most uses will require program modification.

Double Precision Functions Referenced as Single Precision

Referencing double precision functions as single precision under FORTRAN 5 depends on register conventions that are not compatible with NOS/VE FORTRAN. All such uses should be removed.

8 Bit Subroutines

The 8 bit subroutines are not supported under NOS/VE FORTRAN.

Permanent File Subroutines

The permanent file subroutines are not supported under NOS/VE FORTRAN. A similar capability is provided by the SCL interface subprograms.

Maximum Length of Formatted Records

The maximum length of formatted records is reduced from 131071 octal under FORTRAN 5 to 65535 octal under NOS/VE FORTRAN.

PROGRAM Statement

The file buffer length specifier on the NOS/VE FORTRAN PROGRAM statement is included for compatibility with FORTRAN 5 but is disregarded by the compiler. Because of the way in which buffers are used in the NOS/VE FORTRAN environment, assigning buffer lengths is not meaningful.

FORTRAN Command

The FORTRAN command for NOS/VE FORTRAN differs from the FTN5 statement for FORTRAN 5. Parameter names have changed, new parameters are available, and certain FTN5 parameters are no longer supported.

Default Collating Sequence

The default collating sequence established when the DEFAULT_COLLATION parameter is omitted from the FORTRAN command has been changed from USER to FIXED. Under NOS/VE FORTRAN, the USER and FIXED collating sequences are defined as the 'ASCII' and 'DISPLAY' collating sequences, respectively. Under FORTRAN 5, USER and FIXED are defined as 'DISPLAY' and 'ASCII6', respectively.

Default Debugging Options

Under NOS/VE FORTRAN, runtime range checking of subscript and substring expressions is performed by default, and is suppressed by a FORTRAN command option. Under FORTRAN 5, this option is off by default and must be selected by a control statement option.

Other Collating Sequence Differences

NOS/VE FORTRAN uses standard system-defined collating sequences for the NOS-compatible 'ASCII6' and 'COBOL6' collating sequences. The 'STANDARD' sequence of FORTRAN 5 has been eliminated, and an 'INSTALL' sequence, equivalent to 'COBOL6', has been added.

Procedure Communication

Any method of procedure communication, other than through common or an argument list, should be changed to use either common or an argument list.

Files INPUT and OUTPUT

The system files INPUT and OUTPUT have been changed to $INPUT and $OUTPUT. Under NOS/VE, the FORTRAN compiler converts all references to INPUT or OUTPUT on the PROGRAM statement, and all references to unit *, to reference $INPUT or $OUTPUT. OPEN statements must be changed to specify $INPUT or $OUTPUT. Since $OUTPUT cannot be written to in a batch environment, it must be connected to a physical file containing data by the PROGRAM statement or by an SCL CREATE_FILE_CONNECTION command.

SAVE Statement

Under NOS/VE FORTRAN, local variables and arrays in subprograms compiled at OPT=HIGH do not retain their values after an exit from the subprogram, unless the subprogram contains a SAVE statement or the FORCED_SAVE option is specified on the FORTRAN command.

Segment Loading

Segment loading is not supported under NOS/VE. In order to avoid conflicts in common block storage within segmented programs, the names of nonglobal common blocks having the same name in parallel parts of the tree structure must be changed to be unique.

Floating-Point Arithmetic

Differences in NOS and NOS/VE unrounded floating-point arithmetic can lead to different results if the source algorithm is numerically unstable.

Permanent File Subroutines

The permanent file subroutines are not supported under NOS/VE FORTRAN. A similar capability is provided by the SCL interface subprograms.

Maximum Length of Formatted Records

The maximum length of formatted records is reduced from 131071 octal under FORTRAN 5 to 65535 octal under NOS/VE FORTRAN.

Intrinsic Function References in Constant Expressions

NOS/VE FORTRAN allows intrinsic function references in any constant expression. NOS FORTRAN 5 allows intrinsic function references in constant expressions only in PARAMETER statements.

C$ Directives D

A C$ directive is a special form of comment line that controls compiler processing. A particular C$ directive affects an aspect of the compiler's interpretation of those lines following the directive and preceding either a subsequent directive modifying the same aspect, if such a directive appears, or the end of the program unit. The aspects of interpretation that can be controlled are

Listing of the program and associated compiler-produced information, called listing control

Specification of program lines to be processed or ignored, called conditional compilation

Character data comparison collation table, called collation control

Minimum trip count for DO loops, called DO loop control

Definition of SCL parameters to be passed through the execution command (C$ PARAM directive, described in chapter 10)

The general form of a C$ directive is as follows:

C$ keyword (p=c,...,p=c),*lab*

keyword One of LIST, IF, ELSE, ENDIF, COLLATE, or DO. In nonsequenced mode, the keyword must begin in or after character position 7; positions 3 through 6 must be blank. In sequenced mode the keyword can begin in any position following the character $.

p A parameter that depends on the specified keyword and the desired option.

c An integer constant expression or integer symbolic constant with a value of 0 or 1. If a symbolic constant is specified, it must have been defined in a previous PARAMETER statement in the same program unit. Depending on the parameter p, the constant either is optional or must appear. If c is omitted, the preceding equal sign is also omitted.

lab Optional label, which must conform to the rules for symbolic names. The use of a symbolic name as a label does not affect, and is not affected by, its use for any other purpose in the program unit. If a label appears and no parameters are present, a comma must separate the keyword and the label. The label is valid only for the IF, ELSE, and ENDIF directives.

The letter C in position 1 together with the character $ in position 2 identify a line as a C$ directive line. Such a line will be interpreted as a comment only if the COMPILATION_DIRECTIVES parameter is selected on the FORTRAN command. The entire directive must appear on a single line. A C$ directive interrupts statement continuation.

In sequenced mode the letter C in the position immediately to the right of the sequence number together with the character $ immediately to the right of the C identify a C$ directive line. A line with no sequence number in sequenced mode cannot be a C$ directive.

A C$ directive containing a syntax error generally results in a fatal compilation diagnostic.

Listing Control

The listing control directive controls the compiler output list options. This directive has the form

C$ LIST(p=*c*,...,*p=c*)

p One of the symbols S, O, R, A, M, or ALL.

c Optional integer constant:

0 Enable the specified option.

1 Disable the specified option.

If =c is omitted, the effect is the same as p=1.

The listing control directive modifies the state of any initially enabled list option switches. A list option switch is initially enabled when the corresponding list option is requested by the LIST_OPTION parameter on the FORTRAN command. Any attempt to modify a list option switch that was not initially enabled is ignored. A specification of p=0 disables switch p; p=1 enables switch p.

ALL=c is equivalent to S=c, O=c, R=c, A=c, M=c.

A listing control parameter with values other than 0 or 1 results in a warning diagnostic.

The list option switches provide the following control:

S Source lines are listed when enabled.

O Generated object code is listed for statements processed when enabled.

R Symbol references are accumulated for the cross-reference list when enabled. Symbols with no accumulated references will not appear in that list; no accumulation for an entire program unit suppresses cross-reference list.

A The symbol attribute list is generated if this switch is enabled when the END statement is processed.

M The symbol attribute list, DO loop, and common/equivalence map lists are generated if this switch is enabled when the program END statement is processed.

The following example illustrates the listing control directives. All source statements appearing between C$ LIST (S=0) and C$ LIST (S=1) are suppressed in the output listing. (Source statement lines with errors are listed on the file $ERRORS along with diagnostics.) The C$ LIST (ALL=0) directive, active when the END statement is encountered, suppresses the reference map.

```
      PROGRAM P
C     PROGRAM TO TEST LISTING CONTROL DIRECTIVES.

C$    LIST(S=0)
      DIMENSION A(10)

C     THE FOLLOWING CARD CONTAINS A SYNTAX ERROR
C     THE ERROR MESSAGE WILL BE LISTED ON THE $ERRORS FILE.
      INTEGER B/C

C$    LIST(S=1)
      DO 100 I=1,10
      A(I) = 0.0
100   CONTINUE
C$    LIST(ALL=0)
      END
```

Conditional Compilation

A conditional compilation directive controls whether the lines immediately following the directive are to be processed or ignored by the compiler.

The conditional compilation directives are divided into three categories:

An IF directive with the keyword IF

An ELSE directive with keyword ELSE

An ENDIF directive with keyword ENDIF

The IF directive, ELSE directive, and ENDIF directive have the following forms:

C$ IF(lexp), *lab*

C$ ELSE, *lab*

C$ ENDIF, *lab*

lexp Extended logical constant expression. If a symbolic constant appears, it must have been previously defined in a PARAMETER statement in the program containing the IF directive.

lab Optional label.

For each IF directive there must appear exactly one ENDIF directive later in the same program unit, and for each ENDIF directive there must appear exactly one IF directive earlier in the same program unit. Between an IF directive and its corresponding ENDIF directive will appear zero or more lines called a conditional sequence. A conditional sequence can optionally contain one ELSE directive corresponding to the IF directive and ENDIF directive delimiting the conditional sequence. An ELSE directive can appear only within a conditional sequence. A conditional sequence cannot contain more than one ELSE directive unless it contains another conditional sequence. If an ELSE directive is contained within more than one conditional sequence, the ELSE directive corresponds to that IF-ENDIF pair which delimits the smallest, that is, innermost, conditional sequence containing the ELSE directive.

If corresponding IF, ELSE, and ENDIF directives have a label, it must be the same label. No other restriction applies to labels on conditional directives. There is no requirement that any conditional directive have a label. The same label can be used on more than one sequence of corresponding conditional directives in a single program unit, including the case of conditional directives whose conditional sequence contains other conditional directives with the same label.

A conditional sequence can contain any number of properly corresponding conditional directives, and therefore other conditional sequences. If two conditional sequences contain the same line, one conditional sequence must lie wholly within the other conditional sequence.

If an IF directive is processed by the compiler and the logical expression has the value true, following lines are processed as if the IF directive had not appeared, unless a corresponding ELSE directive is encountered. In this case, lines between the ELSE directive and the corresponding ENDIF directive are ignored by the compiler. If an IF directive is processed by the compiler and the logical expression has the value false, the following lines are ignored until the corresponding ENDIF directive is encountered, unless a corresponding ELSE directive is encountered. In this case, lines between the ELSE directive and the corresponding ENDIF directive are processed.

The following example illustrates conditional compilation directives. The sample program contains two DO loops. Conditional compilation directives are included to test the value of the symbolic constant M. If M is 0, the first loop is ignored and the second loop is compiled. If M is 1, the first loop is compiled and the second loop is ignored.

```
      PROGRAM B
      PARAMETER (M=1)
      DIMENSION A(10)
      DATA A/10*0.0/

C$    IF(M .EQ. 0)
      DO 8 I=1,10
      A(I) = A(I) + 1.0
8     CONTINUE

C$    ELSE
      DO 12 I=1,10
      A(I) = A(I) - 1.0

C$    ENDIF

      PRINT*, 'A= ', A
      END
```

Collation Control

The collation control directive specifies whether collation of character relational expressions is directed by the fixed or user-specified weight table. This directive has the form

C$ COLLATE(p)

p One of the following:

FIXED Collate according to the fixed (ASCII) weight table.

USER Collate according to the user-specified (DISPLAY) weight table.

A collation control directive directs the interpretation of character relational expressions and of CHAR or ICHAR intrinsic function references in the lines following the directive and preceding either another collation control directive or the END statement of the program unit. In the case of a character relational expression or a CHAR or ICHAR reference in a statement function definition, the collation that applies is that in effect for the line or lines containing a reference to the statement function. The following example shows a character relational expression used in a statement function:

```
      PROGRAM P
      LOGICAL LSF
      CHARACTER*5, X, Y, S, T
C$    COLLATE(USER)
      LSF(X,Y) = X.LT.Y
      .
      .
      .
C$    COLLATE(FIXED)
      IF (LSF(S,T)) A=1.0
      .
      .
      .
      END
```

The reference LSF(S,T) results in an evaluation of the character relational expression S.LT.T according to the fixed weight table.

DO Loop Control

The DO loop control directive controls the minimum trip count for DO loops. This directive has the form

C$ DO (OT=*c*)

c Integer constant or integer symbolic constant:

0 Minimum trip count is zero

1 Minimum trip count is one

If =c is omitted, minimum trip count is zero.

The DO loop control directive modifies the state of the DO loop switch. The DO loop switch is initially set according to the presence or absence of the ONE_TRIP_DO parameter on the FORTRAN command. A DO loop control directive overrides the ONE_TRIP_DO request.

The DO loop directive controls the minimum trip count for all loops that follow the directive, until either an END statement or another DO loop directive that resets the switch is encountered.

A DO loop control directive affects the interpretation of only those DO loops whose DO statements follow the directive in the same program unit.

I/O Implementation E

This appendix describes the structure of the files read and written by FORTRAN. All files read and written by FORTRAN input/output statements, as well as the files read and written by the FORTRAN compiler, are processed through internal file interface routines.

Indexed sequential files, which can be processed directly through a set of FORTRAN-callable subprograms, are described in chapter 6 under the heading File Interface Input/Output.

Runtime Input/Output

All input and output between a file referenced in a program and the external storage device is under control of the internal file interface routines, which encompass sequential, indexed sequential, and byte addressable file organizations.

Each NOS/VE file is described by an internally-maintained table of file attributes. File processing is governed by values placed in this table by the FORTRAN compiler. Certain of these values are permanent for the life of the file; others can be changed by a SET_FILE_ATTRIBUTE command, a CHANGE_FILE_ATTRIBUTE command, or by certain parameters in the PROGRAM and OPEN statements.

File and Record Definitions

A file is a collection of records. It is the largest collection of information that can be referenced by a file name. A file begins at beginning-of-information and ends at end-of-information. A record is a contiguous group of bytes within a file, which is read or written as a single unit. A record is read or written by:

One execution of an unformatted READ or WRITE statement.

A formatted, list directed, or namelist READ or WRITE statement. (A single execution of these statements can transmit more than one record.)

One call to READMS or WRITMS

One execution of a BUFFER IN or BUFFER OUT.

The record types are:

V Variable length

F Fixed length

U Undefined

FORTRAN uses the V and F record types.

File Structure

FORTRAN sets certain file attributes depending on the nature of the input/output operation and its associated file structure. Most attributes are permanent for the life of a file. After a file is created (that is, after the file is opened for the first time), the permanent attributes cannot be changed. The file attributes for the various types of FORTRAN input/output are shown in table E-1. The attributes which can be overridden by a SET_FILE_ATTRIBUTE (SETFA) command or CHANGE_FILE_ATTRIBUTE (CHAFA) command prior to file creation are indicated by an asterisk; those attributes which can be overridden prior to any open of the file are indicated by two asterisks. Files connected to $INPUT or $OUTPUT retain the attributes of $INPUT or $OUTPUT regardless of SETFA or CHAFA specifications.

Table E-1. Defaults for File Attributes

	Formatted Sequential I/O	Unformatted Sequential I/O	Buffer I/O	Mass Storage I/O	Direct Access I/O
MAXIMUM_RECORD_LENGTH	RECL=in OPEN statement	RECL=in OPEN statement†	n/a	n/a	RECL= in OPEN statement
OPEN_POSITION	$ASIS††	$ASIS††	$ASIS†	n/a	n/a
ACCESS_MODE	R/W/A/M††	R/W/A/M††	R/W/A/M††	R/W/A/M††	R/W/A/M††
FILE_ORGANIZATION	SQ	SQ	SQ	BA	BA
RECORD_TYPE	V†	V†	V†	U	F
PADDING_CHARACTER	n/a	n/a	n/a	n/a	blank†
PAGE_WIDTH	132 characters (nonconnected file)† 72 characters (connected file)†	n/a	n/a	n/a	n/a

†	Can be overridden by SETFA command prior to file creation
††	Can be overridden by SETFA command prior to any open
n/a	Not applicable to this mode of input/output
$ASIS	Current file position
R/W/A/M	READ/WRITE/APPEND/MODIFY
SQ	Sequential
BA	Byte addressable
V	Variable-Length
F	Fixed-Length
U	Undefined

SET_FILE_ATTRIBUTE Command

The SET_FILE_ATTRIBUTE command provides a means of overriding file attributes compiled into a program, and consequently, a means to change processing normally supplied for FORTRAN input/output. In particular, this command enables you to read or create a file with attributes that are different from those supplied by default.

The file attributes specified on a SET_FILE_ATTRIBUTE command are established when a file is created (that is, the first time it is opened).

The SET_FILE_ATTRIBUTE command has the form:

SET_FILE_ATTRIBUTE or SETFA

FILE=file
ACCESS_MODE=list of keyword value
FILE_CONTENTS=keyword value
FILE_CONTENTS=keyword value
FILE_ORGANIZATION=keyword value
FILE_STRUCTURE=keyword value
MAXIMUM_RECORD_LENGTH=integer
OPEN_POSITION=keyword value
PADDING_CHARACTER=character
PAGE_WIDTH=integer
RECORD_TYPE=keyword value

This format shows only those parameters which are applicable to the FORTRAN files described in this chapter. Refer to the SCL Command Interface manual for a complete description of the SET_FILE_ATTRIBUTE command.

Example:

```
PROGRAM ABC
OPEN (FILE='AFILE', UNIT=1)
WRITE (1,100) A, B, C
.
.
.
```

This program opens and writes a file named AFILE. The following SET_FILE_ATTRIBUTE command, specified before the program is executed, overrides the default maximum record length of 150 characters:

SET_FILE_ATTRIBUTE FILE=AFILE MAXIMUM_RECORD_LENGTH=100

A MAXIMUM_RECORD_LENGTH specification in a SETFA command prior to program execution takes precedence over a record length specification in an OPEN or PROGRAM statement. In the case of direct access files, if MAXIMUM_RECORD_LENGTH is specified in a SETFA command prior to execution, and if an OPEN statement specifies a different record length, a fatal error is issued.

Sequential Input/Output

The sequential READ and WRITE statements, namelist I/O statements, list directed I/O statements, and buffer I/O statements process sequential files with V type records. The record type can be overridden by a SET_FILE_ATTRIBUTE command before execution.

The BACKSPACE, REWIND, and ENDFILE operations are valid only for sequential files with V type records. BACKSPACE skips backward (toward beginning-of-information) one record. (The file is positioned before the record just read or written). REWIND positions a file at beginning-of-information. ENDFILE writes an end-of-partition boundary.

When an end-of-partition is encountered during a read, the ERR= specifier and EOF function return end-of-file status. If the end-of-partition does not coincide with end-of-information, you can continue reading the same file until the end-of-information is encountered.

For F and U type records, the EOF and UNIT functions return end-of-file status only at end-of-information.

Direct Access Input/Output

Direct access input/output statements process byte addressable files with F type records. F is the only record type permitted for direct access input/output.

The file positioning statements (BACKSPACE, REWIND, and ENDFILE) cannot be used with direct access files.

Compile-Time Input/Output

The FORTRAN compiler reads a source input file and produces up to three output files: a binary object file, an output listing file, and an error listing file. The compiler expects the input source file to have a certain structure, and it produces output files which have specific structures.

Table E-2 describes the attributes of the compiler input and output files.

Table E-2. File Structure

	Source Input File	Compiler Output Listing File	Error File	Binary Object File
FILE_ORGANIZATION	SQ	SQ	SQ	SQ
FILE_STRUCTURE	DATA†	DATA†	DATA†	DATA
FILE_CONTENTS	LEGIBLE†	LEGIBLE† (Interactive) LIST† (Batch)	LIST†	OBJECT
RECORD_TYPE	V†	V†	V†	V

† Can be overridden by SETFA command prior to file creation
SQ Sequential
V Variable-Length

Language Summary F

The following symbols are used in the descriptions of the FORTRAN 5 statements:

v	variable name, array name, or array element
sl	statement label
iv	integer variable
name	symbolic name
u	input/output unit specifier, which can be an integer expression with a value of 0 through 999, or a boolean expression whose value is a unit name in L format
fs	format specification
iolist	input/output list
ios	input/output status indicator
recn	record number

Other symbols are defined individually in the statement descriptions.

Assignment Page

v = arithmetic expression

boolean v = boolean expression (CDC Extension)

character v = character expression

logical v = logical or relational expression

Multiple Assignment (CDC Extension)

v=...*v*=**expression**

Type Declaration

INTEGER v,...,*v*

REAL v,...,*v*

DOUBLE PRECISION v,...,*v*

COMPLEX v,...,*v*

BOOLEAN v,...,*v* (CDC Extension)

LOGICAL v,...,*v*

CHARACTER **length* ,**v***length*,...,*v***length*

IMPLICIT type(ac,...,*ac*),...,*type(ac*,...,*ac)*

ac	A single letter, or range of letters represented by the first and last letter separated by a hyphen, indicating which variables are implicitly typed.

External Declaration

EXTERNAL name,...,*name*

Intrinsic Declaration

INTRINSIC name,...,*name*

Storage Allocation

type array(d),...,*array(d)*

DIMENSION array(d),...,*array(d)*

type Is INTEGER, CHARACTER, BOOLEAN, REAL, COMPLEX, DOUBLE PRECISION, or LOGICAL.

d Is one through seven array bound expressions separated by commas, as described in chapter 2.

COMMON */name/***nlist**,...,*/name/nlist*

nlist Is a list of variables or arrays, separated by commas, to be included in the common block.

DATA nlist/clist/,...,*nlist/clist/*

nlist Is a list of names to be initially defined. Each name in the list can take the form:

variable
array
element
substring
implied DO list

clist Is a list of constants or symbolic constants specifying the initial values. Forms for list items are described in section 2.

EQUIVALENCE **(nlist)**,...,*(nlist)*

nlist Is a list of variable names, array names, array element names, or character substring names. The names are separated by commas.

PARAMETER (name=exp,...,*name=exp*)

exp Is a constant or constant expression.

SAVE *name*,...,*name*

Flow Control

GO TO sl

GO TO (sl,...,*sl*)**expression**

GO TO iv,*(sl,...,sl)*

ASSIGN sl TO iv

IF (arithmetic or boolean expression) sl_1,sl_2,sl_3

IF (logical expression) statement

IF (logical expression) THEN

ELSE IF (logical expression) THEN

ELSE

END IF

DO sl,v=e_1,e_2,e_3

e_1,e_2,e_3	Are indexing parameters. They can be integer, real, double precision, or boolean constants, symbolic constants, variables, or expressions.

PAUSE *n*

STOP *n*

n	A string of 1 through 5 digits, or a character constant of 1 through 57 characters

END

Main Program

PROGRAM name *(upar,...,upar)*

upar	A unit declaration in one of the following forms: *unitname* *unitname=buffer-length* *unitname=/record-length* *unitname=buffer-length/record-length* *alternate-name=unitname* (buffer-length is disregarded under NOS/VE)

Subprogram

SUBROUTINE name *(argument,...,argument)*

type **FUNCTION name** (*argument,...,argument*)

type	Is BOOLEAN, CHARACTER, INTEGER, REAL, COMPLEX, DOUBLE PRECISION, or LOGICAL.

BLOCK DATA *name*

Statement Function

name (*argument,...,argument*)**=expression**

Subroutine Call

CALL name *(argument,...,argument)*

Function Reference

name (*argument,...,argument*)

Entry Point

ENTRY name *(argument,...,argument)*

Return

RETURN *expression*

Formatted Input/Output

READ (*UNIT*=**u**, *FMT*=**fs**, *IOSTAT*=*ios*, *ERR*=*sl*, *END*=*sl*) *iolist*

READ fs,*iolist*

WRITE (*UNIT*=**u**, *FMT*=**fs**, *IOSTAT*=*ios*, *ERR*=*sl*) *iolist*

PRINT fs, *iolist*

PUNCH fs, *iolist* (CDC Extension)

Unformatted Input/Output

READ (*UNIT*=**u**, *IOSTAT*=*ios*, *ERR*=*sl*, *END*=*sl*) *iolist*

WRITE (*UNIT*=**u**, *IOSTAT*=*ios*, *ERR*=*sl*) *iolist*

List Directed Input/Output

READ (*UNIT*=**u**, *FMT*=*, *IOSTAT*=*ios*, *ERR*=*sl*, *END*=*sl*) *iolist*

READ *, *iolist*

WRITE (*UNIT*=**u**, *FMT*=*, *IOSTAT*=*ios*, *ERR*=*sl*) *iolist*

PRINT *, *iolist*

PUNCH *, *iolist* (CDC Extension)

Direct Access Input/Output

READ (*UNIT*=**u**, *FMT*=**fs**, *IOSTAT*=*ios*, *ERR*=*sl*, **REC=recn**) *iolist*

WRITE (*UNIT*=**u**, *FMT*=**fs**, *IOSTAT*=*ios*, *ERR*=*sl*, **REC=recn**) *iolist*

Namelist Input/Output (CDC Extension)

NAMELIST /name/v,...,v ... /name/v,...,v

READ (*UNIT*=u, *FMT*=name, *IOSTAT*=*ios*, *ERR*=*sl*, *END*=*sl*)

READ name

WRITE (*UNIT*=u, *FMT*=name, *IOSTAT*=*ios*, *ERR*=*sl*)

PRINT name

PUNCH name

name A namelist group name.

Buffer Input/Output (CDC Extension)

BUFFER IN (u,p) (a,b)

BUFFER OUT (u,p) (a,b)

p Disregarded under NOS/VE.

a The first word of the data block to be transferred.

b The last word of the data block to be transferred.

Internal Data Transfer (CDC Extension)

ENCODE (c,fs,v) iolist

DECODE (c,fs,v) iolist

v The starting location of the record to be transferred.

c Specifies the number of characters to be transferred to or from each record.

Format Specification

sl FORMAT (flist)

flist Is a list of items, separated by commas, having the following forms:

red
ned
r(flist)

ed Is a repeatable edit descriptor.
ned Is a nonrepeatable edit descriptor.
r Is a nonzero unsigned integer constant repeat specification.

Edit Descriptors

***sr*Ew.d**	Single precision floating-point with exponent.
***sr*Ew.dEe**	Single precision floating-point with specified exponent length.
***sr*Fw.d**	Single precision floating-point without exponent.
***sr*Dw.d**	Double precision floating-point with exponent.
***sr*Gw.d**	Single precision floating-point with or without exponent.
***sr*Gw.dEe**	Single precision floating-point with or without specified exponent length.
***r*Iw**	Decimal integer.
***r*Iw.m**	Decimal integer with specified minimum number of digits.
rLw	Logical.
***r*A**	Character with variable length.
***r*Aw**	Character with specified length.
***r*Rw**	Rightmost characters with binary zero fill. (CDC Extension)
***r*Ow**	Octal. (CDC Extension)
***r*Ow.m**	Octal with minimum digits and leading zeros. (CDC Extension)
***r*Zw**	Hexadecimal. (CDC Extension)
***r*Zw.m**	Hexadecimal with minimum digits and leading zeros. (CDC Extension)

s	optior al scale factor of the form kP.
r	optional repetition factor.
w	integer constant indicating field width.
d	integer constant indicating digits to right of decimal point.
e	integer constant indicating digits in exponent field.
m	integer constant indicating minimum number of digits in field.
n	positive nonzero decimal digit.
k	integer constant called a scale factor.

kP	Changes the position of a decimal point of an input or output real number.
BN	Blanks ignored on numeric input.
BZ	Blanks treated as zeros on numeric input.
SP	+ characters produced on output.
SS	+ characters suppressed on output.
S	+ characters suppressed on output.
nX	Skip n spaces.
Tn	Tabulate to n^{th} column.
TRn	Tabulate forward.
TLn	Tabulate backward.

nH	Boolean or character string output.
"..."	Boolean or character string output. (CDC Extension)
'...'	Character string output.
:	Format control.
/	End of FORTRAN record.

File Positioning

BACKSPACE (*UNIT=***u**, *IOSTAT=ios, ERR=sl*)

BACKSPACE u

REWIND (*UNIT=***u**, *IOSTAT=ios, ERR=sl*)

REWIND u

ENDFILE (*UNIT=***u**, *IOSTAT=ios, ERR=sl*)

ENDFILE u

File Status

OPEN (*UNIT=***u**, *IOSTAT=ios, ERR=sl, FILE=fin, STATUS=sta,*
ACCESS=acc, FORM=fm, RECL=rl, BLANK=blnk, BUFL=bl)

INQUIRE (unit-or-file, *IOSTAT=ios, ERR=sl*
EXIST=ex, OPENED=od, NUMBER=num, NAMED=nmd,
NAME=fn, ACCESS=acc, SEQUENTIAL=seq, DIRECT=dir,
FORM=fm, FORMATTED=FMT, UNFORMATTED=unf, RECL=fcl,
NEXTREC=nr, BLANK=blnk)

where unit-or-file is one of the following:

UNIT=u
FILE=filename

CLOSE (*UNIT=***u**, *IOSTAT=ios, ERR=sl, STATUS=sta*)

CDC Extensions to Standard FORTRAN G

Following is a list of the FORTRAN features that constitute extensions and additions to ANSI FORTRAN.

Basic Concepts

Symbolic names can be up to seven characters in length (ANSI allows only six).

The " (quote) character has been added to the FORTRAN character set.

Sequenced format for FORTRAN statements has been added.

C$ (compiler control) directives have been added.

A boolean data type has been added.

Constants

Boolean constants have been added (the boolean constants are boolean string, octal, and hexadecimal).

Extended Hollerith constants have been added.

Symbolic constants can appear as the real and imaginary parts of a complex constant.

Arrays and Substrings

Intrinsic function references and boolean constants can appear in dimension bound expressions.

Subscript or substring expressions can be real, double precision, complex, or boolean expressions, as well as integer.

Expressions

Boolean expressions have been added.

Boolean expressions can appear in arithmetic expressions.

Double precision and complex operands can be combined using the +, –, *, and / operators.

A double precision operand can be raised to a complex power.

Boolean entities can appear in relational expressions.

An .XOR. operator has been added.

Constant expressions can include intrinsic function references with constant expressions as arguments.

Specification Statements

A BOOLEAN type statement has been added.

Entities in named (labeled) common can be initialized by a DATA statement in any program unit.

The IMPLICIT statement can declare a boolean type.

The PARAMETER statement can declare boolean symbolic constants.

A variable can be specified as integer following its appearance in a dimension bound expression.

A symbolic constant can appear as the real or imaginary part of a complex constant.

DATA Statement

A replicated value list can appear in a DATA statement.

Boolean entities can appear in a DATA statement.

Assignment Statement

Assignment statements can be type boolean.

A multiple assignment statement has been added.

Flow Control Statements

Real, double precision, complex, or boolean expressions are valid in a computed GO TO statement.

A boolean expression can be used in an arithmetic IF statement.

A boolean expression can be used as an indexing parameter in a DO loop or implied DO list.

A one-trip DO loop option has been added for increased compilation speed.

An extended range capability for DO loops has been added.

Input/Output

The following input/output statements have been added:

NAMELIST
BUFFER IN and BUFFER OUT
ENCODE and DECODE
PUNCH
OPENMS, READMS, WRITMS, CLOSMS, STINDX

A record length specifier can appear in an OPEN statement for a file accessed sequentially.

More than one unit can be associated with a single external file.

Random access files have been added.

Extended internal files have been added.

An external unit identifier can be type boolean.

A buffer length specifier can appear in an OPEN statement (this specifier is disregarded in CDC FORTRAN).

A comma can optionally follow the output list of a list directed output statement.

An implicit file/unit association occurs in the absence of PROGRAM statement or OPEN statement declaration.

The following edit descriptors have been added:

Quoted string (" ... ")
Rw (character)
Ow and Ow.m (octal)
Zw and Zw.m (hexadecimal)

NAMELIST formatting has been added.

A and Aw descriptors can be used for noncharacter data.

A format specification can be contained in a noncharacter array.

A FORM='BUFFERED' option has been added to the OPEN and INQUIRE statements.

PROGRAM Statement

Symbolic unit specifiers can be declared on the PROGRAM statement.

Buffer length (disregarded by FORTRAN) and record length can be declared on the PROGRAM statement.

In a statement function reference, the actual argument is converted to the type of the dummy argument.

A substring reference can appear in the expression of a statement function statement.

External Procedures

Boolean arguments can be associated with integer or real arguments.

An external procedure name can be the same as a common block name.

The name of a block data subprogram can be the same as a common block name.

A RETURN statement can appear in a main program (it has the same effect as an END statement).

The expression in the alternate return form of the RETURN statement can be any arithmetic or boolean expression. (ANSI allows only integer expressions.)

Extended Hollerith constants can be passed as actual arguments.

Intrinsic Functions

The following intrinsic functions have been added:

BOOL

boolean operations (AND, OR, XOR, NEQV, EQV, COMPL)

mathematical (ERF, ERFC, ATANH, SIND, COSD, TAND)

miscellaneous (SHIFT, MASK, RANF)

The type conversion functions INT, REAL, DBLE, and CMPLX can have boolean arguments.

Association of Entities

Partial association can exist between a boolean entity and a double precision entity.

Association can exist between a boolean entity and an integer or real entity.

FORTRAN-Callable Subprograms

The following subprograms can be called from a FORTRAN program:

Indexed sequential input/output subprograms

Sort/Merge subprograms

System Command Language subprograms

Utility subprograms

Input/Output status checking subprograms

Miscellaneous input/output subprograms

Debugging subprograms

Collating sequence control subprograms

Selecting Collation Tables for Indexed Sequential Files H

One of the key types available for use with indexed sequential files (as described in chapter 6) is collated keys. The order in which collated keys are sorted is determined by a collation table. If you specify this key type, you must supply an explicit collation table; there is no system-supplied default collation table. A fatal error occurs if the KEY_TYPE attribute for a file is collated and the file is opened without a collation table supplied.

Do not confuse the collation table required for collated keys with the collation table used for comparing characters as described under Collating Sequence Control in chapter 9. The system supplies a default collation table for character comparison; it does not supply a default collation table for indexed sequential files.

You can specify a collation table by name using the CTN keyword or by address with the DCT keyword. You specify the CTN or DCT keyword on a CALL STOREF statement before the file is opened for its creation run. NOS/VE supplies eleven predefined collation tables; you can specify a predefined collation table or a collation table that you have created.

Predefined Collation Tables

You specify a predefined collation table by specifying its name on a CALL STOREF statement with either the CTN or DCT keyword. For example, the following statement specifies OSV$ASCII6_FOLDED as the collation table name in ISFIT.

```
CALL STOREF(ISFIT,'CTN','OSV$ASCII6_FOLDED')
```

A collation table name should be entered using only uppercase letters.

The collating sequences of the predefined collation tables are listed in appendix B

Creating Your Own Collation Table

The easiest way to create your own collation table within a FORTRAN program is to use the subprograms described under Collating Sequence Control in chapter 9. These subprograms create a collation table from a string of characters.

NOTE

For these subprograms to be effective, you must include a $C COLLATE(USER) directive in your program or specify DEFAULT_COLLATION=USER on the FORTRAN command that compiles the program.

To create a collation table, you assign a character to each position within a 256-character string. The characters assigned must comprise the ASCII character set listed in appendix B. Nonprintable characters are indirectly assigned to their position in the string using the CHAR function. (Before using the CHAR function, specify ASCII on a COLSEQ call as shown in the collation table example.)

The order in which you assign characters to the string is the order in which you want the characters collated. For example, to collate in reverse order, you would assign the characters in reverse order from the order in which the characters are listed in appendix B. After assigning 256 characters to the string, you call the CSOWN subprogram described in section 9 to define the string as the user-specified collating sequence.

The user-specified collating sequence within a FORTRAN program is referenced by the name FTV$USER_COLLATE_TABLE. Therefore, to assign the collating sequence you defined to the CTN file attribute, you specify FTV$USER_COLLATE_TABLE as the collation table name on the STOREF call. For example, the following statement specifies the CTN value for ISFIT.

```
CALL STOREF (ISFIT,'CTN','FTV$USER_COLLATE_TABLE')
```

NOTE

The name FTV$USER_COLLATE_TABLE must be in uppercase letters because it must match a corresponding internal entry point in the FORTRAN run-time routine that handles collation control.

The CALL STOREF statement must appear before the file is first opened for its creation run. When the CALL OPENM statement opens the file, the value of the CTN attribute becomes permanent. Subsequent jobs that read or update the file cannot change the collation table stored with the file. A CALL STOREF statement that attempts to change the collation table is diagnosed as an error.

Collation Table Example

The program in figure H-1 creates and uses a collation table. Note the placement and use of the C$ COLLATE, CALL STOREF, and CALL TABLE statements.

Output from the program is shown in figure H-2. The first part of the output prints each record and key as records are written to the file. After the records are written to the file, the file is closed, opened, and read sequentially. The second part of the output shows the result of the sequential read. The records are in order according to the collating sequence defined in the collation table.

```
      Program CTABLE

C  ***************************************************
C  *  This program creates an indexed sequential file  *
C  *  (IS_FILE) from a sequential file (DATA_FILE).    *
C  *  Also, this program shows how to set up and use a *
C  *  collation table through a FORTRAN program.       *
C  ***************************************************

C  Issue directive telling the compiler that the collation table
C  is user specified.

C$    Collate (user)

C  Declare variables.

      Integer isfit, reclg, stat
      Common iswsa
      Character * 65 iswsa

C  Call subroutine to create the collation table.

      Call Table

C  Set file attributes before opening IS_FILE.

      Call Fileis (isfit,'lfn','IS_FILE',
     +           'mrl',65,'rt','f',
     +           'kl',20,'kt','s','rkp',0,'emk','yes',
     +           'ip',10,'dp',15,'erc',30,
     +           'dfc',3)

C  Store the name of the collate table in CTN attribute.

      Call Storef (isfit,'ctn','FTV$USER_COLLATE_TABLE')
```

(Continued on next page)

Figure H-1. Creation Program

— (Continued from previous page)

```
C  Open DATA_FILE and IS_FILE.  Check for error on IS_FILE open.

      Open (2,file='DATA_FILE')
      Call Openm (isfit,'NEW')
      stat = Ifetch (isfit,'es')
      If (stat.ne.0) go to 90

C  Read each record from DATA_FILE into the working storage area (iswsa),
C  and then put record into IS_FILE.  After put, check for error.  If no
C  error occured, print the record.

   10 Continue
        Read (2,'(A65)',End=30) iswsa
        Call Put (isfit,iswsa)
        Call Ifetch (isfit,'ES',stat)
        If (stat.ne.0) go to 90
        Print '(1X,A65)', iswsa
        Go to 10



C  When all records in DATA_FILE have been read, close IS_FILE and
C  check whether error occurred during CLOSE.

   30 Continue
      Call Closem (isfit)
      stat = Ifetch (isfit,'ES')
      If (stat.eq.0) Go to 40
      Print 900, stat
      Stop

C  Now read IS_FILE.

   40 Continue
      Call Openm (isfit,'input')
      Call Storef (isfit,'WSA',iswsa)
      Call Storef (isfit,'WSL',80)
   50 Continue
        Call Getn (isfit)
        stat = Ifetch (isfit,'ES')
        If (stat.ne.0) Go to 90
        filpos = Ifetch (isfit,'FP')
        If (filpos.eq.64) Go to 70
        Print '('' Record = '',A65)',iswsa
        Go to 50

C  Close IS_FILE and stop.

   70 Continue
      Call Closem (isfit)
      stat= Ifetch (isfit,'ES')
      If (stat.ne.0) Print '(1X,I6)', stat
      Stop
```

— (Continued on next page)—

Figure H-1. Creation Program

— (Continued from previous page)

```
C  If error occurs during OPEN or PUT, control transfers to this point
C  in program.  The error number is printed and the file is closed.
   90 Continue
      Print 900, stat
      Call Closem (isfit)
      Stop
  900 Format (1X,I6)
      End
C   The following section contains subroutine TABLE.
      Subroutine Table
C  ************************************************************************
C  * This subroutine sets up the collation table.  The user MUST set up *
C  * a collation table if collated keys are specified.                   *
C  ************************************************************************
C$    collate (user)
      Character user*256
C  The following section puts all ascii characters into a string structure.
C  Symbols and numbers on the far right show the ascii graphics (or their
C  abbreviations) and corresponding decimal representations.  Nonprintable
C  characters have to be indirectly assigned into the string by referencing
C  the CHAR function (which MUST be operating in ASCII mode by COLSEQ).
      Call colseq ('ascii')
      User(1:1) = '$'                                          $   36
      User(2:2) = 'R'                                          R   65
      User(3:3) = char(9)                                     HT    9
      User(4:4) = '<'                                          <   60
      User(5:5) = 'F'                                          F   82
      User(6:6) = 'd'                                          d  100
      User(7:7) = char(127)                                   RO  127
      User(8:8) = '+'                                          +   43
      User(9:9) = '%'                                          %   37
      User(10:10) = 'x'                                        x  129
      User(11:11) = char(13)                                  CR   13
      User(12:12) = '3'                                        3   51
      User(13:13) = '#'                                        #   35
      User(14:14) = char(26)                                 CUP   26
      User(15:15) = 'V'                                        V   86
      User(16:16) = 'm'                                        m  109
      User(17:17) = '0'                                        0   48
      User(18:18) = char(18)                                 DC2   18
      User(19:19) = ':'                                        :   58
      User(20:20) = '^'                                        ^   94
      User(21:21) = 'r'                                        r  114
      User(22:22) = char(21)                                SKIP   21
      User(23:23) = '*'                                        *   42
      User(24:24) = 'K'                                        K   75
      User(25:25) = '{'                                        {  123
```

(Continued on next page)—

Figure H-1. Creation Program

— (Continued from previous page)

```
User(26:26) = 'c'                                    c   99
User(27:27) = '6'                                    6   54
User(28:28) = 'w'                                    w  119
User(29:29) = 'P'                                    P   80
User(30:30) = char(16)                             DLE   16
User(31:31) = '_'                                    _   95
User(32:32) = 'A'                                    A   70
User(33:33) = char(3)                              ETX    3
User(34:34) = 'h'                                    h  104
User(35:35) = 'H'                                    H   72
User(36:36) = 'D'                                    D   68
User(37:37) = char(2)                              STX    2
User(38:38) = 'M'                                    M   77
User(39:39) = char(29)                              GS   29
User(40:40) = ','                                    ,   44
User(41:41) = 'a'                                    a   97
User(42:42) = '~'                                    ~  126
User(43:43) = 'k'                                    k  107
User(44:44) = '1'                                    1   49
User(45:45) = ';'                                    ;   59
User(46:46) = char(23)                             ETB   23
User(47:47) = ' '                                        92
User(48:48) = 'u'                                    u  117
User(49:49) = '5'                                    5   53
User(50:50) = 'B'                                    B   66
User(51:51) = 'f'                                    f  102
User(52:52) = char(5)                              ENQ    5
User(53:53) = '('                                    (   40
User(54:54) = '7'                                    7   55
User(55:55) = '&'                                    &   38
User(56:56) = 'T'                                    T   84
User(57:57) = 'b'                                    b   98
User(58:58) = 'Z'                                    Z   90
User(59:59) = 'o'                                    o  111
User(60:60) = ' '                                        32
User(61:61) = char(27)                             ESC   27
User(62:62) = char(7)                              BEL    7
User(63:63) = ']'                                    ]   93
User(64:64) = 'X'                                    X   88
User(65:65) = '2'                                    2   50
User(66:66) = char(31)                              US   31
User(67:67) = 'N'                                    N   78
User(68:68) = '`'                                    `   96
User(69:69) = 'q'                                    q  113
User(70:70) = char(11)                              VT   11
User(71:71) = ')'                                    )   41
User(72:72) = 'J'                                    J   74
User(73:73) = '}'                                    }  125
User(74:74) = char(19)                             DC3   19
User(75:75) = 'z'                                    z  131
User(76:76) = 's'                                    s  115
User(77:77) = 'Q'                                    Q   81
```

(Continued on next page)—

Figure H-1. Creation Program

— (Continued from previous page)

```
User(78:78) = '?'                                    ?   63
User(79:79) = '9'                                    9   57
User(80:80) = char(12)                              FF   12
User(81:81) = 'e'                                    e  101
User(82:82) = 'G'                                    G   71
User(83:83) = '='                                    =   61
User(84:84) = char(6)                               ACK   6
User(85:85) = 'U'                                    U   85
User(86:86) = ''''                                   '   39
User(87:87) = '@'                                    @   64
User(88:88) = 'E'                                    E   69
User(89:89) = '4'                                    4   52
User(90:90) = '"'                                    "   34
User(91:91) = char(30)                              RS   30
User(92:92) = char(4)                              EOT    4
User(93:93) = 'g'                                    g  103
User(94:94) = '/'                                    /   47
User(95:95) = char(0)                              NUL    0
User(96:96) = '-'                                    -   45
User(97:97) = 'I'                                    I   73
User(98:98) = '['                                    [   91
User(99:99) = '| '                                   |  124
User(100:100) = char(28)                            FS   28
User(101:101) = 'j'                                  j  106
User(102:102) = 'v'                                  v  118
User(103:103) = char(1)                            SOH    1
User(104:104) = '.'                                  .   46
User(105:105) = 'y'                                  y  130
User(106:106) = char(8)                             BS    8
User(107:107) = '>'                                  >   62
User(108:108) = 'W'                                  W   67
User(109:109) = 'i'                                  i  105
User(110:110) = 'S'                                  S   83
User(111:111) = '!'                                  !   33
User(112:112) = char(24)                           CLR   24
User(113:113) = 'l'                                  l  108
User(114:114) = 'L'                                  L   76
User(115:115) = 't'                                  t  116
User(116:116) = char(22)                          LCLR   22
User(117:117) = 'n'                                  n  110
User(118:118) = 'O'                                  O   79
User(119:119) = char(17)                           DC1   17
User(120:120) = char(25)                          RSET   25
User(121:121) = 'C'                                  C   87
User(122:122) = char(15)                            SI   15
User(123:123) = 'Y'                                  Y   89
User(124:124) = char(10)                            LF   10
User(125:125) = 'p'                                  p  112
User(126:126) = char(14)                            SO   14
User(127:127) = char(20)                           DC4   20
User(128:128) = '8'                                  8   56
```

Figure H-1. Creation Program

```
C     Complete the table using a loop that assigns the leftover characters
C     in reverse order filling the remaining 128 positions.

      Do 5 I = 129,256
   5  User (I:I) = char (256-I+128)
      Call Csown (user)
      Return
      End
Algeria              19709000    919591 Algiers       Africa
Australia            14796000   2967895 Canberra      Australia
Austria              74760000     32374 Vienna        Europe
Belguim               9875000     11781 Brussels      Europe
Canada               24336000   3851791 Ottawa        NAmerica
China              1053788000   3705390 Beijing       Asia
Denmark               5157000     16629 Copenhagen    Europe
England              55717000     94226 London        Europe
France               53844000    211207 Paris         Europe
India               700734000   1269340 New Delhi     Asia
Ireland              3349000     27136 Dublin        Europe
Italy                57513000    116303 Rome          Europe
Ivory Coast           8513000    124503 Abidjan       Africa
Japan                11878300    143750 Tokyo         Asia
Mexico               70143000    761601 Mexico City   SAmerica
Spain                38686000    194897 Madrid        Europe
Sweden               8335000     173731 Stockholm     Europe
Switzerland          63000000     15941 Bern          Europe
Tanzania             18744000    364898 Zanzibar      Africa
Turkey               47284000    301381 Ankara        Asia
USSR                269302000   8649498 Moscow        Asia
United States       225195000   3615105 Washington    NAmerica
Venezuela            15771000    352143 Caracas       SAmerica
West Germany         60948000     95976 Bonn          Europe
--  File IS_FILE :  0 DELETE_KEYs done since last open.
--  File IS_FILE :  0 GET_KEYs done since last open.
--  File IS_FILE :  0 GET_NEXT_KEYs done since last open.
--  File IS_FILE :  24 PUT_KEYs (and PUTREPs->put) since last open.
--  File IS_FILE :  0 PUTREPs done since last open.
--  File IS_FILE :  0 REPLACE_KEYs (and PUTREPs->replace) since last open.
Record = France             53844000    211207 Paris         Europe
Record = Venezuela          15771000    352143 Caracas       SAmerica
Record = Australia          14796000   2967895 Canberra      Australia
Record = Austria            74760000     32374 Vienna        Europe
Record = Algeria            19709000    919591 Algiers       Africa
Record = Denmark             5157000     16629 Copenhagen    Europe
Record = Mexico             70143000    761601 Mexico City   SAmerica
Record = Belguim             9875000     11781 Brussels      Europe
```

(Continued on next page)

Figure H-2. Output From Program

— (Continued from previous page)

```
Record = Tanzania              18744000    364898 Zanzibar      Africa
Record = Turkey                47284000    301381 Ankara        Asia
Record = Japan                 11878300    143750 Tokyo         Asia
Record = USSR                 269302000   8649498 Moscow        Asia
Record = United States        225195000   3615105 Washington    NAmerica
Record = England               55717000     94226 London        Europe
Record = Ireland                3349000     27136 Dublin        Europe
Record = Ivory Coast            8513000    124503 Abidjan       Africa
Record = Italy                 57513000    116303 Rome          Europe
Record = India                700734000   1269340 New Delhi     Asia
Record = West Germany          60948000     95976 Bonn          Europe
Record = Sweden                 8335000    173731 Stockholm     Europe
Record = Switzerland           63000000     15941 Bern          Europe
Record = Spain                 38686000    194897 Madrid        Europe
Record = China               1053788000   3705390 Beijing       Asia
Record = Canada                24336000   3851791 Ottawa        NAmerica
-- File IS_FILE : AMP$GET_NEXT_KEY has reached a file boundary : EOI.
-- File IS_FILE :  0 DELETE_KEYs done since last open.
-- File IS_FILE :  0 GET_KEYs done since last open.
-- File IS_FILE :  24 GET_NEXT_KEYs done since last open.
-- File IS_FILE :  0 PUT_KEYs (and PUTREPs->put) since last open.
-- File IS_FILE :  0 PUTREPs done since last open.
-- File IS_FILE :  0 REPLACE_KEYs (and PUTREPs->replace) since last open.
```

Figure H-2. Output From Program

Creating a Collation Weight Table

It is also possible to create a collation table to be specified by address using the DCT keyword. The collation table must be in the form of a collation weight table. (The CSOWN subprogram generates a collation weight table from a character string.)

A collation weight table is 256 contiguous bytes (32 words) with each byte containing an integer value. The 256 bytes within the table correspond to the 256 character codes in the ASCII character set. The collation weight, or ordinal, for each character is the value stored in the byte corresponding to the character within the table.

Figure H-3 illustrates the collation weight table for the ASCII collation sequence with the weights in hexadecimal. Weights are assigned in ascending order just as the characters are ordered in the set. The character codes from 80 through FF hexadecimal do not have graphic characters associated with them. However, each character code is assigned a collating weight within the table.

As illustrated, the weights for the uppercase letters are in bytes 41 through 5A hexadecimal of the string and the weights for the lowercase letters are in bytes 61 through 7A.

Suppose you want the lowercase letters to be collated the same as the uppercase letters (case insensitive). You would then assign the collating weight of each uppercase letter to the corresponding lowercase letter. The following is a listing of words 12 through 15 of the collation weight table showing the changed values for the lowercase letters.

`	a	b	c	d	e	f	g
60	41	42	43	44	45	46	47
h	i	j	k	l	m	n	o
48	49	4A	4B	4C	4D	4E	4F
p	q	r	s	t	u	v	w
50	51	52	53	54	55	56	57
x	y	z	{	|	}	~	DEL
58	59	5A	7B	7C	7D	7E	7F

To create a collation table, you declare a 32-word integer array and then assign a hexadecimal constant to each word in the array. For example, the following statement declares an array named TABLE with bounds 0 and 31.

```
INTEGER TABLE (0:31)
```

You then assign a hexadecimal constant to each of the 32 words in the array. For example, when creating a case insensitive collation table, you would assign the following hexadecimal constants to words 12 through 15 of the array.

```
TABLE(12) = Z"6041424344454647"

TABLE(13) = Z"48494A4B4C4D4E4F"

TABLE(14) = Z"5051525354555657"

TABLE(15) = Z"58595A7B7C7D7E7F"
```

Word									Word								
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	16								
	0	1	2	3	4	5	6	7		80	81	82	83	84	85	86	87
1	BS	HT	LF	VT	FF	CR	SO	SI	17								
	8	9	A	B	C	D	E	F		88	89	8A	8B	8C	8D	8E	8F
2	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	18								
	10	11	12	13	14	15	16	17		90	91	92	93	94	95	96	97
3	CAN	EM	SUB	ESC	FS	GS	RS	US	19								
	18	19	1A	1B	1C	1D	1E	1F		98	99	9A	9B	9C	9D	9E	9F
4	SP	!	"	#	$	%	&	`	20								
	20	21	22	23	24	25	26	27		A0	A1	A2	A3	A4	A5	A6	A7
5	(	)	*	+	,	-	.	/	21								
	28	29	2A	2B	2C	2D	2E	2F		A8	A9	AA	AB	AC	AD	AE	AF
6	0	1	2	3	4	5	6	7	22								
	30	31	32	33	34	35	36	37		B0	B1	B2	B3	B4	B5	B6	B7
7	8	9	:	;	<	=	>	?	23								
	38	39	3A	3B	3C	3D	3E	3F		B8	B9	BA	BB	BC	BD	BE	BF
8	@	A	B	C	D	E	F	G	24								
	40	41	42	43	44	45	46	47		C0	C1	C2	C3	C4	C5	C6	C7
9	H	I	J	K	L	M	N	O	25								
	48	49	4A	4B	4C	4D	4E	4F		C8	C9	CA	CB	CC	CD	CE	CF
10	P	Q	R	S	T	U	V	W	26								
	50	51	52	53	54	55	56	57		D0	D1	D2	D3	D4	D5	D6	D7
11	X	Y	Z	[	\	]	^	_	27								
	58	59	5A	5B	5C	5D	5E	5F		D8	D9	DA	DB	DC	DD	DE	DF
12	.	a	b	c	d	e	f	g	28								
	60	61	62	63	64	65	66	67		E0	E1	E2	E3	E4	E5	E6	E7
13	h	i	j	k	l	m	n	o	29								
	68	69	6A	6B	6C	6D	6E	6F		E8	E9	EA	EB	EC	ED	EE	EF
14	p	q	r	s	t	u	v	w	30								
	70	71	72	73	74	75	76	77		F0	F1	F2	F3	F4	F5	F6	F7
15	x	y	z	{	|	}	~	DEL	31								
	78	79	7A	7B	7C	7D	7E	7F		F8	F9	FA	FB	FC	FD	FE	FF

Figure H-3. Collation Weight Table

Index

A

A edit descriptor 6-17
A edit descriptor for noncharacter list items 6-18
ABORT subprogram 9-16
ABS intrinsic function 8-6
ACOS intrinsic function 8-6
Actual argument 7-9
Adjustable dimensions 7-13
AIMAG intrinsic function 8-6
AINT intrinsic function 8-6
ALOG intrinsic function 8-6
ALOG10 intrinsic function 8-6
Alternate return 7-16
AMAX0 intrinsic function 8-7
AMAX1 intrinsic function 8-7
AMIN0 intrinsic function 8-7
AMIN1 intrinsic function 8-7
AMOD intrinsic function 8-7
AND intrinsic function 8-7
ANINT intrinsic function 8-7
Apostrophe edit descriptor 6-31
Argument association
  Arrays as arguments 7-11
  Asterisks as arguments 7-12
  Characters as arguments 7-12
  Procedure names as arguments 7-12
  Variables as arguments 7-11
Arguments 7-9
Arithmetic
  Assignment statement 4-15
  Expressions 4-1
  IF statement 5-3
  Operators 4-2
Array
  Assumed-size 7-14
  Dimensions 2-14
  References 2-16
  Storage 2-16
Arrays 2-14
Arrays in namelist 6-43
ASCII character set and collating weight tables B-1, B-2
ASCII6 collating sequence 9-26
ASIN intrinsic function 8-7
ASSIGN statement 5-1, 5-2, F-3
Assigned GO TO statement 5-1, 5-2, F-2
Assignment statements
  Arithmetic 4-1, 4-15
  Boolean 4-1, 4-17
  Character 4-1, 4-16
  Logical 4-1, 4-16
  Multiple 4-1, 4-17
  Syntax summary F-1
Assumed-length character strings 7-14
ATAN intrinsic function 8-7
ATANH intrinsic function 8-7
ATAN2 intrinsic function 8-8

B

BACKSPACE statement 6-116, F-7
Beginning-of-Information (BOI), definition A-1
Blank common, definition A-1
Block
  Definition of A-1
  Indexed sequential file 6-59
BLOCK DATA statement F-3
Block data subprogram 7-6
Block IF statement 5-3, 5-4
Block IF statement structures 5-3, 5-5
BN edit descriptor 6-30
BOI (See Beginning-of-Information)
BOOL intrinsic function 8-8
Boolean
  Assignment statement 4-17
  Constant 2-9
  Data type 1-1
  Expressions 4-12
  Operators 4-14
  Variable 2-13
Boolean string constant 2-9
BOOLEAN type statement 3-1, 3-4, F-1
BUFFER IN statement 6-44, F-5
BUFFER OUT statement 6-45, F-5
Buffer statement, definition A-1
Byte, definition A-1
Byte addressable file, definition A-1
BZ edit descriptor 6-30

C

CABS intrinsic function 8-8
CALL statement 5-14, 7-8, F-4
Calling sequence, definition A-1
Calling Sort/Merge 9-1
CCOS intrinsic function 8-8
CDC extensions to standard FORTRAN 1-1, G-1
CEXP intrinsic function 8-8
CHAR intrinsic function 8-8
Character
  Assignment statement 4-16
  Constant 2-11
  Data initialization 3-21
  Definition of A-1

Expressions 4-7
Format specification 6-12
Operators 4-7
Subtrings 2-18
Variable 2-13
CHARACTER type statement 3-1, 3-5, F-1
CLOCK utility subprogram 9-28
CLOG intrinsic function 8-8
Close, definition A-1
CLOSE statement 6-103, F-7
CLOSEM call statement 6-75
CLOSMS call statement 6-48
CMPLX intrinsic function 8-9
COBOL subprograms 7-19
COBOL6 collating sequence 9-27
Collated key, definition A-2
Collating sequence
Control for indexed sequential files H-1
Control subprograms 9-26
Definition of A-2
Collation control directives D-4
Colon edit descriptor 6-34
COLSEQ utility subprogram 9-27
Command interface subprogram (SCLCMD) 9-20
Comment lines
Nonsequenced mode 2-2
Sequenced mode 2-3
Common and equivalence maps 10-17
Common blocks 7-13, A-2
COMMON statement 3-1, 3-10, F-2
Compilation
Statistics 10-12
Time, definition A-2
Compilation and execution 10-1
Compile Command
Examples 10-12
FORTRAN 10-1
Parameter formats 10-3
Parameter names 10-4
Parameter options 10-4
Parameters Inside front cover, 10-1
Compiler
Options 10-5
Output listing 10-12
Reference map 10-13
Source listing 10-12
Statistics 10-12
Compiler directive lines
Nonsequenced mode 2-2
Sequenced mode 2-3
Compile-time input/output E-4
COMPL intrinsic function 8-9
Complex
Constant 2-8
Variable 2-13
COMPLEX type statement 3-1, 3-4, F-1
Computed GO TO statement 5-1, F-2
Conditional compilation D-2
CONJG intrinsic function 8-9
CONNEC routine 6-120
Constants
Boolean 2-9
Boolean string 2-9
Character 2-11
Complex 2-8
Double precision 2-7
Extended Hollerith 2-10
Hexadecimal 2-11
Integer 2-5
Logical 2-8
Octal 2-11
Real 2-6
Continuation lines
Nonsequenced mode 2-2
Sequenced mode 2-3
CONTINUE statement 5-13
COS intrinsic function 8-9
COSD intrinsic function 8-9
COSH intrinsic function 8-9
Creating a collation table J-1
Creating an indexed sequential file 6-65
CREV subprogram 9-17
CSIN intrinsic function 8-9
CSOWN utility subprogram 9-27
CSQRT intrinsic function 8-9
CYBIL Procedures 7-17
C$ directives 1-1, 2-3, 2-5, D-1
C$ PARAM directive 9-11

D

D edit descriptor 6-19
DABS intrinsic function 8-9
DACOS intrinsic function 8-10
DASIN intrinsic function 8-10
Data block, definition 6-60, A-2
DATA statement 3-1, 3-18, F-2
DATAN intrinsic function 8-10
DATANH intrinsic function 8-10
DATAN2 intrinsic function 8-10
DATE utility subprogram 9-28
DBLE intrinsic function 8-10
DCOS intrinsic function 8-10
DCOSH intrinsic function 8-10
DDIM intrinsic function 8-10
Debugging subprograms 9-22
DECODE statement 6-58, F-5
Default, definition A-2
Default data type, definition A-2
DELV subprogram 9-17
DEXP intrinsic function 8-11
Differences between NOS/VE FORTRAN and NOS FORTRAN 5 C-1
DIM intrinsic function 8-11
DIMENSION statement 3-1, 3-8, F-2

DINT intrinsic function 8-11
Direct access
  Files 6-53
  Input/output E-3, A-2
  READ statement 6-53, F-5
  WRITE statement 6-53, F-5
DISCON routine 6-121
DISPLA utility subprogram 9-29
DISPLAY collating sequence 9-29
DLOG intrinsic function 8-11
DLOG10 intrinsic function 8-11
DLTE call statement 6-76
DMAX1 intrinsic function 8-11
DMIN1 intrinsic function 8-11
DMOD intrinsic function 8-11
DNINT intrinsic function 8-11
DO
  Extended range 5-13
  Loop 5-9
  Loop control D-4
  Loops map 10-22
  Nested 5-10
  Syntax summary F-3
DO list, implied 3-20
DO statement 5-8, F-3
Double precision
  Constant 2-7
  Variable 2-12
DOUBLE PRECISION type statement 3-1, 3-3, F-1
DPROD intrinsic function 8-11
DSIGN intrinsic function 8-12
DSIN intrinsic function 8-12
DSINH intrinsic function 8-12
DSQRT intrinsic function 8-12
DTAN intrinsic function 8-12
DTANH intrinsic function 8-12
Dummy argument 7-10
DUMP utility subprogram 9-23

E

E edit descriptor 6-19
Edit descriptors
  Nonrepeatable 6-29, F-6
  Repeatable 6-14, F-6
ELSE IF statement 5-3, 5-5, F-3
ELSE statement 5-3, 5-5
Embedded key, definition A-2
ENCODE statement 6-61, F-5
END IF statement 5-3, 5-5, F-3
END statement 5-14, F-3
END= specifier 6-5
ENDFILE statement 6-117, F-7
End-of-File, definition A-2
End-of-Information, definition A-2
End-of-Partition, definition A-3
End-of-Record edit descriptor 6-33
Entry point, definition A-3
Entry points map 10-19
ENTRY statement 7-15, F-4
EOF (See End-of-File)
EOF routine 6-118
EOI (See End-of-Information)
Equivalence class, definition A-3
EQUIVALENCE statement 3-12, F-2
EQV intrinsic function 8-12
ERF intrinsic function 8-12
ERFC intrinsic function 8-12
ERR= specifier 6-5
Error handling subprograms 9-23
Example programs 6-86, 11-1
Execution
  Command 10-21
  Command parameters 10-21
Execution time, definition A-3
EXIT utility subprogram 9-29
EXP intrinsic function 8-13
Expressions
  Arithmetic 4-1
  Boolean 4-12
  Character 4-7
  Logical 4-9
  Relational 4-8
  Rules 4-13
Extended constant expression 4-1
Extended Hollerith constant 2-10
Extended internal file 6-57
External
  File, definition A-3
  Function 7-5
  Function references 7-8, A-3
EXTERNAL statement 3-15, F-2

F

F edit descriptor 6-22
Field, definition A-3
File
  Access methods 6-1
  Attributes 6-1, 6-93, A-3, E-1, E4
  Connection routines 6-120
  Definition of A-3
  Information table (FIT) 6-65, E-1
  Interface calls 6-75
  Name substitution 10-22
  Opening 6-2
  Organization, definition A-3
  Positioning 6-116, 10-4
  Reference, definition A-4
  Status statements 6-110
  Structure E-1
  Use of 6-1
File and record definition E-1

File attribute defaults E-2
File attributes
  Compiler output files E-4
  Indexed sequential files 6-90
  Run time files E-1
File Management Utility (FMU) 1-2
FILEIS call statement 6-76
FLOAT intrinsic function 8-13
Floating-point number, definition A-4
Flushing, definition A-4
FLUSHM call statement 6-77
FMT= specifier 6-5
FMU (See File Management Utility)
Format specification
  Character 6-11
  Description of 6-11
  Execution time 6-13
  Noncharacter 6-13
FORMAT statement 6-11, F-5
Formatted
  PRINT statement 6-10, F-4
  PUNCH statement 6-11, F-4
  READ statement 6-9, F-4
  WRITE statement 6-10, F-4
FORTRAN
  Callable subprograms 9-1
  Character set 2-1
  Collating sequence control subprograms 9-27
  Command parameters Inside front cover 10-1
  Compile command 10-1
  Compiler 1-1
  Debugging subprograms 9-22
  Differences between NOS/VE FORTRAN and FORTRAN 5 C-1
  Environment 1-3
  Error handling subprograms 9-23
  Example programs 6-86, 11-1
  Execution command 10-21
  Input/output related subprograms 9-30
  Sort/Merge subprograms 9-1
  Statements 2-1
  System Command Language subprograms 9-10
  Utility subprograms 9-21
FUNCTION statement 7-5, F-4
Function subprogram 7-5
F-type record, definition A-4

G

G edit descriptor 6-23
General utility subprograms 1-2, 2-1, 9-21
Generic function name, definition A-4
GET call statement 6-77
GETBVAL subprogram 9-12
GETCVAL subprogram 9-13
GETIVAL subprogram 9-13
GETN call statement 6-79
GETSCNT subprogram 9-14
GETSVAL subprogram 9-14
GETVCNT subprogram 9-14
GETVREF subprogram 9-15
Glossary A-1
GO TO statements
  Assigned 5-2
  Computed 5-1
  Syntax summary F-3
  Unconditional 5-1
Graphic, definition A-4

H

H edit descriptor 6-31

I

I edit descriptor 6-24
IABS intrinsic function 8-13
ICHAR intrinsic function 8-13
IDIM intrinsic function 8-13
IDINT intrinsic function 8-13
IDNINT intrinsic function 8-13
IF statements
  Arithmetic 5-4
  Block 5-5, 5-6
  Logical 5-4
  Nested 5-8
IFETCH call statement 6-87, 6-94
IFIX intrinsic function 8-13
Implicit type, definition A-4
IMPLICIT type statement 3-7, F-2
Implied DO list 3-20, 6-7
Indefinite value, definition A-4
Index
  Block 6-60, A-4
  Key types 6-49
  Master 6-50
  Multilevel file 6-50
  Record, definition A-4
INDEX intrinsic function 8-13
Indexed sequential
  Blocks 6-60
  Definition of A-4
  File processing 6-64
  File structure 6-59
  Keys 6-59
  Records 6-59
Infinite value, definition A-5
Initial lines
  Nonsequenced mode 2-2
  Sequenced mode 2-3

Input/Output
  Buffer 6-44
  Compile-time E-4
  File interface 6-59
  Formatted 6-9
  Internal 6-55
  List directed 6-37
  Lists 6-6
  Mass storage 6-46
  Namelist 6-40
  Related subprograms 6-110, 9-28
  Restrictions 6-4
  Runtime E-1
  Statement specifiers 6-5
  Status checking routines 6-117
  Unformatted 6-36
  Units 6-3
  Units map 10-20
INQUIRE statement 6-114, F-7
INT intrinsic function 8-14
Integer key, definition A-5
INTEGER type statement 3-2, F-1
Internal
  Data transfer routines 6-119
  READ statement 6-56
  WRITE statement 6-55
Internal File
  Definition of A-5
  Extended 6-57
  Standard 6-55
Intrinsic functions 7-6
INTRINSIC statement 3-16, F-2
Introduction to input/output 6-1
Introduction to NOS/VE FORTRAN 1-1
I/O Implementation E-1
IOCHEC routine 6-118
IOSTAT= specifier 6-6
ISIGN intrinsic function 8-14

J

JDATE utility subprogram 9-28
Job, definition A-5
Job log, definition A-5

K

Key, record 6-59, A-5

L

L edit descriptor 6-26
Language elements 2-1
Language syntax summary F-1
LEGVAR utility subprogram 9-23
LEN intrinsic function 8-14
LENGTH routine 6-119
LGE intrinsic function 8-14
LGT intrinsic function 8-14
Library, definition A-5
LIMERR utility subprogram 9-25
Lines
  Continuation 2-2, 2-4
  Nonsequenced 2-2
  Sequenced 2-3
List directed
  PRINT statement 6-38, F-4
  PUNCH statement 6-38, F-4
  READ statement 6-37, F-4
  WRITE statement 6-38, F-4
Listing control directives D-2
LLE intrinsic function 8-14
LLT intrinsic function 8-14
Load time, definition A-5
Local file name, definition A-5
LOG intrinsic function 8-14
Logical
  Assignment 4-16
  Constants 2-8
  Expressions 4-9
  IF statements 5-4
  Variables 2-13
LOGICAL type statement 3-5, F-1
LOG10 intrinsic function 8-15

M

Main program 7-2
Maintaining an indexed sequential file 6-105
Major key, definition A-5
Maps
  Common and equivalence 10-17
  DO loops 10-18
  Entry points 10-19
  General format 10-14
  Input/output units 10-20
  Namelist 10-16
  Procedures 10-19
  Statement labels 10-17
  Symbolic constants 10-16
  Unclassified names 10-20
  Variables 10-15
MASK intrinsic function 8-15
Mass storage
  CLOSMS statement 6-48
  Definition of A-6
  File, definition A-6
  Input/output 1-1, 6-46, A-6
  Input/output examples 6-51
  OPENMS statement 6-47
  Random file 6-46
  READMS statement 6-48
  STINDX statement 6-49

Master index 6-50
MAX intrinsic function 8-15
MAX0 intrinsic function 8-15
MAX1 intrinsic function 8-15
MIN intrinsic function 8-15
MIN0 intrinsic function 8-15
MIN1 intrinsic function 8-15
MOD intrinsic function 8-15
Monitoring an indexed sequential file 6-107
MOVLCH routine 6-120
MOVLEV routine 6-119
Multilevel file indexing 6-50
Multiple
  Assignment statement 4-1, 4-17
  Procedure entry 7-15

N

Named common block 3-10, A-6
Namelist
  Arrays in 6-43
  Map 10-16
  PRINT statement 6-41, F-5
  PUNCH statement 6-41, F-5
  READ statement 6-40, F-5
  WRITE statement 6-41, F-5
NAMELIST statement 6-40, F-5
NEQV intrinsic function 8-15
Nested
  DO loops 5-9, 5-10
  IF statements 5-3, 5-8
NINT intrinsic function 8-16
Nonembedded key, definition A-6
Nonrepeatable edit descriptors
  Apostrophe 6-31
  BN 6-30
  BZ 6-30
  Colon 6-34
  End-of-Record 6-34
  H 6-31
  P 6-29
  Quote 6-31
  S, SS, and SP 6-31
  Slash 6-33
  T, TR, and TL 6-33
  X 6-32
Nonsequenced
  Comment lines 2-3
  Compiler directive lines 2-3
  Continuation lines 2-2
  Initial lines 2-2
  Mode 2-2
  Positions 73 through 80 2-3
  Statement labels 2-2
NOS/VE environment 1-2
Notations 6
NUMERR utility subprogram 9-25

O

O edit descriptor 6-26
Object
  Code, definition A-6
  Program 1-1
Object Code Utility (OCU) 1-2
Octal constant 2-11
OCU (See Object Code Utility)
Open, definition A-6
OPEN statement 6-110, F-7
Opening a file 6-2
OPENM call statement 6-80
OPENMS statement 6-46
Optimization 10-9, A-6
OR intrinsic function 8-16
OSV$ASCII6_FOLDED collating sequence B-5
OSV$ASCII6_STRICT collating sequence B-7
OSV$COBOL6_FOLDED collating sequence B-9
OSV$COBOL6_STRICT collating sequence B-11
OSV$DISPLAY63_FOLDED collating sequence B-13
OSV$DISPLAY63_STRICT collating sequence B-15
OSV$DISPLAY64_FOLDED collating sequence B-17
OSV$DISPLAY64_STRICT collating sequence B-19
OSV$EBCDIC collating sequence B-21
OSV$EBCDIC6_FOLDED collating sequence B-26
OSV$EBCDIC6_STRICT collating sequence B-28
Output listing file 10-9
Owncode
  Definition of A-6
  Predefined points for execution 9-3
  Return codes 9-4
  Routines 9-3

P

P edit descriptor 6-29
Padding, definition A-6
Parameter interface subprograms 9-12
PARAMETER statement 3-1, 3-9, F-2
Parameters
  Compile command 10-1
  Execution command 10-21
Partition, definition A-6
Pass by reference, definition A-7
Pass by value, definition A-7
PAUSE statement 5-13, F-3
PDUMP utility subprogram 9-23
Permanent file, definition A-7
Positions 73 through 80 2-3
Predefined points for execution of owncode routines 9-4
Primary key, definition A-7
PRINT statement
  Formatted 6-10, F-4
  List directed 6-38, F-5
  Namelist 6-41, F-5

Printer control character 6-34
Procedure
  Argument association 7-10
  Arguments 7-9
  Communication 7-7
  Definition of 7-1, A-7
  Entry 7-15
  Maps 10-19
  Return 7-16
Processing an indexed sequential file 6-72
PROGRAM statement 7-2, 7-3, F-3
PROGRAM statement usage 7-2, 7-4
Program units 7-1, A-7
PUNCH statement
  Formatted 6-11, F-4
  List directed 6-38, F-5
  Namelist 6-41, F-5
PUT call statement 6-81
PUTREP call statement 6-82

Q

Quote edit descriptor 6-31

R

R edit descriptor 6-27
Random
  Access, definition A-7
  File, definition A-7
  Number generation 9-22
RANF intrinsic function 8-16
RANGET utility subprogram 9-22
RANSET utility subprogram 9-22
READ statement
  Direct access 6-53, F-5
  Formatted 6-9, F-4
  Internal 6-56
  List directed 6-37, F-5
  Namelist 6-40, F-5
  Unformatted 6-36, F-4
READMS statement 6-48
Real
  Constant 2-6
  Variable 2-12
REAL intrinsic function 8-16
REAL type statement 3-1, 3-3, F-1
REC= specifier 6-5
Record
  Definition of A-7
  Keys 9-2
  Length, definition A-7
Record and file definition E-1
Record length for unformatted I/O 6-53
Recreating an indexed sequential file 6-107
REDBVAR subprogram 9-17
REDCVAR subprogram 9-18
REDIVAR subprogram 9-18
REDSVAR subprogram 9-18
Reference
  Listing, definition A-7
  Map 1-1, 10-13
Relational
  Expressions 4-8
  Operators 4-9
REMARK utility subprogram 9-29
Repeatable edit descriptors
  A 6-17
  A for a noncharacter list item 6-18
  D 6-19
  E 6-19
  F 6-22
  G 6-23
  I 6-24
  L 6-26
  O 6-26
  R 6-27
  Z 6-28
REPLC call statement 6-83
Return codes set by owncode routines 9-5
RETURN statement 5-14, 7-16, F-4
Rewind, definition A-7
REWIND statement 6-116, F-7
REWND call statement 6-83
Rules for expressions 4-13
Run time
  Definition of A-7
  Input/output E-1

S

S, SS, and SP edit descriptors 6-31
SAVE statement 3-1, 3-14, F-2
SCL (See System Command Language)
SCLCMD subprogram 9-20
SCLKIND subprogram 9-15
SCU (See Source Code Utility)
SECOND utility subprogram 9-29
Sequenced
  Comment lines 2-4
  Compiler directive lines 2-4
  Continuation lines 2-4
  Initial lines 2-4
  Mode 2-3
  Statement labels 2-4
Sequential
  Access, definition 6-1, A-8
  Access input/output 6-1, A-8, E-3
  File, definition A-8
SET_FILE_ATTRIBUTE command 6-66, E-2
SHIFT intrinsic function 8-16

SIGN intrinsic function 8-16
SIN intrinsic function 8-16
SIND intrinsic function 8-16
SINH intrinsic function 8-16
SKIP call statement 6-84
Slash edit descriptor 6-33
SM5E Sort/Merge subprogram 9-4
SM5EL Sort/Merge subprogram 9-5
SM5END Sort/Merge subprogram 9-5
SM5FROM Sort/Merge subprogram 9-5
SM5KEY Sort/Merge subprogram 9-5
SM5LIST Sort/Merge subprogram 9-6
SM5MERG Sort/Merge subprogram 9-6
SM5OFL Sort/Merge subprogram 9-6
SM5OMRL Sort/Merge subprogram 9-7
SM5OWNn Sort/Merge subprogram 9-7
SM5RETA Sort/Merge subprogram 9-7
SM5SEQx Sort/Merge subprogram 9-7
SM5SEQA Sort/Merge subprogram 9-8
SM5SEQN Sort/Merge subprogram 9-8
SM5SEQR Sort/Merge subprogram 9-8
SM5SEQS Sort/Merge subprogram 9-9
SM5SORT Sort/Merge subprogram 9-9
SM5ST Sort/Merge subprogram 9-9
SM5SUM Sort/Merge subprogram 9-9
SM5TO Sort/Merge subprogram 9-10
SNGL intrinsic function 8-17
Sort/Merge
  Data types 9-2
  Subprograms 1-2, 9-4
Source
  Code, definition A-8
  Listing 10-12, A-8
  Program 1-1
Source Code Utility (SCU) 1-2
Specific function name, definition A-8
SQRT intrinsic function 8-17
SSWTCH utility subprogram 9-30
Standard FORTRAN capabilities 1-1
Standard internal files 6-55
STARTM call statement 6-85
Statement
  Function 7-7
  Function reference 7-9
Statement labels
  Map 10-17
  Nonsequenced mode 2-3
  Sequenced mode 2-4
Statement order 2-20
STINDX call statement 6-49
STOP statement 5-14, F-3
STOREF call statement 6-85
STRACE utility subprogram 9-23
Subindex 6-50
Subprograms
  Block data 7-6
  Debugging 9-22
  Error handling 9-25
  FORTRAN-callable 9-1
  Function 7-5
  Subroutine 7-4
SUBROUTINE statement 7-4, F-4
Subroutine subprogram 7-4
Substring references 2-18
Substrings and arrays 2-19
Symbolic constants map 10-16
Symbolic names 2-5
System Command Language
  Parameter interface subprograms 9-12
  Parameters 9-10
  Subprograms 1-2, 9-10
  User-defined parameters 10-23
  Variable interface subprograms 9-16
System Command Language (SCL), definition A-8
SYSTEM utility subprogram 9-24
SYSTEMC utility subprogram 9-24

T

T, TR, and TL edit descriptors 6-33
TAN intrinsic function 8-17
TAND intrinsic function 8-17
TANH intrinsic function 8-17
Task, definition A-8
TIME utility subprogram 9-28
Traceback 9-23, A-8
TSTPARM subprogram 9-15
TSTRANG subprogram 9-16
Type declaration statements
  BOOLEAN 3-1, 3-4
  CHARACTER 3-1, 3-5
  COMPLEX 3-1, 3-4
  DOUBLE PRECISION 3-1, 3-3
  INTEGER 3-1, 3-2
  LOGICAL 3-1, 3-4
  REAL 3-1, 3-3
  Syntax summary F-1

U

Unclassified names map 10-20
Uncollated key, definition A-8
Unconditional GO TO statement 5-1
Unformatted
  Input/output 6-36
  READ statement 6-36
  WRITE statement 6-36
Unit identifier 6-5, A-8
UNIT routine 6-117
Utility subprograms 1-2, 9-21
U-type record, definition A-8

V

Variable interface subprograms 9-16
Variables
  Boolean 2-13
  Character 2-13
  Complex 2-13
  Double precision 2-12
  Integer 2-12
  Logical 2-13
  Real 2-12
Variables map 10-15
V-type record, definition A-9

W

Working storage area 6-65, A-9
WRITE statement
  Direct access 6-53, F-5
  Formatted 6-10, F-4
  Internal 6-55
  List directed 6-38, F-5
  Namelist 6-41, F-5
  Unformatted 6-36, F-4
Writing FORTRAN statements 2-1
WRITMS call statement 6-47
WRTBVAR subprogram 9-19
WRTCVAR subprogram 9-19
WRTIVAR subprogram 9-19
WRTSVAR subprogram 9-20
WTSET subprogram 9-27

X

X edit descriptor 6-32
XOR intrinsic function 8-17

Z

Z edit descriptor 6-28

FORTRAN for NOS/VE, Usage 60485913 B

We would like your comments on this manual. While writing it, we made some assumptions about who would use it and how it would be used. Your comments will help us improve this manual. Please take a few minutes to reply.

Who Are You?

- ☐ Manager
- ☐ Systems Analyst or Programmer
- ☐ Applications Programmer
- ☐ Operator
- ☐ Other _____

How Do You Use This Manual?

- ☐ As an Overview
- ☐ To Learn the Product/System
- ☐ For Comprehensive Reference
- ☐ For Quick Look-up

Do You Also Have?

- ☐ FORTRAN Tutorial

What programming languages do you use? _____

Which are helpful to you? ☐ Parameter Summary (inside cover) ☐ Related Manuals Page
☐ Character Set ☐ Other _____

How Do You Like This Manual? Check those that apply.

Yes	Somewhat	No	
☐	☐	☐	Is the manual easy to read (print size, page layout, and so on)?
☐	☐	☐	Is it easy to understand?
☐	☐	☐	Is the order of topics logical?
☐	☐	☐	Are there enough examples?
☐	☐	☐	Are the examples helpful? (☐ Too simple ☐ Too complex)
☐	☐	☐	Is the technical information accurate?
☐	☐	☐	Can you easily find what you want?
☐	☐	☐	Do the illustrations help you?
☐	☐	☐	Does the manual tell you what you need to know about the topic?

Comments? If applicable, note page number and paragraph.

Would you like a reply? ☐ Yes ☐ No

From:

Name _____ Company _____

Address _____ Date _____

_____ Phone No. _____

_____

Please send program listing and output if applicable to your comment.

TAPE TAPE

FOLD FOLD

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 8241 MINNEAPOLIS, MINN.

POSTAGE WILL BE PAID BY

CONTROL DATA CORPORATION

Publications and Graphics Division

P.O. BOX 3492

Sunnyvale, California 94088-3492

CUT ALONG LINE

FOLD FOLD

FORTRAN Command Parameters

(Continued from inside front cover)

LIST_OPTIONS (LO)

default	LO=S
LO=A	List attributes of symbolic names.
LO=O	Produce object listing.
LO=R	Produce cross reference listing.
LO=M	Produce DO loop and common block map.
LO=S	List program source statements.
LO=SA	Same as S, but include lines turned off by C$ NOLIST.
LO=NONE	Do not produce an output listing.

You can select multiple options for the LO parameter.

MACHINE_DEPENDENT (MD)

default	MD=NONE
MD=NONE	Do not diagnose machine-dependent usages.
MD=T	Diagnose machine-dependent usages as trivial errors.
MD=W	Diagnose machine-dependent usages as warning errors.
MD=F	Diagnose machine-dependent usages as fatal errors.

ONE_TRIP_DO (OTD)

default	OTD=OFF
OTD=ON	Minimum trip count for DO loops is one.
OTD=OFF	Minimum trip count for DO loops is zero.

OPTIMIZATION (OPT)

default	OPT=DEBUG
OPT=DEBUG	Generate object code modified for debugging.
OPT=LOW	Perform minimum optimization.
OPT=HIGH	Perform maximum optimization.

RUNTIME_CHECKS (RC)

default	RC=(R,S)
RC=NONE	Do not perform run-time range checking.
RC=R	Perform range checking for character substring expressions.
RC=S	Perform range checking for subscript expressions.
RC=(R,S)	Select both R and S options.

SEQUENCED_LINES (SL)

default	SL=OFF
SL=ON	Source program is in sequenced format.
SL=OFF	Source program is in nonsequenced format.

STANDARDS_DIAGNOSTICS (SD)

default	SD=NONE
SD=NONE	Do not diagnose non-ANSI usages.
SD=T	Treat non-ANSI usages as trivial errors.
SD=W	Treat non-ANSI usages as warning errors.
SD=F	Treat non-ANSI usages as fatal errors.

STATUS

default	Do not return error status.
STATUS=v	Write error status code to SCL specified variable.

TERMINATION_ERROR_LEVEL (TEL)

default	TEL=F
TEL=T	Return abnormal status for trivial errors only.
TEL=W	Return abnormal status for warning, fatal, and catastrophic errors.
TEL=F	Return abnormal status for fatal and catastrophic errors.
TEL=C	Return abnormal status for catastrophic errors only.

CONTROL DATA

60485913