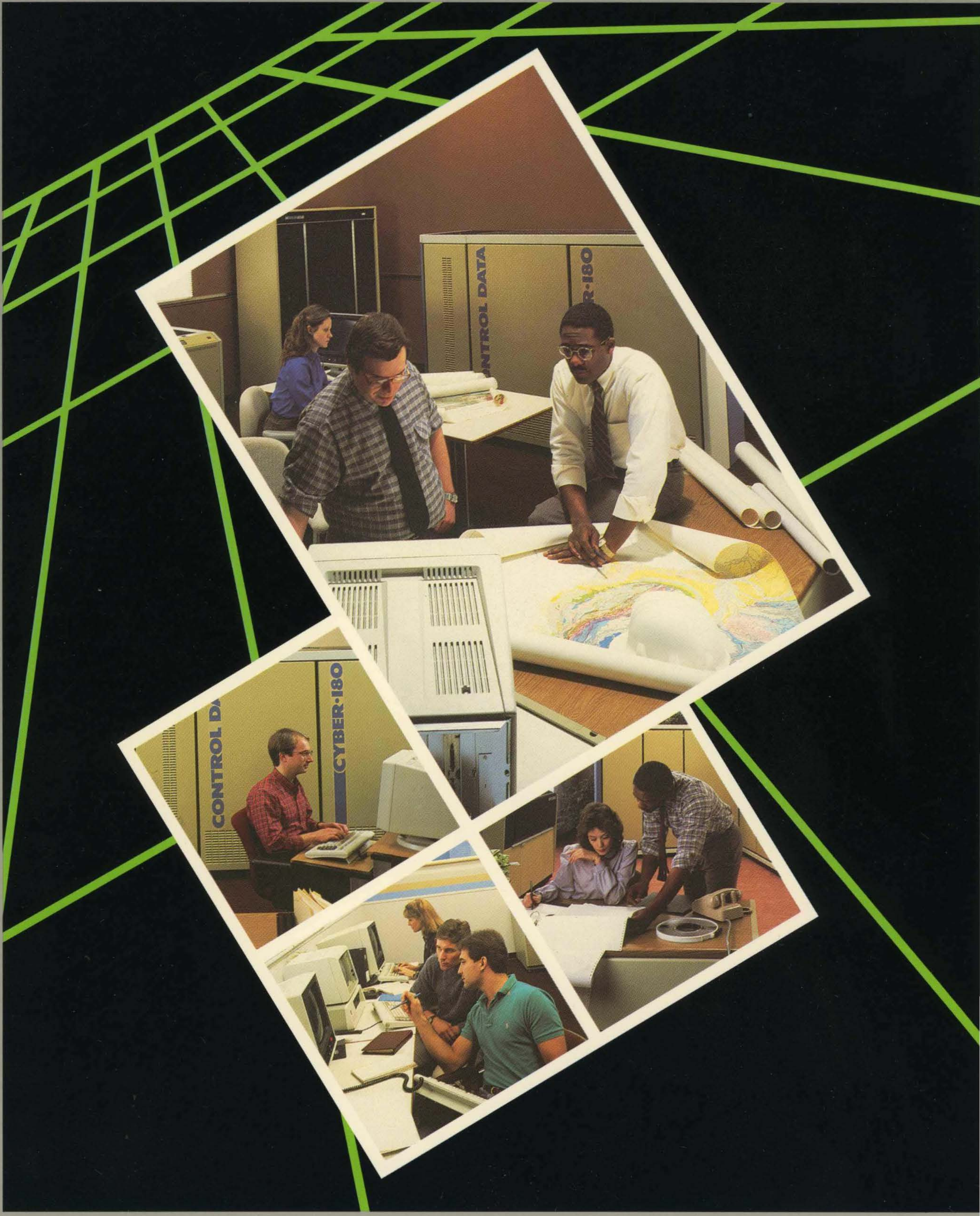
FORTRAN Version 1 for NOS/VE Language Definition

CONTROL DATA



Usage

60485913

FORTRAN Version 1 for NOS/VE Language Definition

Usage

Publication Number 60485913

Manual History

Revision	System Version	Product Level	PSR Level	Date
A	1.0.2	1.0	598	October, 1983
B	1.1.1	1.0	613	June, 1984
C	1.1.2	1.0	630	March, 1985
D	1.1.3	1.1	644	October, 1985
E	1.1.4	1.2	649	January, 1986
F	1.2.1	1.2	664	July, 1986
G	1.2.2	1.2	678	April, 1987
H	1.2.3	1.2	688	September 1987

Revision H documents the FORTRAN Version 1 language for NOS/VE at release 1.2.3, PSR level 688. This revision reflects technical and editorial changes as well as the following new features: Variable length (*n) integer, real, and complex constants, variables, arrays, and functions; four NOS/VE status processing routines; a user condition flag handling routine (CHGUCF); C$ EXTERNAL interface with FORTRAN Version 2; and the TARGET_MAINFRAME parameter on the FORTRAN command. The $LOG_RESIDENCE and $LOGGING_OPTIONS recovery attributes have been added to the keyed-file interface.

Contents

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.

FORTRAN Quick Reference (Online)

This manual provides an online quick reference for the FORTRAN commands, statements, functions, and subprograms. Parameter descriptions and examples are included.

Language Definition Manual

This manual presents detailed descriptions and definitions of all the statements and features of the NOS/VE FORTRAN language, including the Sort/Merge and Keyed-File interfaces. Examples of statements and programs are included.

Conventions

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.

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.
Δ	Space character. This symbol is used wherever there might otherwise be doubt as to how many spaces 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.

Vertical bars in the margin indicate changes or additions to the text from the previous revision. An example of a change bar is shown in the margin next to this paragraph.

Control Data Extension

Control Data (CDC) Extensions

Major descriptions of Control Data Extensions to standard ANSI FORTRAN (CDC Extensions) are marked as this section is marked. References to CDC extensions (other than the major description or definition) are indicated by shading of the text referring to the nonstandard feature, as this sentence is shaded. A complete list of CDC extensions is presented in appendix G.

End of Control Data Extension

Ordering Printed Manuals

Control Data manuals are available through your local Control Data sales offices. Sites within the U.S. can also order manuals directly from Control Data Literature Distribution Services at the following address:

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
Technology and Publications Division
P. O. Box 3492
Sunnyvale, California 94088-3492

Please indicate whether you would like a written reply.

Be sure to include the following information with your comment:

FORTRAN for NOS/VE Language Definition Usage Manual
Publication number 60485913
Revision G

Also, if you have access to SOLVER, the CDC online facility for reporting problems, you can use it to submit comments about this manual. When it prompts you for a product identifier for your report, please specify FN8.

In Case of Trouble

Control Data's Central Software Support maintains a hotline to assist you if you have trouble using our products. If you need help beyond that provided in the documentation or find that the product does not perform as described, call us at one of the following numbers and a support analyst will work with you.

From the USA and Canada: (800) 345-9903

From other countries: (612) 851-4131

The preceding numbers are for help on product usage. Address questions about the physical packaging and/or distribution of printed manuals to Literature and Distribution Services at the following address:

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

or you can call (612) 292-2101. If you are a Control Data employee, call (612) 292-2100.

Introduction to NOS/VE FORTRAN 1

This chapter present a brief introduction to NOS/VE FORTRAN Version 1.

Introduction to NOS/VE FORTRAN 1

NOS/VE FORTRAN Version 1 provides the features and capabilities set forth in the ANSI FORTRAN Standard as well as several Control Data unique features.

Standard Features

NOS/VE FORTRAN Version 1 (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.

Control Data (CDC) Extensions

In addition to the standard features, NOS/VE 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 detects 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; 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 to individual records than conventional sequential input/output. Mass storage input/output is described in chapter 6.

Segment access files

Allows you to map a file to a named common block and reference entities in the file as normal variables in a named common block. Segment access files are described in chapter 6 and appendix D.

C$ Directives

Allow you to 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. Descriptions of CDC extensions in this manual are indicated by shaded text.

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 a 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 that can make creation and maintenance of FORTRAN programs easier and more efficient. The following facilities can be used outside your program:

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. SCU is described in the SCL Source Code Management manual.

Object Code Utilities (OCM)

Allows 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. They can be used to measure and analyze program performance. These utilities are described in the SCL Object Code Management manual.

File Migration Aid (FMA)

Allows you to read, write, and edit ANSI standard FORTRAN files on the NOS or NOS/BE side of a dual state system from your FORTRAN program running on NOS/VE.

File Management Utility (FMU)

Allows you to convert data files from one format to another. FMU can be used to convert files from other Control Data systems for use on NOS/VE. FMU is described in the SCL Advanced File Management manual.

Debug Utility

Enables you to debug a program during its execution. You can stop the program at selected points or on the occurrence of an error, and request formatted displays of variables and arrays. The Debug utility is described in the Debug for NOS/VE manual. A brief introduction is presented in appendix K.

Programming Environment (PE)

Allows you to create, debug, and run FORTRAN programs in an integrated environment with a full-screen interface. Access is available to the editor, Debug, and the online manuals. The Programming Environment is described in the Programming Environment manual. A brief introduction is presented in appendix I.

Professional Programming Environment (PPE)

Coordinates complex programming projects using an integrated, full-screen environment. Provides access to NOS/VE development tools including the Full Screen Editor, the Source Code utility, the Debug utility, and the object library generator. The Professional Programming Environment is described in the Professional Programming Environment manual. A brief introduction is presented in appendix I.

The following facilities can be used inside your FORTRAN program.

Sort/Merge Calls

These calls enable you to sort the records of one of more files into a specific order, and to merge the sorted records of two or more files into a single file. These calls are described in chapter 12.

System Command Language Calls

These calls provide a method of communicating with the operating system using the System Command Language (SCL). The parameter interface calls enable you to reference the parameters specified on the command that began execution of the 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. The SCLCMD call allows you to execute any SCL command from within your program. These calls are described in chapter 9.

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 system. These calls are described in chapter 9.

Keyed File Calls

These calls enable you to create and use the keyed file organizations (indexed-sequential and direct-access) in your FORTRAN program. These calls are described in chapter 11.

Screen Design Facility (SDF)

The Screen Design Facility (SDF) is an interactive screen designing tool that runs in the NOS/VE environment. SDF enables the designers of a system or application to create, modify, and maintain the display and data entry screens that are presented to the end user of a system or application. The Screen Design Facility is described in the Screen Design Facility manual.

Screen Formatting

A tool that enables the application designer to provide a full-screen environment for the end user. Screen formatting provides a comprehensive set of object routines that control the flow of displaying SDF screens and manipulating data associated with the screens. Screen Formatting is described in the Screen Formatting manual.

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

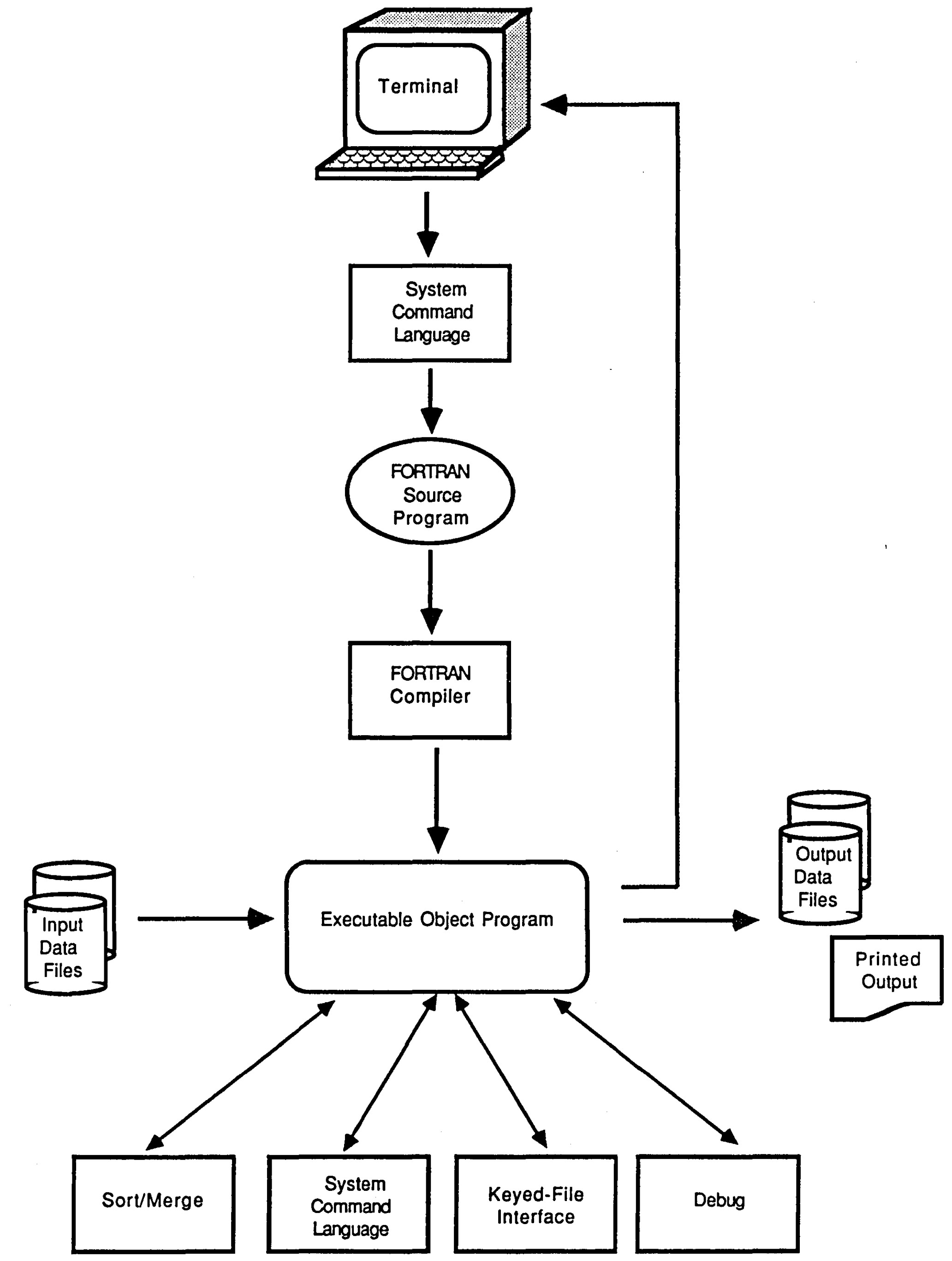


Figure 1-1. FORTRAN in the NOS/VE Environment

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:[1]

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
	! exclamation point
	space

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
```

1. The ASCII representations of the characters are shown in appendix J.

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 string, boolean string, and extended Hollerith constants

Apostrophe, H, and quote edit descriptors in format specifications

Comments (inline and line)

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 through 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                  ! THIS INITIALIZES THE
         LROW(I) = 1                  ! ARRAY LROW
 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 spaces to improve readability. The initial line of a statement can contain a statement label in positions 1 through 5. Position 6 contains a space.

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 contain spaces.

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. Spaces 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.

Comments

Comments provide a method of placing program documentation in the source program. Comments in nonsequenced mode can appear as inline comments or as entire comment lines.

An inline comment (CDC Extension) is written on the same line as a FORTRAN statement. The exclamation point (!) terminates the FORTRAN statement and marks the beginning of an inline comment. The appearance of an inline comment does not affect the preceding FORTRAN statement (even if it is an END statement). An exclamation point (!) appearing in column 6 marks a continuation line.

A comment line is indicated by a C, an asterisk (*), or an exclamation point (!) in position 1. Comment lines do not affect the program and can be placed anywhere within the program. Comment lines can appear between an initial line and a continuation line, or between two continuation lines. A comment line following an END statement is treated as the first line of the next program unit. (Program units are described in Chapter 7.) Any line with spaces in positions 1 through 72 is also a comment line.

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 J for the character set being used.

Control Data Extension

Compiler Directive Lines

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.

End of Control Data Extension

Positions 73 and Beyond

Positions 73 and beyond 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 cards, positions 73 through 80 can be used for identification information.

Control Data Extension

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 digits. 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       ! THIS INITIALIZES THE
00160        LROW(I) = 1       ! ARRAY LROW
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 non-space characters in the line. The sequence number consists of one through five digits, usually at the beginning of a line. Spaces can precede the sequence number.

The statement can begin immediately after one or more spaces 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 space 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 spaces. Leading spaces and leading zeros within a label are disregarded. A label must not contain embedded spaces.

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. Spaces are optional between the + and the continuation of the statement.

Comments

Comments in sequenced mode can appear as inline comments or as entire comment lines.

An inline comment appears on the same line as a FORTRAN statement. The exclamation point (!) terminates the FORTRAN statement and marks the beginning of an inline comment. The appearance of an inline comment does not affect the preceding FORTRAN statement. An exclamation point (!) appearing in column 6 marks a continuation line.

A comment line in sequenced mode is indicated by any character except blank or + immediately following the sequence 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 spaces between C$ and the beginning of the directive. The directive cannot be continued on subsequent lines.

End of Control Data Extension

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 and spaces within a symbolic name are suppressed. Symbolic names are used for the following:

- Main program names
- Common block names
- Element names

  Variable name
  Array name
  Symbolic constant name
- Subprogram names

  Subroutine name
  Block data subprogram name
  Dummy subprogram name
- Function names

  Statement function name
  Intrinsic function name
  External function name
- Namelist group names

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. Integer constants are one of three sizes: two bytes (one fourth of a computer word), four bytes (one half of a computer word, and eight bytes (a full computer word). The value of the constant determines the integer size as follows:

Constant value	Size
-32768 to 32767	2 byte
-2147483648 through -32768 and 32768 through 2147483647	4 byte
-(2**63) through -2147483848 and 2147483848 through (2**63) - 1	8 byte

NOTE

(2**63)-1 is equal to 9,223,372,036,775,807.

When a value is converted from real to integer, the above ranges are still valid.

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

NOTE

Throughout this manual, whenever an integer constant or variable is allowed, it can be of any length unless otherwise noted.

For Better Performance

Four-byte, and especially two-byte integer constants, variables, and functions can increase the execution time of your program. This is because such integers are byte aligned rather than word aligned and therefore slower to load and store.

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.

A real constant can be one of two sizes: eight bytes (one computer word), and 16 bytes (two consecutive computer words). Real values that are 16 bytes in size are written with a D exponent as described for double precision constants; 16-byte real constants are treated as double precision values. The range of real constants is as follows:

Range	Constant Size
-10.**(-1232) through -10.**(1234) 0. 10.**(-1234) through 10.**(1232)	8 byte
-10.**(-1232) through -10.**(1234) 0. 10.**(-1234) through 10.**(1232)	16 byte

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 be present. 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 computer words (16 bytes), 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

For Better Performance

Double precision constants, variables, arrays, and functions require more execution time because of the extra precision they support (two words).

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 (16 bytes) 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 a word with a leftmost bit of 1 and a word with a leftmost bit of 0, respectively.

Control Data Extension

Boolean

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 boolean string constant 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. (Boolean string constants in format specifications are not limited to eight characters.)

The nHs and "s" forms 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 the space character (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 nHs 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

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:

`O"77"`	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
`Z"ace"`	Value is 00...ACE 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((N + 8 − 1)/8)

where N 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 spaces 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. Spaces 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 the space character (ASCII code 20 hex).

Example:

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

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 hex

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 hex

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

Examples:

`"ABCDEFGHIJ"`	Value is 4142434445464748|494A20...20 hex
`"QRSTU""VWXYZ"`	Value is 5152535455235657|58595A20...20 hex (represents the string QRSTU"VWXYZ)

End of Control Data Extension

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. Spaces 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

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 is one of three sizes: two bytes (one fourth of a computer word), four bytes (one half of a computer word), and eight bytes (a full computer word). You can specify the size of a variable when it is explicitly or implicitly typed. Default integer variables are within the ranges specified below:

Variable Size	Range
-(2**63) through (2**63)-1	8 byte
-(2**31) through (2**31)-1	4 byte
-(2**15) through (2**15)-1	2 byte

NOTE

(2**63)-1 is equal to 9,223,372,036,775,807.

(2**31)-1 is equal to 2,147,483,647.

(2**15)-1 is equal to 32767.

See chapter 3 for restrictions on integer variables and constants in DO statements.

Examples:

```
ITEM1  jsum  J

Nsum   N72   K2SO4
```

For Better Performance

Four-byte, and especially, two-byte integer constants, variables, and functions can increase the execution time of your program. This is because such integers are byte aligned rather than word aligned and therefore slower to load and store.

Real Variables

A real variable is a variable that is typed explicitly, implicitly, or by default as real. A real variable can occupy one computer word (8 bytes) or two consecutive computer words (16 bytes). The valid values for a real variable are:

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

0

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

One word real variables support approximately 14 significant digits of precision. Two word real variables support approximately 29 digits of precision.

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 computer words. The first word contains the more significant part of the number and the second word contains the less significant part.

Example:

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

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

For Better Performance

Double precision and 16 byte real constants, variables, arrays, and functions require more execution time because of the extra precision they support (two words).

Complex Variables

A complex variable is a variable that is typed explicitly or implicitly as complex. A complex variable occupies two computer words (16 bytes); 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 computer 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.

Control Data Extension

Boolean Variables

A boolean variable is a variable that is typed explicitly or implicitly as boolean. A boolean variable occupies one computer 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.

End of Control Data Extension

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. The maximum length of a character variable is 65,535 characters.

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 first letter of the variable name determines its type). 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 (any size) 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 (any size) 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 the distance between the lower bound and upper bound.

A dimension bound expression can contain symbolic constants defined in previous PARAMETER statements; integer constants can be of any size. A dimension bound expression in a function or subroutine can contain dummy arguments.

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

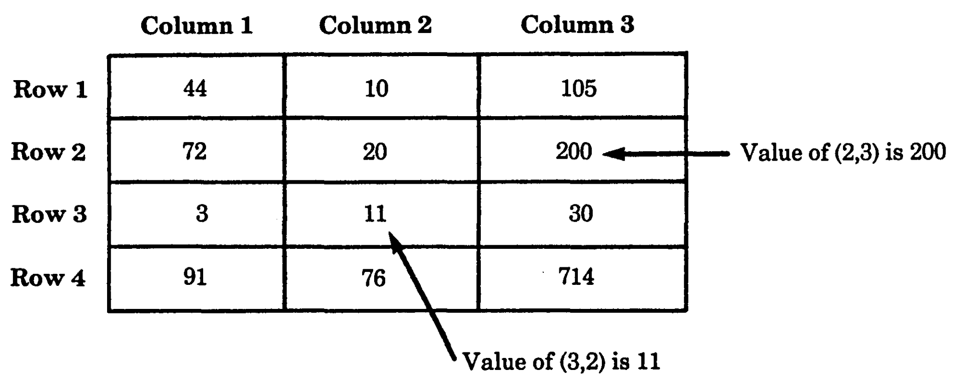
The following examples show storage patterns for a one-dimensional, two-dimensional, and three-dimensional array respectively. Arithmetic values are shown for the array elements, but an array can be any data type or size. Array elements are stored in ascending locations by positions. The first subscript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

Example of a one-dimensional 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:



	Column 1	Column 2	Column 3
Row 1	44	10	105
Row 2	72	20	200
Row 3	3	11	30
Row 4	91	76	714

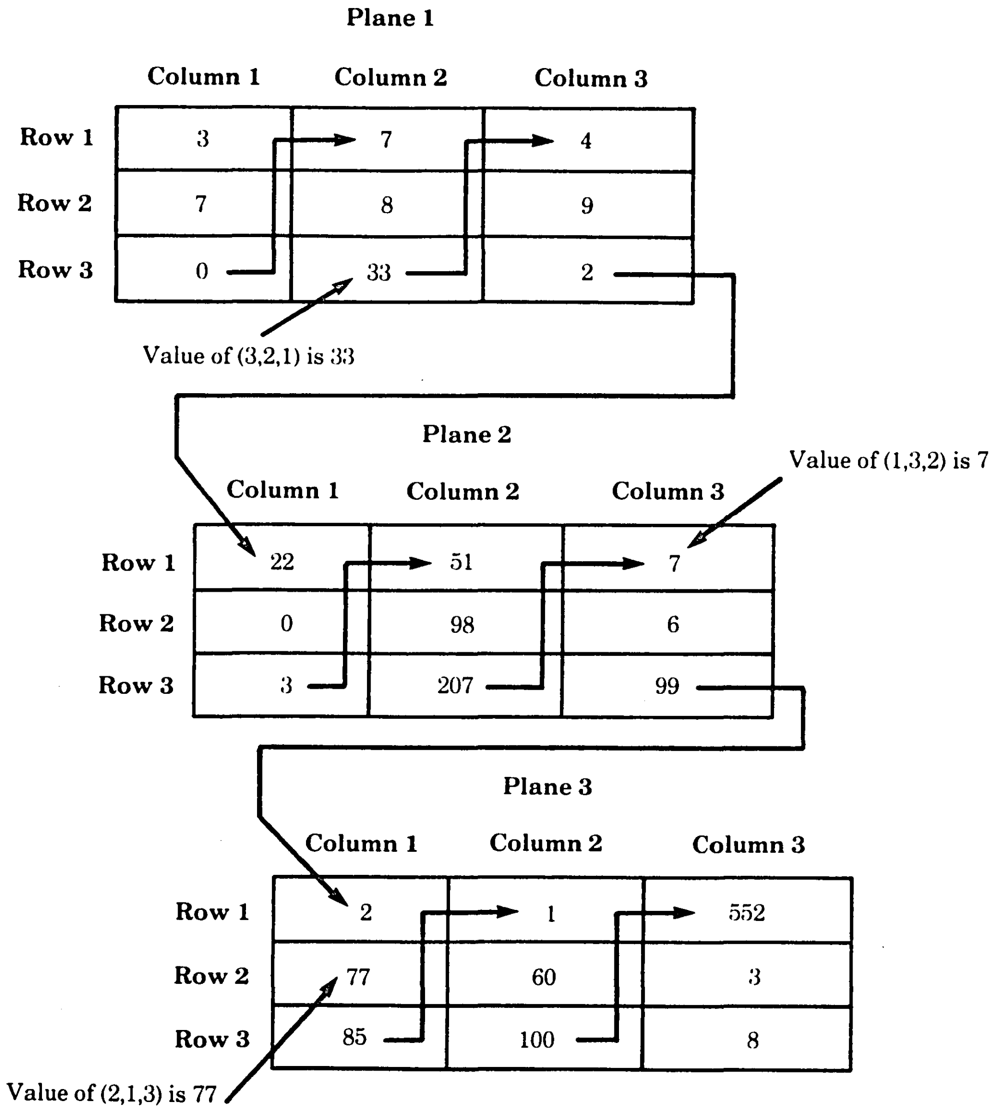
Value of (2,3) is 200
Value of (3,2) is 11

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

For Better Performance

Adjustable (variable-dimensioned) arrays increase execution time; replace with constant arrays whenever possible.

Example of a three-dimensional array:



The array has three rows, three columns, and three planes, for a total of 27 elements.

For Better Performance

The number of different variable names in subscript expressions should be minimized. For example, the following statements use two variables in each subscript expression:

```
IP1=I+1
IM1=I-1
X=A(IP1, IM1) + B(IM1,IP1)
```

The following statement produces the same result but executes faster:

```
X=A(I+1,I-1) + B(I-1, I+1)
```

Array Storage

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

The span (number of elements) 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 computer words reserved for an array is determined by the type and size of the array and the spans of its dimensions. An array of type eight-byte integer, boolean, eight-byte real, or logical occupies n computer words, where n is the product of the spans of all dimensions. An array of type four-byte integer occupies n/2 computer words and an array of type two-byte integer occupies n/4 computer words. 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.)

NOTE

For a program compiled with OPTIMIZATION_LEVEL = HIGH on the FORTRAN command, storage is not allocated at load time for arrays unless they are in a common block, saved (by a SAVE statement or FORCED_SAVE=ON compiler option), initialized in a DATA statement, or used as actual arguments. Instead, storage is allocated for them on the runtime stack during execution, only when the containing program unit becomes active. This storage is then given up on execution of a RETURN or END statement in the program unit. The default runtime stack size is about 2 million bytes. If this limit is exceeded, a runtime error results, usually of the form "Tried to read/write beyond maximum segment length" or "A stack segment contains invalid frames". For programs where the number of active items allocated on the runtime stack exceeds the default limit, you can increase the runtime stack size by specifying the STACK_SIZE parameter on the EXECUTE_TASK command (as described in the SCL Object Code Management Usage manual).

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 (any size), 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 (by the INT function). 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 selected 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:

Table 2-1. Array Element Positions

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; ni = ki - ji + 1. If the lower bound is one, ni = ki.

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, position (1 + (6 - 1)).

Example:

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

The array element reference CQ(6,3) refers to the element at position 18, that is, position (1 + (6 - 1) + (3 - 1) * 6). The character storage position is 52 relative to the first position of the array, 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 < first < last < len

For example, substring 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. The colon is required in substring references; S1(3) is not a valid substring reference.

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

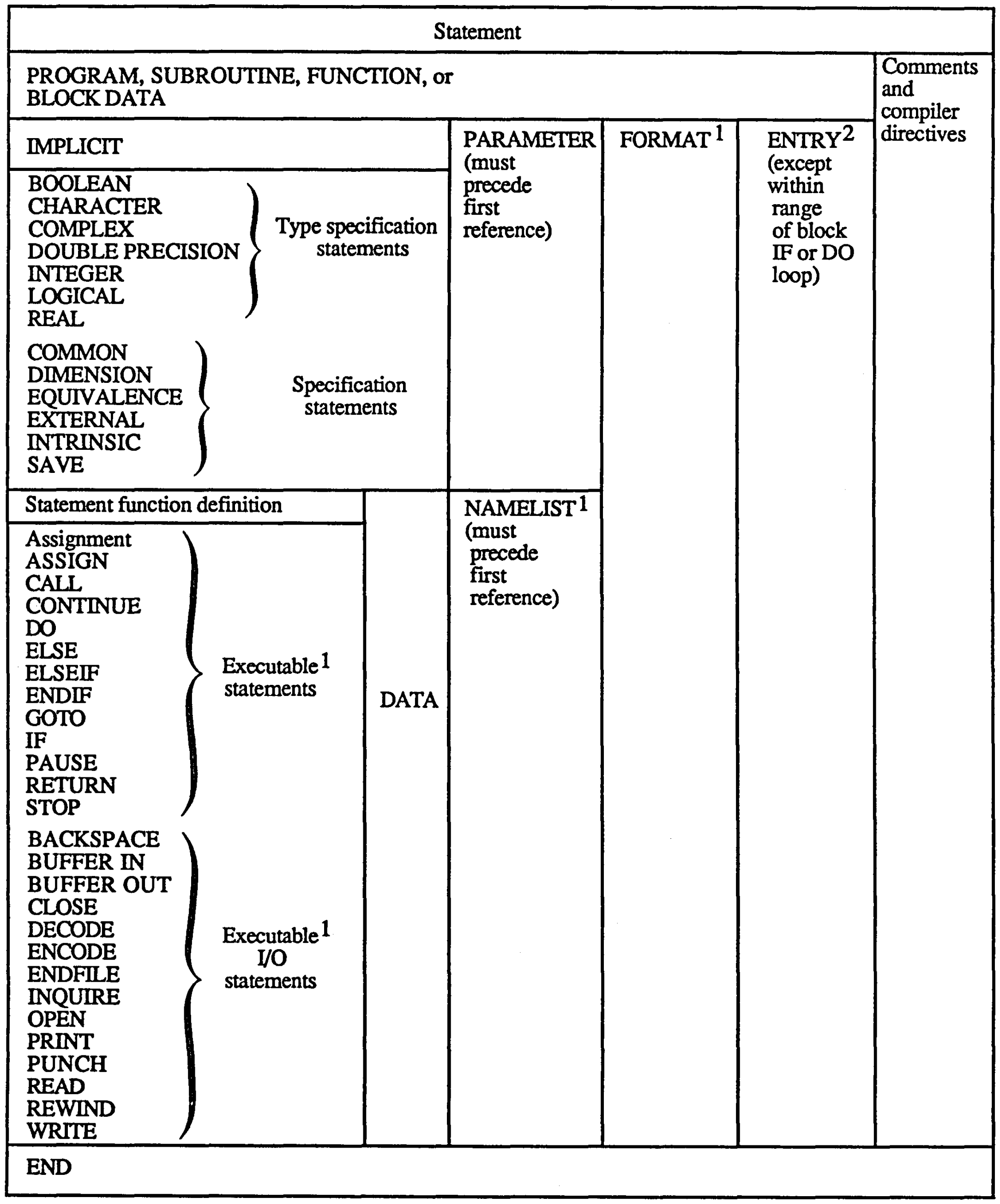
If a substring reference is used to select a substring from an array element of a character array, the combined reference includes specification of the array element followed by specification of the substring. For example:

```
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 figure 2-1.



[1] Cannot be used in a BLOCK DATA subprogram.

[2] Cannot be used in a main program or BLOCK DATA subprogram.

Figure 2-1. Statement Order

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 except:

- Within the range of a DO loop (between the DO statement and the final statement of the DO loop)
- 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 (including IMPLICIT NONE) 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

This chapter describes the statements used to specify the characteristics of entities within a FORTRAN program.

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 of size 2, 4, or 8 bytes. Eight bytes is one computer word. This statement has the form:

INTEGER*len* **name** **len*, ..., *name*len*

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.)

len

Length (in bytes) of name. Options are 2, 4, and 8. If omitted, the length is 8 bytes which is equal to one computer word.

A length specification immediately following the INTEGER keyword 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. For an array, the length specified is for each array element. If a length is not specified for an entity, either explicitly or by the IMPLICIT statement, the length is 8.

Examples:

```
INTEGER ITEM1, NSUM, JSUM

integer a72, h2so4

INTEGER M5(2)

INTEGER*4  HALFI, HALFJ

INTEGER*2  FLAG

INTEGER I*4, J, K
```

REAL Statement

The REAL statement declares a variable, array, symbolic constant, function name, or dummy procedure name to be type real of size 8 or 16 bytes. Eight bytes is one computer word. The REAL statement has the form:

REAL*len* **name***len*, ..., *name*len*

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.

len

Length (in bytes) of name. Options are 8 and 16. A name declared with length 16 is treated as a double precision value. If omitted, the length is 8 bytes.

A length specification immediately following the REAL keyword 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. For an array, the length specified is for each array element. If a length is not specified for an entity, either explicitly or by the IMPLICIT statement, the length is 8.

Examples:

```
REAL IVAR, NSUM3, RESULT

real total2, BETA, XXXX

REAL*8  TR(10,5)

REAL*16 ALPHA

REAL A, B*16
```

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

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.

len

Length (in bytes) of name; 16 is the only option. If omitted, the default length is 16 bytes.

A length specification immediately following the COMPLEX keyword 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. For an array, the length specified is for each array element. If a length is not specified for an entity, either explicitly or by the IMPLICIT statement, the length is 16.

Examples:

```
COMPLEX CPVAR

COMPLEX RES(5,5)
```

Control Data Extension

BOOLEAN Statement

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)
```

End of Control Data Extension

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 logical values true or false.

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 (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 (one byte of memory).

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.

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, integer, real, or complex, 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)
```

The variable AB is 5 characters long while the variable CD is 8 characters long. 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
```

Incorrect

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 (BR is passed). 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)*

IMPLICIT NONE

type

One of the keyword values INTEGER**len*, REAL**len*, DOUBLE PRECISION, COMPLEX**len*, BOOLEAN, LOGICAL, or CHARACTER**clen*.

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.

clen

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

len

For integer entities, len must be an extended integer constant expression with a positive value of 2, 4, or 8. For real entities, len must be an extended integer constant expression with a positive value of 8 or 16. For complex entities, len must be an extended integer constant expression with a positive value of 16.

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 or IMPLICIT NONE 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, integer, and real, you can also specify an implicit length. If you do not specify the length of character entities, it defaults to one. The length must have a positive nonzero value. The specified length applies to all entities implicitly typed as character. Options for integer entities are 2, 4, and 8. Options for real entities are 8 and 16. For entities of type integer and real, the default length is eight. Real entities typed with a length of 16 are treated as though they were typed double precision. The only option for complex entities is 16.

IMPLICIT NONE specifies that there should be no default implied typing for symbolic names in a program unit. No other form of the IMPLICIT statement may be used with the IMPLICIT NONE statement. If another form of the IMPLICIT statement does occur, a fatal diagnostic will be issued.

An IMPLICIT NONE statement does not affect the type of any of the intrinsic functions. The use of IMPLICIT NONE in a program unit causes the detection of undeclared names and a fatal error is issued.

If a program, subroutine or function contains an IMPLICIT NONE statement, then all names of variables, arrays, symbolic constants, external functions and statement functions within that program unit must be explicitly declared in a type statement.

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.

Example:

```
   IMPLICIT INTEGER*2 (I-L)
```

The default typing is changed for names beginning with the letters I, J, K, or L. These names are typed as integers occupying two bytes of memory each; their range is -2**15 through (2**15)-1.

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.

Example:

```
   PROGRAM MAIN
   IMPLICIT NONE
   REAL  A,B,C
   INTEGER  I,J,X
   CHARACTER*4  JOHN
```

The IMPLICIT NONE requires that all variables in the program be explicitly typed by a type statement (INTEGER, REAL, DOUBLE PRECISION, COMPLEX, BOOLEAN, LOGICAL or CHARACTER).

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
*lower:***upper**

where upper and lower are as follows:

lower

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

upper

Upper bound of the dimension; a dimension bound expression in which all variables and intrinsic function references are type integer, and all constants, and symbolic names of constants, are type integer or boolean. (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 or 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 constant is not the default length for that type, 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 for symbolic character constants 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

- In a dimension bound expression in an array declaration

A symbolic constant cannot appear in a format specification. A character symbolic constant cannot be used in a substring reference.

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. You should ensure that every entity in a common block definition is word aligned if you intend to equivalence. A word contains 8 bytes or characters. For example:

```
INTEGER J, K, L
CHARACTER CDAY*5, CFILL*3
CFILL=' '
COMMON /BLK1/J, CDAY, CFILL, K, L
  :
EQUIVALENCE K, M
```

The character variable CFILL is placed in the common block after CDAY to fill a word. (A word contains eight characters.) The integer values K and L will begin on a word boundary.

Variables and arrays in a common block can be of different data types . 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 computer words (for character data, 4,294,967,296 characters).

The actual size of any common block is the number of computer 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 computer words at the end of the common block or by using the C$ EXTEND directive. (See the descriptions of the EQUIVALENCE statement in this chapter and the C$ EXTEND directive in appendix D.) 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 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.

Variables and arrays in named common blocks can be initially defined by a DATA statement in a block data subprogram, or 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.

Variables and arrays in blank or unlabeled common remain defined at all times, and do not become undefined on returning from a subprogram.

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 computer 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 computer words are known by the names Y and Z in function T.

Example:

```
COMMON JCOUNT
  :
JCOUNT = 6
  :
FUNCTION AB(A)
COMMON /C/ STX(4)
DATA STX/1., 2., 2., 1./
```

For Better Performance

You can access large blocks of data more efficiently using segment access files mapped to common blocks. See the description of C$ SEGFILE in appendix D.

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
INTEGER J COMMON /CVAL/ D, E, J
DATA D, E, J/'TEST', 'PROD', 10/
```

The common block named CVAL contains two character variables and one integer variable. The variables 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 computer 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 computer words.

For Better Performance

Common blocks should not contain unnecessary items; common blocks can be used to store items that would otherwise be passed as parameters to the subroutines.

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 references. 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. (Note that ANSI does not allow equivalencing of character and noncharacter data.) 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. The lengths of 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 computer 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. When equivalencing non full-word (byte aligned) character entities in a common block to other full-word aligned entities, you must ensure that each entity in the common block begins on a word boundary (see 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 (without a subscript) 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 computer 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. The equivalencing associates the first computer 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)          }  Not legal
```

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

```
REAL BZ(7),CZ(5)
EQUIVALENCE (BZ,CZ),(BZ(3),CZ(4))         }  Not legal
```

is also illegal.

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

1. 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                       }  Not legal
   EQUIVALENCE (T,S)
   ```

   is not legal.

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

   ```
   COMMON /X/ A
   REAL B(5)                              }  Not legal
   EQUIVALENCE (A, B(4))
   ```

   is not legal, whereas

   ```
   COMMON /X/ A
   REAL B(5)                              }  Legal
   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 or array name enclosed in slashes. The same name must not appear more than once. If no names are specified all variables and arrays are saved.

A SAVE statement is optional and has no effect in a main program or in any subprogram compiled with OL=DEBUG or OL=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. You do not have to specify a named or blank common block or an entity within a named or blank common block; storage given to a common block will not be reused during execution of your program. A SAVE statement with no list is treated as though it contained the names of all allowable items in the program unit.

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 or named common do not become undefined.
- Entities that have been defined in a DATA statement do not become undefined.
- Entities in a subprogram compiled with OL=DEBUG or OL=LOW specified on the FORTRAN command.
- Entities that are associated with saved entities by EQUIVALENCE statements do not become undefined.

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 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 and the two entities in common block C1 remain defined. 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.

Specifying the name of a block data subprogram in an EXTERNAL statement causes the loader to search the object libraries for the block data subprogram.

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.

A character constant associated with an arithmetic or boolean data item is treated as a boolean string constant.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 initially defined more than once in a DATA statement, the last definition overrides any previous ones.

Names of dummy arguments, functions, and entities in blank, extensible, or C$ SEGFILE common (including any entities associated with an entity in the common block) cannot be initially defined. Entities in a named common block can be initially defined within a block data subprogram, or (in NOS/VE 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.

For arithmetic data types, the constant is converted to the type of the associated nlist item if the types differ. For all other types, the data type of each constant in clist must be compatible with the data type of the nlist item. The correspondence is shown in the following table:

Data Type of nlist Item	Data Type of Corresponding clist Constant
Integer, real, double precision, complex, or boolean.	Integer, real, double precision, complex, character, or boolean. The value of the nlist item is the same as would result from an assignment statement of the form: nlist-item = clist-constant. Only the first word of double precision or complex nlist data is defined by boolean or character clist data. When assigning character data to noncharacter nlist items, you must ensure word alignment if equivalencing of the data is to occur.
Logical	Logical
Character	Character

If the length of the nlist item is not the same as the corresonding clist Hollerith constant, the Hollerith constant is blank filled if too small, or truncated from the right if too large. Constants in clist that are octal or hexadecimal are zero-filled if too small, or truncated from the left if too large.

Each subscript expression used in an array element name in nlist must be an extended integer constant expression, except that implied DO variables can be used if the array element name is in dlist. A reference to an implied DO variable must be within the range of the implied DO. Each substring expression used for an item in nlist must be an extended integer constant expression.

Example:

```
INTEGER K(6)
DATA JR/4/
DATA AT/5.0/, AQ/7.5/
DATA NRX, SRX/17.0, 5.2/
DATA K/1, 2, 3, 3, 2, 1/
```

The variables JR, AT, AQ, and SRX are initially defined with the values 4, 5.0, 7.5, and 5.2, respectively. The variable NRX is initially defined with the value 17, after type conversion of the real 17.0 to the integer 17. The array K with six elements is initially defined with a value for each array element.

Example:

```
REAL R(10,10)
DATA R/50*5.0, 50*75.0/
```

The array R is initially defined with the first 50 elements set to the value 5.0 and the remaining 50 elements set to the value 75.0.

Example:

```
DIMENSION TQ(2)
EQUIVALENCE (RX, TQ(2))
DATA TQ(1)/32.0/
DATA RX/47.5/
```

The first element of array TQ is initially defined with the value 32.0. The variable RX and the second element of array TQ are initially defined as 47.5, since TQ(2) is equivalenced to variable RX.

Example:

```
BOOLEAN MASK
DATA MASK/Z"FFFF"/
```

The variable MASK is initially defined with the hexadecimal value 00...00FFFF hex.

For Better Performance

Use DATA statements, instead of assignment statements, to initialize arrays. Since DATA statements are evaluated at load time, they require no execution time.

Implied DO List in DATA Statement

You can use an implied DO list as an item in nlist. The implied DO list has the form:

(dlist, i=init, term, *incr)*

dlist

A list of array element names and implied DO lists. Subscript expressions must consist of extended integer constant expressions and active DO variables from dlist, except that the expression may contain implied DO variables of implied DO lists that have the subscript expression within their ranges.

i

An integer variable called the implied DO variable.

init

An extended integer constant expression specifying the initial value of i; may contain implied DO variables of other implied DO lists that have this DO list within their ranges.

term

An extended integer constant expression specifying the terminal value for i; may contain implied DO variables of other implied DO lists that have this DO list within their ranges.

incr

An optional extended integer constant expression specifying the increment for i; may contain implied DO variables of other implied DO lists that have this DO list within their ranges. Default value is 1.

An iteration count and the values of the implied DO variable are established from init, term, and incr just as for DO loops, except that the iteration count must be positive. When the implied DO list appears in a DATA statement, the list items in dlist are specified once for each iteration of the implied DO list, with appropriate substitution of values for any occurrence of the implied DO variable i.

The appearance of a name as an implied DO variable in a DATA statement does not affect the value or definition status of a variable with the same name in the program unit.

Example:

```
REAL X(5,5)
DATA ((X(J,I), I=1,J), J=1,5) /15 * 1.0/
```

Elements of array X are initially defined with the DATA statement. Elements in the lower diagonal part of the matrix are set to the value 1.0. The elements initialized are:

(1,1)				
(2,1)	(2,2)			
(3,1)	(3,2)	(3,3)		
(4,1)	(4,2)	(4,3)	(4,4)	
(5,1)	(5,2)	(5,3)	(5,4)	(5,5)

Example:

```
PARAMETER (PI=3.14159)
REAL Y(5,5)
DATA ((Y(J+1,I), J=I+1,4), I=1,3) /6 * PI/
```

The following array elements are initially defined with the value 3.14159:

(3,1)		
(4,1)	(4,2)	
(5,1)	(5,2)	(5,3)

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 spaces.

- 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 a space. Variable STR2 is set to 'FGH', with the fourth and fifth characters of the constant ignored. Example:

```
CHARACTER  STRING*6
DATA  STRING(2:5)/'BCDEF'/
```

The second through fifth positions of STRING are set to BCDE. The first and sixth positions of STRING are undefined, and the last character of the constant is ignored.

Expressions and Assignment Statements 4

This chapter describes the ways in which expressions are written and evaluated. This chapter also describes assignment statements. Assignment statements are executable statements that use expressions to define or redefine the values of variables.

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 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, or function references 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. Function references are described in chapter 7.

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

where aexp is an arithmetic expression in one of the forms:

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

where term is an arithmetic term in one of the forms:

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

where fact is an arithmetic factor in one of the forms:

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

where bprim is a boolean primary, as described for boolean expressions

where prim is 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 expression can be an unsigned arithmetic constant, symbolic arithmetic constant, arithmetic variable reference, arithmetic array 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 eight-, four-, and two-byte integer, eight- and 16-byte 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.

The length of an integer expression containing items of different lengths is the length of the longest item. For example, given the declarations

```
INTEGER A*2, B*2, C*4, D*4, E*8, F*8
BOOLEAN G
```

the following expression lengths exist:

Expression	Length	
A + B	2 bytes	
B + G	8 bytes	
C * D	4 bytes	Arithmetic overflow occurs if the result of C * D is larger than four bytes.
A + E	8 bytes	

The length of integer constants or integer constant expressions is determined by the value of the constant or constant expression. See integer constants in this chapter.

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.

If the exponent e in an arithmetic constant expression 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. Note that arithmetic constant expressions do not contain variable, array element, or function references.

Any arithmetic operation whose result is not mathematically defined will cause an error in the evaluation of an arithmetic expression. Examples are:

- Dividing by zero
- Raising a zero-valued primary to a zero-valued or negative-valued power
- Raising a negative-valued power to a real or double precision power

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[1]	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

[1] Valid only if the fixed collation weight table is in effect. (Collation sequences 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). The length of an integer constant expression is determined by the value of the constants in the expression. Note that variable, array element, and function references are not allowed in an integer constant expression.

Examples of integer constant expressions:

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

  -3     O"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[1]	IABS	INDEX	MAX0	NEQV
AND	ICHAR[2]	INT	MAX1	NINT
BOOL	IDIM	ISIGN	MIN[1]	OR
COMPL	IDINT	LEN	MIN0	SHIFT
DIM[1]	IDNINT	MASK	MIN1	SIGN[1]
EQV	IFIX	MAX[1]	MOD[1]	XOR

[1] Valid only with integer or boolean arguments.

[2] 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.

NOTE

The compiler may reorder individual operations in expressions to perform optimizations such as removal of repeated subexpressions. The reordering can cause unexpected results for real, double precision, and complex type data due to truncation errors. Only expressions that are mathematically associative or commutative are candidates for reordering. For example:

Source Expression	Compiler options
A/B*C	(A/B)*C
	(A*C)/B

You can inhibit reordering of operations of equal precedence by using parentheses or specifying EXPRESSION_EVALUATION=CANONICAL 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:

Table 4-1. Resulting Data Type for X1X2**

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	DBLE(x1)	x2	Double
Integer	Complex	CMPLX(REAL(x1),0.)	x2	Complex
Real	Integer	x1	x2	Real
Real	Real	x1	x2	Real
Real	Double	DBLE(x1)	x2	Double
Real	Complex	CMPLX(x1,0.)	x2	Complex
Double	Integer	x1	x2	Double
Double	Real	x1	DBLE(x2)	Double
Double	Double	x1	x2	Double
Double	Complex	CMPLX(SNGL(x1),0.)	x2	Complex
Complex	Integer	x1	x2	Complex
Complex	Real	x1	CMPLX(x2,0.)	Complex
Complex	Double	x1	CMPLX(SNGL(x2),0.)	Complex
Complex	Complex	x1	x2	Complex

Table 4-2. 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	DBLE(x1)	x2	Double
Integer	Complex	CMPLX(REAL(x1),0.)	x2	Complex
Real	Integer	x1	REAL(x2)	Real
Real	Real	x1	x2	Real
Real	Double	DBLE(x1)	x2	Double
Real	Complex	CMPLX(x1,0.)	x2	Complex
Double	Integer	x1	DBLE(x2)	Double
Double	Real	x1	DBLE(x2)	Double
Double	Double	x1	x2	Double
Double	Complex	CMPLX(SNGL(x1),0.)	x2	Complex
Complex	Integer	x1	CMPLX(REAL(x2),0.)	Complex
Complex	Real	x1	CMPLX(x2,0.)	Complex
Complex	Double	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.

NOTE

The following condition must be met in order to ensure the correct evaluation of a real, double precision, complex, or boolean expression: the operand must be a standard normalized floating point number or zero (represented as all zero bits) FORTRAN automatically normalizes all real non-zero constants, and the results of all floating point operations with standard normalized or zero operands are normalized or zero. However, it is possible to generate unnormalized or nonstandard operands by means of boolean expressions, equivalencing, or various input operations.

Character Expressions

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

cexp

where cexp is a character expression in one of the forms:

cprim
cexp // cprim

where cprim is 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

where // is the 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

where rexp is a relational expression in one of the forms:

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

where rop is 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

where aexp is an arithmetic expression

where bprim is a boolean primary, as described for boolean expressions

where cexp is 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 (assume that J and ITEM are type integer, and VAR, B, and C are type real):

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

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 J. Also refer to Collation Control in chapter 9.

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 spaces to the length of the longer operand.

NOTE

You should ensure that real operands in relational expressions involving the .EQ. or .NE. operators contain normalized standard floating point numbers or zero (represented as all zero bits), if a floating point comparison is desired. A bit-by-bit comparison is performed for these operators, in order to allow comparisons of Hollerith data in real variables. This means that two different unnormalized representations of the same floating point value will compare as unequal unless the EXPRESSION_EVALUATION parameter specifies MP on the FORTRAN command. FORTRAN automatically normalizes all real non-zero constants, and the results of all floating point operations with standard normalized or zero operands are normalized or zero. However, it is possible to generate unnormalized or nonstandard operands by means of boolean expressions, equivalencing, or various input operations.

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

where lexp is a logical expression in one of the forms:

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

where ldis is a logical disjunction in either of the forms:

lterm
ldis .OR. lterm

where lterm is a logical term in either of the forms:

lfact
lterm .AND. lfact

where lfact is a logical factor in either of the forms:

lprim
.NOT. lprim

where lprim is 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
Relational expression

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	Logical product of A and B
.OR.	Logical inclusive disjunction	A .OR. B	Logical 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	A .XOR. B	Logical 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:

1. .NOT. (Highest)
2. .AND.
3. .OR.
4. .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.

Control Data Extension

Boolean Expressions

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

where bexp is 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

where bdis is a boolean disjunct in one of the forms:

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

where bterm is a boolean term in one of the forms:

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

where bfact is a boolean factor in one of the forms:

bprim
.NOT. bprim
.NOT. aexp

where aexp is an arithmetic expression

where bprim is a boolean primary; one of the following:

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

Examples (P, Q, R, S, X, Y and Z can be any type except character or logical):

```
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. Boolean expressions containing two- or four-byte integers are considered eight bytes long. 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

End of Control Data Extension

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:

1. Arithmetic (Highest)
2. Character
3. Relational
4. 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 an assigned 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 consists of a variable followed by an equal sign followed by an expression. (In the case of multiple assignment statements, multiple equal signs and multiple variables 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

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. If the type of v is integer, and the size of v is less than the size of exp, then exp is truncated on the left to match the size of v. The truncation can change the sign of exp.

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.

For Better Performance

Use DATA statements, instead of assignment statements, to initialize arrays. Since DATA statements are evaluated at load time, they require no execution time.

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. If the character positions being defined in v are referenced in exp, results can be incorrect unless you specify EXPRESSION_EVALUATION=OVERLAPPING_STRING_MOVES on the FORTRAN command.

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 spaces 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 exp 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.

Control Data Extension

Boolean Assignment Statement

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.

End of Control Data Extension

Control Data Extension

Multiple Assignment Statement

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. If the type of v is integer, and the size of v is less than the size of exp, then exp is truncated on the left to match the size of v. The truncation can change the sign of exp.

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)
```

Example:

```
X=M=1.5
```

M is assigned the value 1 and X is assigned the value 1.0.

End of Control Data Extension

Flow Control Statements 5

Flow control statements provide a means of altering, interrupting, terminating, or otherwise modifying the normal sequential flow of execution.

Flow Control Statements 5

The flow control statements change the order in which statements are executed. The flow control statements are:

GO TO

IF

DO

CONTINUE

PAUSE

STOP

END

The CALL and RETURN statements, while sometimes considered to be flow control statements, are described in chapter 7 along with program units.

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 (or the END or ERR specifiers on the input/output statements).

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 of 8 bytes in length.

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 whose label was 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 of 8 bytes in length. 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 value of variable iv must be 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 the statement following 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.

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 of these if-blocks is executed. Each logical expression is evaluated in order of the source statements 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 (expl) 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.

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 that cannot equal zero; default is 1. If exp3 is omitted, the preceding comma must also be omitted.

The parameters 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. You should not alter the value of the DO varable during execution of a DO loop.

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. The incrementation parameter, m3, must not equal zero.
mtc	Minimum trip count; in NOS/VE 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 in ONE_TRIP_DO mode executes the loop one time. In ANSI FORTRAN, mtc has a value of zero.

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 final 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. Execution then continues with the statement following the final statement of the loop.

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

| m2 - m1 | < (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 following the DO loop. 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 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. For instance, if IVAR=30 and I=30, the branch to the statement labeled 10 is executed and I retains the value 30. However, if IVAR is greater than 200, the statements following the IF statement are executed sequentially and I contains the value 201.

For Better Performance

Multidimensional arrays in DO loops should be used with care. Whenever possible, the innermost loop should iterate over the first subscript, the next innermost loop should iterate over the second subscript, and so forth. This is because each reference to the array is made to the next closest array element (arrays are stored in column major order). For example, the following statements do not reference elements in the order they are stored:

```
      DIMENSION A(20,30,40),B(20,30,40)
        :
      DO 10 I=1,20
      DO 10 J=1,30
      DO 10 K=1,40
   10 A(I, J, K) = B(I, J, K)
```

The following example ensures that elements are referenced in the order they are stored and therefore executes faster:

```
      DIMENSION A(20,30,40), B(20,30,40)
        :
      DO 10 K=1,40
      DO 10 J=1,30
      DO 10 I=1,20
   10 A(I, J, K) = B(I, J, K)
```

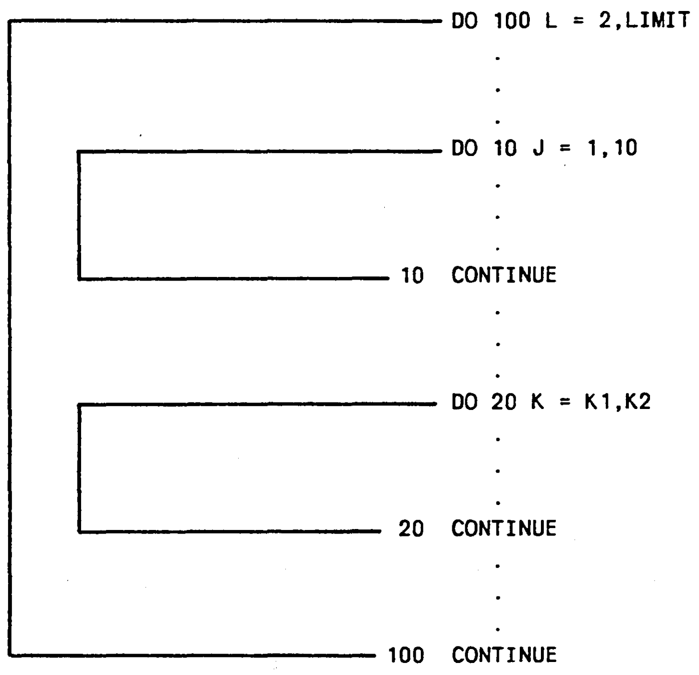
Nested DO Loops

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 loop.

The final statement of an inner DO loop must be either the same as the final statement of any containing 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) of the innermost DO.

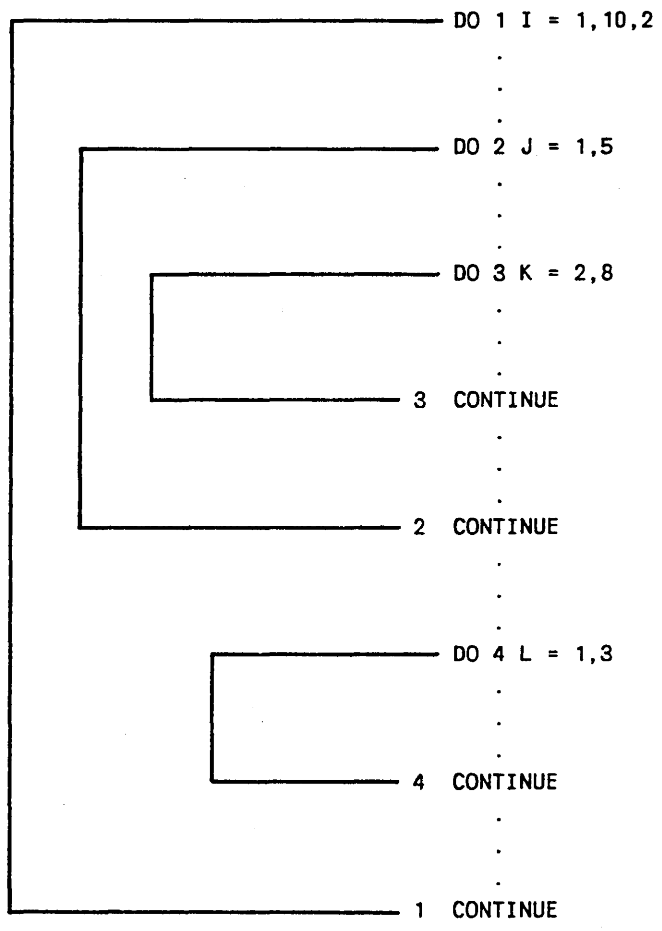
The following examples show some possible DO loop nests. Note that loops can be completely nested or can share a final statement.

Example 1



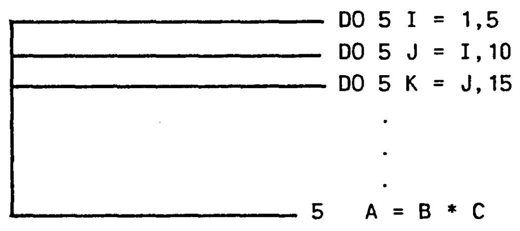
```
      DO 100 L = 2,LIMIT
         .
         .
         .
      DO 10 J = 1,10
         .
         .
         .
   10 CONTINUE
         .
         .
         .
      DO 20 K = K1,K2
         .
         .
         .
   20 CONTINUE
         .
         .
         .
  100 CONTINUE
```

Example 2



```
      DO 1 I = 1,10,2
         .
         .
         .
      DO 2 J = 1,5
         .
         .
         .
      DO 3 K = 2,8
         .
         .
         .
    3 CONTINUE
         .
         .
         .
    2 CONTINUE
         .
         .
         .
      DO 4 L = 1,3
         .
         .
         .
    4 CONTINUE
         .
         .
         .
    1 CONTINUE
```

Example 3



```
DO 5 I = 1,5
DO 5 J = I,10
DO 5 K = J,15
    .
    .
    .
5   A = B * C
```

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
```

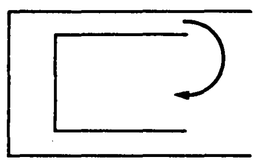
A DO loop can be initially entered only through the DO statement. However, a loop can be reentered from its extended range (CDC extension; see Extended Range DO Loops).

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,

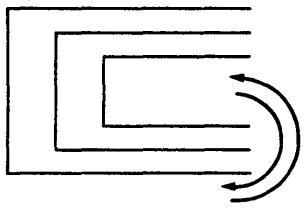
```
      DO 10 I=1,100
      IF (I .GT. IVAR) GO TO 10
         DO 20 J=1,10
         .
         .
         .
20       CONTINUE
10    CONTINUE
```

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

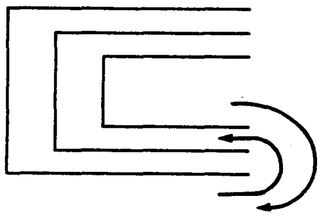
A transfer from an outer DO loop into an inner DO loop is not allowed, unless the transfer is from the extended range of the inner loop. However, a transfer from an inner DO into an outer DO is allowed because such a transfer is still within the range of the outer DO loop. A transfer back into an inner 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:



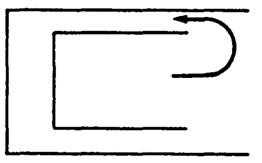
Illegal; you cannot transfer from an outer loop to an inner loop unless the transfer is from the extended range of the inner loop.



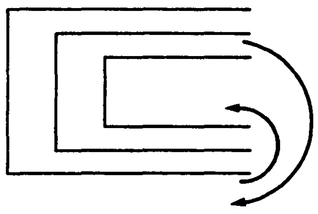
Legal for NOS/VE FORTRAN only; you can transfer back into an inner or outer loop from the extended range of the inner loop.



Legal for NOS/VE FORTRAN only; you can transfer back into an inner or outer loop from the extended range of the inner loop.



Legal; you can transfer from an inner loop to an outer loop.



Illegal; you cannot transfer back into an inner loop from the extended range of an outer loop.

Control Data Extension

Extended Range DO Loops

Under certain conditions, control can be transferred out of a loop and can subsequently be transferred 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:

1. The DO variable is not redefined outside the loop.
2. The program unit containing the loop has not been exited by a RETURN, STOP, or END statement.

3. 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.

End of Control Data Extension

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 70 characters, 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 70 characters, 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
```

Input/Output 6

This chapter describes the various categories of input and output operations that can be performed in a FORTRAN program, and the statements used to perform those operations.

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.

Sequential Acess

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

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 keyed-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 keyed-file interface subprograms, along with the background information required to use those subprograms, are described in chapter 11.

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.

Control Data Extension

Segment Access Files

Segment access files allow fast and efficient access to large blocks of data. These files are associated, or mapped, with a named common block. Values are accessed and modified through FORTRAN assignment statements rather than input/output statements. This association is accomplished by using the C$ SEGFILE directive and by using the common block name as the UNIT specifier in the OPEN, CLOSE, and INQUIRE statements.

End of Control Data Extension

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 associate a segment access file with a common block name by using the common block name as the UNIT specifier (including slashes). For example,

```
      COMMON /CHBLK/CH(1000)
C$    SEGFILE (/CHBLK/)
      OPEN (UNIT=/CHBLK/, FILE='SEG1')
```

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. To terminate data entry, enter *EOI (must be uppercase) immediately after the promt. 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 using the CREATE_FILE_CONNECTION command. For example:

```
          PROGRAM S
          READ (*,100)
    100   FORMAT (2I3)
```

The READ statement reads from $INPUT, which has the effect shown in the above table. You can change the connection to read from another file as follows:

```
    CREFC $INPUT MYFILE
```

The above program will now read from file MYFILE.

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.

When fixed-length records (RT=F) are being written to a sequential file, and attempt is made to write more characters to a record that the record length allows, the record is truncated and no warning message is issued.

The only indication that truncation occurred is given when the truncated record is subsequently read. For sequential unformatted input/output, an error message is issued when an attempt is made to read more characters than the record contains. For sequential formatted input/output, the truncated portion of the record is read as blanks. In this case, no message explicitly indicating this specific data item is issued.

An attempt to exceed the record length of a direct access file results in an error message.

For Better Performance

Input/output operations now execute faster due to improved internal processing. The benefits are most noticed with buffered, direct-access, and sequential input/output. However, the improved execution can cause different behavior during input/output processing. See Input/Output in Appendix E for more information.

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.
- The name of a common block (including slashes) when used to associate the common block with a segment access file.
- An integer or boolean expression having one of 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.
  - 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.

When unit is an integer expression and it is passd to an input/output related subroutine or function (such as UNIT, LENGTH, or CONNEC), it must be a full-word (8 byte) integer.

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 or array element 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	Either a FORTRAN error number in the range 1 through 9999 or another product's status condition code in integer form that includes the product's identifier encoded with its condition number.

All runtime errors under NOS/VE are identified by a unique status condition code that is an integer formed by combining the ASCII equivalent of the two-character product identifier with a condition number. Condition numbers within the range 1 through 9999 are reserved for Control Data defined errors.

A FORTRAN error number is a FORTRAN condition code without the encoded product identifier 'FL'. Errors are listed by error number and condition name in the Diagnostic Messages for NOS/VE manual, which provides descriptions and suggested action for the errors.

To determine the condition name of another product's condition code, use the SCL function $CONDITION_NAME with the returned condition code. See the SCL Language Definition Usage and SCL System Interface manuals for more information.

iolist

Input/output list specifying items to be transmitted (described in the next section under

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, I2, 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 I2.

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=expl, exp2, *exp3*)

dlist

A list of input/output items.

i

Established the same way as for DO variables.

expl,exp2,*exp3*

Established the same way as for DO loop indexing parameters.

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 expl, 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 the sequence CAT, DOG, 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:

```
    WRITE(2,200)((I,E(I,J),J=1,3),I=1,3)
200 FORMAT (1X,I1,9F7.2)
```

This sequence writes the variable I and nine elements from 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*

*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 one of 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.
  - 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.

When unit is an integer expression and it is passd to an input/output related subroutine or function (such as UNIT, LENGTH, or CONNEC), it must be a full-word (8 byte) integer.

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.

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=.

IOSTAT=ios

Specifies an integer variable or array element 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	Either a FORTRAN error number in the range 1 through 9999 or another product's status condition code in integer form that includes the product's identifier encoded with its condition number.

All runtime errors under NOS/VE are identified by a unique status condition code that is an integer formed by combining the ASCII equivalent of the two-character product identifier with a condition number. Condition numbers within the range 1 through 9999 are reserved for Control Data defined errors.

A FORTRAN error number is a FORTRAN condition code without the encoded product identifier 'FL'. Errors are listed by error number and condition name in the Diagnostic Messages for NOS/VE manual, which provides descriptions and suggested action for the errors.

To determine the condition name of another product's condition code, use the SCL function $CONDITION_NAME with the returned condition code. See the SCL Language Definition Usage and SCL System Interface manuals for more information.

ERR=sl

Specifies the label of an executable statement to which control transfers if an error condition is encountered during input/output processing.

END=sl

Specifies the label of an executable statement to which control transfers when an end-of-file is encountered during an input operation.

iolist

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.

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 first form of the READ statement 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

```
    TEAD (5, 100) (VEC(I), I=1,10)
100 FORMAT (5F7.2)
```

reads data (consisting of five data items 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 from the file associated with unit 2 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*

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 one of 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.
  - 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.

When unit is an integer expression and it is passd to an input/output related subroutine or function (such as UNIT, LENGTH, or CONNEC), it must be a full-word (8 byte) integer.

*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.

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=.

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 or array element 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	Either a FORTRAN error number in the range 1 through 9999 or another product's status condition code in integer form that includes the product's identifier encoded with its condition number.

All runtime errors under NOS/VE are identified by a unique status condition code that is an integer formed by combining the ASCII equivalent of the two-character product identifier with a condition number. Condition numbers within the range 1 through 9999 are reserved for Control Data defined errors.

A FORTRAN error number is a FORTRAN condition code without the encoded product identifier 'FL'. Errors are listed by error number and condition name in the Diagnostic Messages for NOS/VE manual, which provides descriptions and suggested action for the errors.

To determine the condition name of another product's condition code, use the SCL function $CONDITION_NAME with the returned condition code. See the SCL Language Definition Usage and SCL System Interface manuals for more information.

iolist

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 formatted output, only information completely contained within the FORMAT statement, such as character strings, 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 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 (and no H, quote, or apostrophe edit descriptors are specified), 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. The default association of unit OUTPUT is with file $OUTPUT. This statement has the form:

PRINT fn, *iolist*

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.

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=.

iolist

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 formatted output, only information completely contained within the FORMAT statement, such as character strings, 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).

Example:

```
PRINT 4, A, B, N
```

This statement transfers data from A, B, and N to unit OUTPUT according to the format statement labeled 4.

Example:

```
PRINT '(3E14.4)', X1, X2, X3
```

This statement transfers data from X1, X2, and X3 to unit OUTPUT according to the format specification 3E14.4.

Control Data Extension

Formatted PUNCH

The PUNCH statement transfers data from the specified storage locations to the unit PUNCH. The default association of unit PUNCH is with file PUNCH. This statement has the form:

PUNCH fn, *iolist*

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.

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=.

iolist

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 formatted output, only information completely contained within the FORMAT statement, such as character strings, 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).

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.

End of Control Data Extension

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.
- The name of a namelist group.

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, I2, F7.2)
```

The format specification consists of a sequence of edit descriptors enclosed in parentheses. Spaces 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 single precision, floating-point (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. A D descriptor is permitted for a complex input item, but the transferred value is truncated to single precision.

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 spaces. 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,2I2,1','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, 2I2, 10X, F6.2)
```

Control Data Extension

Noncharacter Format Specification

You can place format specifications in a noncharacter array. If the array is of type integer, each element must be a full-word (8 byte) integer. The rules for noncharacter format specifications are the same as facter format specifications.

End of Control Data Extension

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	Spaces ignored.
BZ		Spaces 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 spaces, + 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 no more items are in the iolist.)

Leading spaces are not significant in numeric input conversions; other spaces 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-spaces field is considered to be zero, except for logical input, where an all-spaces field is considered to have the logical value false, or for character input.

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 spaces 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 a numeric 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 spaces. 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:

```
EXAMPLEΔΔ
```

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:

```
THISΔISΔ| ANΔEXAMPLEΔI|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 spaces 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|586.25|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 spaces.

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. If the exponent exceeds 999 in magnitude, a field of asterisks is written.

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 space; 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 spaces 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

An exponent subfield alone is not permitted. 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 spaces 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.
spaces	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 spaces. 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 spaces 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 a real variable containing 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	ΔΔΔΔ.673E+02
-67.32	E12.3	ΔΔΔ-.673E+02

F Descriptor

The F descriptor specifies conversion between an internal real or double precision number and 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.
spaces	F5.2	0	Spaces 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 spaces 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	ΔΔΔ32,694
−32.694	F6.3	******
.32694	F4.3	.327
32.694	F6.0	ΔΔΔ33.

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 in magnitude, the number is output under modified F format as shown below. For values outside this range, Gw.d output is identical to Ew.d, and Gw.dEe is identical to Ew.dEe.

Size of N	F format conversion if Gw.d is used	F format conversion if Gw.dEe is used
0.1≤N<1	F(w−4).d 'ΔΔΔΔ'	F(w−e−2).d, e + 2('Δ')
1≤N<10	F(w−4).(d−1) 'ΔΔΔΔ'	F(w−e−2).(d+1) e + 2('Δ')
10**(d−2)≤N<10**(d−1)	F(w−4).1 'ΔΔΔΔ'	F(w−e−2).1 e + 2('Δ')
10**(d−1)≤N<10**d	F(w−4).0 'ΔΔΔΔ'	F(w−e−2).0 e + 2('Δ')

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 spaces within a field depends on the BLANK= specifier in the OPEN statement (described later in this chapter). If this specifier is omitted, spaces are ignored. An all-blank field is always considered to be zero. Decimal points are not permitted. Any character other than a decimal digit, space, or the leading plus or minus sign in an integer field on input will cause an error.

are ignored (BLANK= specifier omitted from OPEN statement, or BLANK='NULL' specified). If BLANK='ZERO' were specified, spaces 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, I2, I3, I2, I4)
```

Input Record:

```
139|ΔΔ-15ΔΔ|18|ΔΔ7|ΔΔ|1Δ4          (Bars mark input fields.)
```

In Storage:

I:	139
J:	−15
K:	18
L:	7
M:	0
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 spaces. If m=w, no spaces 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
J: 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 spaces 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
```

Control Data Extension

O Descriptor

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. Spaces are allowed and a plus or minus sign can precede the first octal digit. Spaces are interpreted as zeros and an all spaces 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|666Δ6644Δ444|-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 spaces 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 spaces 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 spaces. If m is specified, up to w−m leading zeros are written as spaces. If the number cannot be output in w octal digits, the rightmost w*3 are converted to octal and written.

End of Control Data Extension

Control Data Extension

R Descriptor

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 input, 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 are read and stored right-justified with upper binary zero fill.

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:

```
RESULTSΔ|OFΔTEST
```

In Storage:

HOO: RESULTSΔ
RAY: 0OFΔTEST (Leftmost character is ASCII null, not ASCII zero)

End of Control Data Extension

Control Data Extension

Z Descriptor

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 spaces 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:

```
A309FFFFCC4ΔD1
```

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 spaces on the left.

If w is greater than 32 for complex or double precision list items, the 32 hexadecimal digits are right-justified with spaces on the left.

The value of m is used to cause leading zeros to be written as spaces. If m is specified, up to w−m leading zeros are written as spaces. If the number cannot be written in w hexadecimal digits, a field of asterisks is written.

End of Control Data Extension

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
P E20.4	-3.1416E+00
3P E20.4	-314.16E-02

For G editing, the effect of the scale factor is nullified 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 spaces (other than leading spaces). In the absence of a BN or BZ descriptor, spaces 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 spaces in succeeding numeric input fields are ignored; that is, the field is treated as if spaces had been removed, the remaining portion of the field right-justified, and the field padded with leading spaces. A field of all spaces has a value of zero.

If a BZ descriptor is encountered in a format specification, all spacesin 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

```
1ΔΔ2ΔΔ3ΔΔ
```

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 spaces.

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 spaces; 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 spaces; 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 space 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 FILE_CONTENTS parameter on the SET_FILE_ATTRIBUTE command allows you to specify whether printer control characters are to be recognized or disregarded. The SET_FILE_ATTRIBUTE command is described in appendix E.

The printer control characters are shown the following table.

Table 6-3. Printer Control Characters

Character	Action
Space	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 System Interface 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 (1H0, F7.3, I2, 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*

*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 one of 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.
  - 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.

When unit is an integer expression and it is passd to an input/output related subroutine or function (such as UNIT, LENGTH, or CONNEC), it must be a full-word (8 byte) integer.

IOSTAT=ios

Specifies an integer variable or array element 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	Either a FORTRAN error number in the range 1 through 9999 or another product's status condition code in integer form that includes the product's identifier encoded with its condition number.

All runtime errors under NOS/VE are identified by a unique status condition code that is an integer formed by combining the ASCII equivalent of the two-character product identifier with a condition number. Condition numbers within the range 1 through 9999 are reserved for Control Data defined errors.

A FORTRAN error number is a FORTRAN condition code without the encoded product identifier 'FL'. Errors are listed by error number and condition name in the Diagnostic Messages for NOS/VE manual, which provides descriptions and suggested action for the errors.

To determine the condition name of another product's condition code, use the SCL function $CONDITION_NAME with the returned condition code. See the SCL Language Definition Usage and SCL System Interface manuals for more information.

ERR=sl

Specifies the label of an executable statement to which control transfers if an error condition is encountered during input/output processing.

iolist

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 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).

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*

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 one of 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.

  - 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.

When unit is an integer expression and it is passd to an input/output related subroutine or function (such as UNIT, LENGTH, or CONNEC), it must be a full-word (8 byte) integer.

IOSTAT=ios

Specifies an integer variable or array element 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	Either a FORTRAN error number in the range 1 through 9999 or another product's status condition code in integer form that includes the product's identifier encoded with its condition number.

All runtime errors under NOS/VE are identified by a unique status condition code that is an integer formed by combining the ASCII equivalent of the two-character product identifier with a condition number. Condition numbers within the range 1 through 9999 are reserved for Control Data defined errors.

A FORTRAN error number is a FORTRAN condition code without the encoded product identifier 'FL'. Errors are listed by error number and condition name in the Diagnostic Messages for NOS/VE manual, which provides descriptions and suggested action for the errors.

To determine the condition name of another product's condition code, use the SCL function $CONDITION_NAME with the returned condition code. See the SCL Language Definition Usage and SCL System Interface manuals for more information.

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.

iolist

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.

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).

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, END=sl) iolist*

READ *, iolist

*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 one of 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.
  - 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.

When unit is an integer expression and it is passd to an input/output related subroutine or function (such as UNIT, LENGTH, or CONNEC), it must be a full-word (8 byte) integer.

*FMT=**

Specifies a list-directed format. The asterisk (*) is required.

IOSTAT=ios

Specifies an integer variable or array element 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	Either a FORTRAN error number in the range 1 through 9999 or another product's status condition code in integer form that includes the product's identifier encoded with its condition number.

All runtime errors under NOS/VE are identified by a unique status condition code that is an integer formed by combining the ASCII equivalent of the two-character product identifier with a condition number. Condition numbers within the range 1 through 9999 are reserved for Control Data defined errors.

A FORTRAN error number is a FORTRAN condition code without the encoded product identifier 'FL'. Errors are listed by error number and condition name in the Diagnostic Messages for NOS/VE manual, which provides descriptions and suggested action for the errors.

To determine the condition name of another product's condition code, use the SCL function $CONDITION_NAME with the returned condition code. See the SCL Language Definition Usage and SCL System Interface manuals for more information.

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.

iolist

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.

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).

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 spaces, or by a comma or slash, either of which can be preceded or followed by any number of spaces. 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 spaces 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. Two-byte and four-byte integers are allowed; values that are too large for an associated specification cause a runtime error.

Real numbers

Any valid FORTRAN format for real or double pecision numbers. Real*16 values are allowed. 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 spaces.

Character values

A string of characters (which can include spaces) 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, spaces, 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
```

Spaces 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 spaces. 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. Integer input values must not be too large for an associated iolist item. For example, if iolist specifies a variable typed as INTEGER*2, then the associated item in the input record must be within the range -32,768 to 32,767. 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)

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 one of 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.
  - 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.

When unit is an integer expression and it is passd to an input/output related subroutine or function (such as UNIT, LENGTH, or CONNEC), it must be a full-word (8 byte) integer.

*FMT=**

Specifies a list-directed format. The asterisk (*) is required.

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=.

IOSTAT=ios

Specifies an integer variable or array element 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	Either a FORTRAN error number in the range 1 through 9999 or another product's status condition code in integer form that includes the product's identifier encoded with its condition number.

All runtime errors under NOS/VE are identified by a unique status condition code that is an integer formed by combining the ASCII equivalent of the two-character product identifier with a condition number. Condition numbers within the range 1 through 9999 are reserved for Control Data defined errors.

A FORTRAN error number is a FORTRAN condition code without the encoded product identifier 'FL'. Errors are listed by error number and condition name in the Diagnostic Messages for NOS/VE manual, which provides descriptions and suggested action for the errors.

To determine the condition name of another product's condition code, use the SCL function $CONDITION_NAME with the returned condition code. See the SCL Language Definition Usage and SCL System Interface manuals for more information.

ERR=sl

Specifies the label of an executable statement to which control transfers if an error condition is encountered during input/output processing.

iolist

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 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).

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.

List directed output statements always produce a space for printer control as the first character of the output record. The maximum length of an output line is the smaller of the PAGE_WIDTH and MAXIMUM_RECORD_LENGTH attribute values of the file. (These attributes can be set by the SET_FILE_ATTRIBUTE command, which is described in appendix E.)

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
```

Control Data Extension

Namelist Input/Output

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.

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. For integer variables or arrays, only full-word (eight-byte) values are allowed in the list.

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.

End of Control Data Extension

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***)**

*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 one of 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.
  - 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.

When unit is an integer expression and it is passd to an input/output related subroutine or function (such as UNIT, LENGTH, or CONNEC), it must be a full-word (8 byte) integer.

FMT=**name**

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=.

IOSTAT=ios

Specifies an integer variable or array element 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	Either a FORTRAN error number in the range 1 through 9999 or another product's status condition code in integer form that includes the product's identifier encoded with its condition number.

All runtime errors under NOS/VE are identified by a unique status condition code that is an integer formed by combining the ASCII equivalent of the two-character product identifier with a condition number. Condition numbers within the range 1 through 9999 are reserved for Control Data defined errors.

A FORTRAN error number is a FORTRAN condition code without the encoded product identifier 'FL'. Errors are listed by error number and condition name in the Diagnostic Messages for NOS/VE manual, which provides descriptions and suggested action for the errors.

To determine the condition name of another product's condition code, use the SCL function $CONDITION_NAME with the returned condition code. See the SCL Language Definition Usage and SCL System Interface manuals for more information.

ERR=sl

Specifies the label of an executable statement to which control transfers if an error condition is encountered during input/output processing.

END=sl

Specifies the label of an executable statement to which control transfers when an end-of-file is encountered during an input operation.

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.

An & 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.

Spaces must not appear:

Between $ (or &) and the group name

Within array names and variable names

Spaces 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. Each namelist group must begin in a new record. 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.

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.

Spaces 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 spaces, a fatal error results.

The following example illustrates a namelist input group:

`$AGRP`	Group name
`XVAL=5.0,`	Real number
`ARR=5*(1.7, 2.4),`	Five complex numbers
`CHAR='HI THERE',`	Character string
`$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

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.
- b integer or boolean expression having one of 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.
  - 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.

When unit is an integer expression and it is passd to an input/output related subroutine or function (such as UNIT, LENGTH, or CONNEC), it must be a full-word (8 byte) integer.

FMT=**name**

Specifies a namelist group name.

IOSTAT=ios

Specifies an integer variable or array element 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	Either a FORTRAN error number in the range 1 through 9999 or another product's status condition code in integer form that includes the product's identifier encoded with its condition number.

All runtime errors under NOS/VE are identified by a unique status condition code that is an integer formed by combining the ASCII equivalent of the two-character product identifier with a condition number. Condition numbers within the range 1 through 9999 are reserved for Control Data defined errors.

A FORTRAN error number is a FORTRAN condition code without the encoded product identifier 'FL'. Errors are listed by error number and condition name in the Diagnostic Messages for NOS/VE manual, which provides descriptions and suggested action for the errors.

To determine the condition name of another product's condition code, use the SCL function $CONDITION_NAME with the returned condition code. See the SCL

ERR=sl

Specifies the label of an executable statement to which control transfers if an error condition is encountered during input/output processing. Language Definition Usage and SCL System Interface manuals for more information.

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 output line is the smaller of the PAGE_WIDTH and MAXIMUM_RECORD_LENGTH attribute values of the file. (These attributes are described in appendix E.) 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

Control Data Extension

Buffer Input/Output

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 and type INTEGER, 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. Integer values must be of size integer*8.

b

Last variable or array element of the block of memory to which data is to be transmitted; cannot be type character. Integer values must be of size integer*8.

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 or by LENGTHB in terms of bytes.

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 and type integer, 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. Integer values must be full-word (8 byte) integers.

b

Last variable or array element of the block of memory from which data is to be transmitted; cannot be type character. Integer values must be full-word (8 byte) integers.

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 block of memory and fwa is the first word address. For fixed-length records (RECORD_TYPE=FIXED attribute), the record length is the length (number of bytes) 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.

Following a BUFFER OUT, the UNIT function must be referenced before another reference is made to the file or to the contents of the block of memory.

End of Control Data Extension

Control Data Extension

Mass Storage Input/Output

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.

All integer arguments to the mass storage routines must be full-word (8 byte) integers.

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 READ and WRITE, mass storage I/O routines, and the keyed-file interface 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 specify 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. 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 should be noncharacter. Integerarrays must be full-word integer arrays, that is, typed as INTEGER*8. The array becomes the directory, or index, to the file contents. In addition to the record key information, 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. Before using the array as a subindex, you must set the array elements to zero 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 assigned index array is automatically cleared. 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 mass storage. 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.

Index Key Types

There are two types of index keys: name and number. A name key can be specified as any nonzero boolean or full-word (8 byte) integer expression. A number key must be expressed as a full-word (8 byte) 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.

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 (in words) 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:

$1 \leq k \leq \text{len}-1$

where len is the length of the master index. (See OPENMS call.)

For name index, k can be any character, 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:

0	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.

This argument can be omitted, in which case the index has the same type as the previously-specified 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.

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,-1)
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
```

Figure 6-1. Program MS1

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
```

Figure 6-2. Program MS3

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. Then, one record indexed under the second subindex is read.

```
      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
```

Figure 6-3. Program MS4

End of Control Data Extension

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 circumvent these limitations, the FORTRAN READ and WRITE statements have 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 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.

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.

You cannot use internal files for direct access input/output.

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, the record 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 is specified with the REC= specifier in a READ or WRITE statement.

Unlike other methods of random access, direct access files do not use a record key index. The disk address of the records in a direct access file are based on 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.)

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 space 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.

Direct Access File Creation

To create a direct access, file you must specify an OPEN statement with the ACCESS='DIRECT' option. You must also specify the record length with the RECL= specifier in the OPEN statement. For example, the following statement opens an unformatted file named DAFL for direct access:

```
   OPEN (2, FILE='DAFL', ACCESS='DIRECT', RECL=120)
```

The file is associated with unit 2 and has a record length of 120 words.

Direct Access File Examples

The following example writes variables A, B, and C to record number 6, and variables I, J, and X to record number 1 of the direct access file associated with unit 2:

```
WRITE (2, '(3E10.4)', REC=6) A, B, C
WRITE (2, '(2I4, G20.10)', REC=1) I, J, X
```

The following example reads records 10, 8, 6, 4, and 2 from direct access file DARG:

```
      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.

Direct Access Record Length Calculation

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 characters
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, array element 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 AR 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. (The format specification I3 is repeated for each variable.)

Example:

```
CHARACTER*8 FMT
DATA FMT /'(ΔΔF8.2)'/
  :
WRITE (FMT(2:3), '(I2)') 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 (ΔΔF8.2). 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, array element 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. (The format specification A3 is repeated for each variable.)

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.

Control Data Extension

Extended Internal Files

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. For integer type data, only full-word (8 byte) integers are allowed. 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

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 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. For integer type data, only full-word (8 byte) integers are allowed.

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 (4X, 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.

End of Control Data Extension

Control Data Extension

Segment Access Files (CDC Extension)

The segment access file capability provides an efficient means of sharing large, randomly accessed blocks of data among applications. A named common block is associated (mapped) to a segment access file. You access the file directly through the common block's variables and arrays using assignment statements. The variables and arrays in the common block represent locations within the file.

You associate a common block with a segment access file by first specifying the common block name in a C$ SEGFILE directive (see appendix D) and then using the OPEN statement specifying the common block name as the UNIT specifier. When using the OPEN statement to associate a file with a common block, the UNIT and FILE specifiers are both required.

The UNIT specifier must specify the name of the common block, including slashes. The FILE specifier must specify the name of the segment access file you wish to associate with the common block. The IOSTAT, ERR and STATUS specifiers are optional. All other specifiers for the OPEN statement are not valid with segment access files.

You may not reference a segment access file until after the OPEN statement has opened and associated the file with a common block. Therefore, you should initialize values in segment access files after the OPEN statement, rather than in a DATA statement, since the OPEN statement will change the values when it is executed.

Segment access files that are associated with a common block are closed using the CLOSE statement. The SIZE parameter is an optional specifier on the CLOSE statement that can be used to specify the length of the segment access file in bytes.

If a segment access file is specified on an INQUIRE statement, then only the IOSTAT, ERR, EXIST, OPENED, NAMED and NAME specifiers may be used. The NAME specifier will return the common block name and the NAMED specifier will always be true.

To create and open a segment access file, specify the common block name as the UNIT specifier and the segment access file name as the FILE specifier (the FILE specifier is required for segment access files). For example:

```
      CHARACTER CH*5000
      COMMON /CHBLK/ CH
C$    SEGFILE (/CHBLK/)
      OPEN (UNIT=/CHBLK/, FILE='SEG_FILE_1', ERR=10, IOSTAT=IVAR)
      CH(1:1)='A'
```

This creates and opens file SEG_FILE_1 and associates the file with common block CHBLK. The assignment statement will set the first byte of the file to the value of the character A.

For Better Performance

Segment access input/output is faster than other types of input/output.Your program may automatically use segment access files on a file that is not shared and is generally used or attached with default values. See Fast I/O in appendix E for more information.

End of Control Data Extension

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. For a segment access file, the unit identifier specifies the named common block (including slashes) to be associated with the FILE specifier.

ios

Full-word (8 byte) variable or array element that receives an error number if an error occurs during the open; zero indicates that no error occurred. More information about ios is given under Input/Output statement specifiers.

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 local file 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 $LOCAL catalog. You can use the C$ PARAM command to access files that do not 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.)

This specifier is not valid for segment access files.

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.

This specifier is not valid for segment access files.

rl

Positive full-word (8 byte) variable or constant specifying the maximum 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.

This specifier is not valid for segment access files.

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.

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'

This specifier is not valid for segment access files.

bl

This parameter is provided only for compatibility with previous versions of FORTRAN, and is disregarded.

Example:

```
OPEN (UNIT=2, FILE='AAA', ACCESS='DIRECT', RECL=120)
```

This statement opens file AAA for direct access input/output and associates AAA with unit 2.

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.

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 and you must specify the FILE specifier to open a segment access file. 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 file. For example, a MAXIMUM_RECORD_LENGTH 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.

To create and open a segment access file, specify the common block name as the UNIT specifier and the segment access file name as the FILE specifier (the FILE specifier is required for segment access files). For example:

```
      CHARACTER CH*5000
      COMMON /CHBLK/ CH
C$    SEGFILE (/CHBLK/)
      OPEN (UNIT=/CHBLK/, FILE='SEG_FILE_1', ERR=10, IOSTAT=IVAR)
      CH(1:1)='A'
```

This creates and opens file SEG_FILE_1 and associates the file with the COMMON block CHBLK. The assignment statement will set the first byte of the file to the value of character A.

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*, *SIZE*=*n*)

u

Unit identifier of the file to be closed. For a segment access file, the unit identifier specifies the named common block (including slashes).

ios

Full-word (8 byte) integer variable or array element in which, upon completion of the close, contains an error number if an error occurrs; zero indicates no error occurred. More information about ios is given under Input/Output Statement Specifiers.

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'.

n

Full-word (8 byte) integer expression specifying the length of a segment access file in bytes. Default is the size of the segment access file before an association was made; the default size will be extended if a reference is made beyond the current size (this will usually be the case if the file is being created). This specifier is optional and used only with segment access files.

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. For a segment access file, the unit identifier specifies the named common block (including slashes).

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

Full-word (8 byte) integer variable or array element which, upon completion of the INQUIRE, contains an error number; contains zero if no error occurred. More information about ios is given under Input/Output Statement Specifiers.

sl

Label of an executable statement to which control passes if an error occurs during an inquire.

ex

Logical variable or array element 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 or array element 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

Full-word (8 byte) integer variable or array element 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 or array element 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 open for sequential access input/output.

'DIRECT'

The file is open for direct access input/output.

If the file is not open, acc is undefined.

seq

A character variable or array element 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 or array element 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 or array element indicating whether the file is opened for formatted or unformatted input/output:

'FORMATTED'

The file is open for formatted input/output.

'UNFORMATTED'

The file is open for unformatted input/output.

'BUFFERED'

The file is open for buffered input/output. (CDC Extension)

If the file has not been opened, fm is undefined.

fmt

A character variable or array element 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 or array element 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*8 variable or array element 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 open for direct access.

nr

A full-word (8 byte) integer variable or array element; 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 or array element indicating blank interpretation currently in effect for a file open 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.

If a common block name is used for an inquire by unit, or if a segment access file is specified on an inquire by file, then only the IOSTAT, ERR, EXIST, OPENED, NAMED and NAME specifiers are valid.

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	

On an inquire by unit for a common block name, EXIST will always be true and NAMED will be the same as OPENED. On an inquire by file for a segment access file, NAME will return the common block name incuding slashes and NAMED will always be true.

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.

The RECL value returned is zero for an inquire by unit on an unopened but existing file.

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

A full-word (8 byte) 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

External unit identifier. The characters UNIT= are optional.

ios

A full-word (8 byte) 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.

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

External unit identifier. The characters UNIT= are optional.

ios

A full-word (8 byte) 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.

ENDFILE should not be the first operation on a file. The end-of-partition can be detected by the END= and IOSTAT= specifiers. ENDFILE can be used to unconditionally flush terminal output. If the unit specified is associated with a file that is connected to a terminal, the output buffer for the file is flushed to display messages immediately.

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.

Control Data Extension

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 OL=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 the number of words in the last record read by the previous formatted READ, list directed READ, 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 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.

A length of zero is returned if the previous READ was NAMELIST.

Example:

```
DIMENSION CALC(51)
BUFFER IN (1,K) (CALC(1), CALC(51))
J = LENGTH(1)
IF (UNIT(1) .GE. 0) GO TO 20
```

The variable J contains the number of words read on unit number 1.

LENGTHB

The LENGTHB function returns the number of bytes in the last record read by the previous formatted READ, list directed READ, BUFFER IN, or READMS of the specified unit. The LENGTHB function has the form:

LENGTHB (u)

u

Unit identifier.

The value returned by LENGTHB is of type INTEGER.

A length of zero is returned if the unit is not open, or if the unit is open but no input activity has been previously completed on that unit.

A length of zero is also returned if the previous READ was NAMELIST.

Example:

```
DIMENSION CALC(51)
BUFFER IN (1,K) (CALC(1), CALC(51))
J = LENGTHB(1)
IF (UNIT(1) .GE. 0) GO TO 20
```

The variable J contains the number of bytes read on unit number 1.

End of Control Data Extension

Control Data Extension

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.The data must be full-word data. 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.

End of Control Data Extension

Control Data Extension

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 with terminal attributes appropriate for the character set designated by the cs parameter. This call has the form:

CALL CONNEC (u, *cs***)**

u

Unit identifier.

cs

Character set designator; a full-word (8 byte) integer expression having one of the following values:

0 Selects the ASCII-128 character set. The terminal attributes currently in effect are used.

1 Same as 0, except that if transparent mode is on, it will be turned off and associated transparent mode attributes are reset to their default values.

2 Selects the ASCII-256 character set. Transparent mode is turned on so that all bits of each byte (including control characters) can be transmitted as data provided your terminal uses 8 bits with no parity. Also allows multi-message behavior so type-ahead can retain transparent mode.

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 is open but is not connected to a terminal, the file is first closed (and its buffer flushed) and then connected to the terminal under the specified character set option.This procedure is also followed for files connected using the REQUEST_TERMINAL command.

If a unit specified in a CONNEC call is already connected to the terminal and is also open, the terminal attributes are set according to the specified option; the associated file is not closed.

The ASCII 128-character set is shown in appendix J.

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.

End of Control Data Extension

Program Units 7

This chapter describes the basic units that are used to form FORTRAN programs.

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 (10I2)
200   FORMAT ('1THE AVERAGE OF ',/10(' ',I2),/,' IS ',F7.3)
      END

      FUNCTION NADD(IARRAY)
      INTEGER IARRAY(10)
      N = 0
      DO 10 I = 1,10
      N = IARRAY(I) + N
10    CONTINUE
      NADD = N
      RETURN
      END

      SUBROUTINE DIVIDE (N,I,Q)
      INTEGER N, I
      REAL Q
      Q = REAL(N) / REAL(I)
      RETURN
      END
```

Figure 7-1. 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 inline.

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 a main entry point for a FORTRAN program. Every program must have a main program unit.	Usually begins with PROGRAM statement.
Subroutine	Can be called from other program units in a program. Returns values through argument list or common.	Begins with SUBROUTINE statement.
External function	Can be called from other program units. Returns 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 the 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 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. The OPEN statement (which conforms to the ANSI standard) can also be used to declare input/output units. The PROGRAM statement has the forms:

PROGRAM name

PROGRAM name (upar, ..., *upar*)

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.

Control Data Extension

PROGRAM Statement Usage

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.

End of Control Data Extension

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 or different programs.

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(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.

For Better Performance

Subroutine calls and returns within a loop involve longer execution time. If possible, put the loop inside of the function or subroutine itself and call it only once. This way, the subroutine is only called once, rather than on each pass through the loop. For example:

```
      DIMENSION DATA(100)
      DO 20 J=1,100
      CALL SUBA(DATA(J))
   20 ANSWER(J) = DATA(J)
        :
      SUBROUTINE SUBA(VALUE)
      VALUE=VALUE**2
      RETURN
      END
```

can be changed to execute faster:

```
      DIMENSION DATA(100)
      CALL SUBA(DATA)
        :
      SUBROUTINE SUBA(VALUE)
      INTEGER VALUE(100)
      DO 20 J=1,100
      VALUE(J)=VALUE(J)**2
   20 CONTINUE
      RETURN
```

Also, if the subprogram contains a RETURN statement that causes frequent immediate returns, more the source of the rturn from the subprogram. For example, if the value of N is frequently 0,

```
      CALL X(A, B, N)
        :
      SUBROUTINE X(A, B, N)
      IF (N .EQ. 0) RETURN
        :
      RETURN
      END
```

can be changed to execute faster:

```
      IF (N .NE. 0) THEN
         CALL X(A, B, N)
      ENDIF
        :
      SUBROUTINE X(A, B, N)
        :
      RETURN
      END
```

Function Subprogram

Function subprograms can be external functions or intrinsic functions. Both external and intrinsic functions are external to the program unit that references them. External functions are written by the user, but intrinsic functions are supplied by the FORTRAN library.

External Functions

An external function begins with a FUNCTION statement. This statement has the form:

type **FUNCTION name*****len* (*d*, ..., *d*)

type

Type indicator. Can be INTEGER, REAL, DOUBLE PRECISION, COMPLEX, LOGICAL, BOOLEAN, or CHARACTER. If omitted, the function name determines the type according to the rules described in chapter 3.

name

Name of the function subprogram.

len

Length of the result of the function. Options for len are 2, 4, and 8 for type integer functions, 8 and 16 for type real functions and 16 for type complex functions.

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 length declared for a function name must be the same in the referencing program unit and in the referenced function.

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 FSMALL*2

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 as a variable or redefined later in the function. However, a recursive call to the function is not permitted. 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 (*).

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 the RETURN statement) 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.

For Better Performance

Double precision and real*16 constants, variables, arrays, and functions require more execution time because of the extra precision they support (two words).

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, 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 or extended Hollerith 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, size, 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, size, 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.

An actual argument that is an integer constant expression is passed as an eight-byte integer regardless of the actual size of the integer values in the expression.

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, or INTRINSIC 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).
- Dummy arguments of type integer must be equal in length to the associated actual argument.

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, subroutine name, or another dummy procedure.

For Better Performance

Whenever possible, replace each actual argument that is an array element with two actual arguments; one that is the entire array and one that is the subscript. For example:

```
      CALL SUBA(NUMA(I))
```

should be replaced with:

```
      CALL SUBA(NUMA,I)
```

where NUMA is the array name and I is the subscript variable. This is faster because the address of NUMA(I) is not known at compile time but the addresses of NUMA and I are known.

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 and size 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 element, 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. This can happen if the number of actual arguments in a subprogram reference is less than the number of dummy arguments, which is incorrect. It can also happen in procedures with multiple entry points which do not have the same dummy arguments. There is no error in this case unless an undefined dummy argument is referenced. 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 reference. 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 is not diagnosed, but causes an error during execution.)

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 (s+1−v), where s is the size of the actual argument array and v 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 s is 20, and v 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 wih 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.

For Better Performance

Assumed-length character strings require more execution time; replace with constant size strings whenever possible.

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. If an entry point name is of type INTEGER*2 or INTEGER*4, all other entry point names in the same function or subprogram must be integers of the same size. A dummy argument cannot be used in an ixecutable 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,K. Note that the SUBROUTINE and ENTRY argument lists legally contain different numbers of arguments. Dummy argument K cannot be referenced by any statement before the ENTRY statement except a declarative statement (such as a type statement).

RETURN Statement

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, which 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 returns. 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. For any other value, control returns to the statement following the CALL statement.

Control Data Extension

Calling Other-Language Subprograms

FORTRAN allows you to call subprograms written in languages other than FORTRAN. These languages include CYBIL, COBOL, C, and FORTRAN Version 2. 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 or function reference. Values can be passed between the CYBIL procedure and FORTRAN program through the argument list and function return value. If the CYBIL procedure name is a valid FORTRAN name (1 through 7 letters, digits, or underscores, beginning with a letter), the FORTRAN procedure must be declared with the XDCL attribute in the CYBIL procedure. If the CYBIL procedure name is not a valid FORTRAN name, you must use the C$ EXTERNAL statement to rename the procedure. The C$ EXTERNAL statement is described in appendix D. The dummy (formal) parameters must be declared in VAR of the CYBIL routine 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. When passing arrays to a CYBIL subprogram, you should not pass assumed-size or adjustable arrays.

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	—
CHARACTER*264	OST$STATUS

The OST$STATUS variable of system-resident CYBIL procedures can be processed as a FORTRAN character variable by using the NOS/VE status subprograms described in chapter 9.

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)

Arithmetic variables, constants, symbolic constants, and expressions that are passed to a CYBIL routine are normally passed by value, that is, the value of the parameter is passed rather than its address. You can pass variables, constants, symbolic constants, and expressions by address using the PTR function and the C$ EXTERNAL directive. You can also use the C$ EXTERNAL directive to call a CYBIL routine name that does not conform to FORTRAN naming conventions. Character data is always passed by reference.

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= ', I2)
       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 COBSUB, 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
                                                  with items in the
                                                  COMMON-STORAGE section.

      STR1 = 'Here are some numbers'
      I = 4
      X = 2.414
C
C     Call COBOL subprogram.
C
      CALL CBSUB (STR1, STR2) <----------------    Actual arguments are associated
                                                  with items declared in LINKAGE
                                                  section.
      PRINT *, STR2
      END
```

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
                                                   arguments.
77 STRING-2 PIC X(9).
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
```

Calling C Routines

You can call a C routine from a FORTRAN program using a standard CALL statement or function reference. Because of different calling sequences and naming conventions, you must first declare the name of the C routine in a C$ EXTERNAL compiler directive (described in appendix D). The C$ EXTERNAL compiler directive allows your program to reference the external routine by a valid FORTRAN program unit name. Values can be passed between the FORTRAN program and C routine through the argument list.

The arguments passed to the C routine can be constants, symbolic constants, variables, expressions with operators, or arrays. Arguments can be of type integer, real, or character. In C, character data are considered to be arrays of single characters. For example, if a character variable of length 3 is passed to a C routine, it will be treated as a character array of size 3, with each array element representing a character value:

FORTRAN Character Data	C Character Data
CHARACTER COLOR*3 COLOR='RED'	char color [3];

The variable COLOR contains the character string 'RED'	The array COLOR has the following values:
	color[0]='R'
	color[1]='E'
	color[2]='D'

The correspondence of data types between FORTRAN and C is as follows:

FORTRAN Data Type	C Data Types
REAL	float or double
INTEGER(full word)	int
CHARACTER*N	char array[n]
DOUBLE PRECISION	—
COMPLEX	—
BOOLEAN	—
LOGICAL	—

Variables, constants, symbolic constants, and expressions that are passed to a C routine are normally passed by value, that is, the value of the parameter is passed rather than its address. You can pass variables, constants, symbolic constants, and expressions by address using the PTR function. Arrays are passed to a C routine by address; the address of the first data element is passed rather than its value.

Only integer and real function values are supported by C.

If a routine written in C expects an address of a data value, rather than the value itself, use the PTR intrinsic function with the data value name as an argument in the CALL statement or function reference. PTR(a) is a generic function that returns the address of a. The result of the PTR function can not be used within a FORTRAN program unit.

Before using the C compiler or executing a routine written in C, you must execute these two SCL commands:

```
        $SYSTEM.C.SETUP
        SET_WORKING_CATALOG,$USER
```

The following example shows a FORTRAN main program that calls a C routine named SEE_ME:

FORTRAN program:

```
      PROGRAM MEETING
      INTEGER IHOUR
      CHARACTER WEEKDAY*7
C$    EXTERNAL (ALIAS='see_me', LANG=C) CPROC
      IHOUR=5
      CALL CPROC (WEEKDAY)
      PRINT *, WEEKDAY, 'AT', IHOUR
      END
```

C Routine:

```
see_me (weekday)
  char weekday[7];
{         weekday[0]='T';
          weekday[1]='U';
          weekday[2]='E';
          weekday[3]='S';
          weekday[4]='D';
          weekday[5]='A';
          weekday[6]='Y';
}
```

The argument IHOUR and WEEKDAY are passed by value to the C routine. The character variable WEEKDAY of size 7 is treated as an array of size 7 in the C routine. The value of the first character of WEEKDAY is the same as the value of the first element of WEEKDAY and so forth.

Example:

```
      PROGRAM P
C$   EXTERNAL (ALIAS='c_routine', LANG=C) CSUB
      CALL CSUB(PTR(J))
      END

      c_routine (pj)
      int *pj;
```

```
{
          int a=10;
          pj=&a;
}
```

The main program calls the C routine and passes the argument J by address rather than by value. The address of J was needed in the C routine.

Calling FORTRAN Version 2 Subprograms

Normally, you reference FORTRAN routines by CALL statements or function references. However, if a FORTRAN routine name does not confrom to FORTRAN Version 1 naming conventions (1 through 7 letters or digits, beginning with a letter), you can use the C$ EXTERNAL directive to access the routine. The C$ EXTERNAL directive allows your program to reference the routine with a valid FORTRAN Version 1 name.

The arguments passed to the FORTRAN routine follow the same rules as arguments to a FORTRAN subroutine or function.

Example (FORTRAN Version 1 calling program):

```
      PROGRAM MAIN
      REAL ARAY(5)
        :
C$    EXTERNAL (ALIAS='FTN_SUBROUTINE', LANG=FTN) FTNSUB
        :
      END
```

FORTRAN Version 2 subroutine:

```
      SUBROUTINE FTN_SUBROUTINE(ARAY)
      REAL ARAY(5)
      ARAY(1) = LOG (ARAY(1))
      RETURN
      END
```

The subroutine is wrtten in FORTRAN Version 2 which allows 31 character program unit names. The C$ EXTERNAL statement causes the calling program to recognize the subroutine by the name FTNSUB, which is a valid FORTRAN Version 1 name.

To execute the program (FMAIN contains the main program's binary object file; FSUB contains the subroutine's binary object file):

```
/execute_task (fmain, fsub)
```

End of Control Data Extension

Intrinsic Functions 8

This chapter describes the FORTRAN and Math Library supplied functions that perform various mathematical operations.

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 (boolean type will be converted). An IMPLICIT NONE statement does not affect the type of the results of any intrinsic functions.

Intrinsic routines are in the Math Library and normally accessed through the call-by-value calling procedure. To access an intrinsic function through the call-by-reference calling procedure, specify EE=R on the FORTRAN command.

NOTE

You should ensure that real, double precision, and complex arguments to intrinsic functions are in normalized standard floating point form, as unnormalized or non-standard arguments can cause undefined results. FORTRAN automatically normalizes all real, double precision, and complex constants, and results of all floating point operations with standard normalized or zero operands are normalized or zero. However, it is possible to generate unnormalized or nonstandard operands by means of boolean expressions, equivalencing, or various input operations.

Generic and Specific Names

Certain intrinsic functions have ageneric 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. An integer*2 or integer*4 value used as an argument to a function is converted to a full word (8 byte) integer before being used as an argument. The conversion does not change the sign of the argument. A function accepting integer, real, complex or double precision type arguments will also accept boolean arguments. A boolean argument is converted to integer, if it's an allowable argument type; otherwise, it is converted to real, before computation. 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, complex or boolean (converted to real). 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 or boolean argument and returns the result. Likewise, the specific name DLOG is for double precision (or boolean) arguments and double precision results and the specific name CLOG is for complex (or boolean) arguments and complex 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. Boolean arguments are not listed in the table, but follow the conversion rules described above. 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*2	Integer	Absolute value
		Integer*4		
		Integer		
	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	AMAX0	Integer	Real	Maximum value
None	AMIN0	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*2	Character	Integer conversion to character
		Integer*4	Character	
		Integer	Character	
None*	COMPL	Any type but character	Boolean	Complement
None*	COTAN	Real	Real	Cotangent (argument in radians)

Integer denotes full word (eight-byte) integers.

(Continued)

Table 8-1. Intrinsic Functions *(Continued)*

Generic Name	Specific Names	Type of Argument	Type of Function	Description
CMPLX	—	Integer*2	Complex	Conversion to complex
		Integer*4	Complex	
		Integer	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.
DBLE	—	Integer*2	Double	Conversion to double
		Integer*4	Double	
		Integer	Double	
	—	Real	Double	precision
	—	Double	Double	
	—	Complex	Double	
DIM	IDIM	Integer*2	Integer	Positive difference
		Integer*4		
		Integer		
	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	
EXTB	None	a1: Any type but character a2,a3: Integer	Boolean	Extract a string of bits
None	ICHAR	Character	Integer	Character conversion to integer

Integer denotes full word (eight-byte) integers.

(Continued)

Table 8-1. Intrinsic Functions *(Continued)*

Generic Name	Specific Names	Type of Argument	Type of Function	Description
None	INDEX	Character	Integer	Index of a substring
INSB	None	a1,a4: Any type but character a2,a3: Integer	Boolean	Insert a string of bits
INT	INT	Integer*2	Integer	Conversion to integer
		Integer*4	Integer	
		Integer	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 or equal
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*2	Integer	Largest value
		Integer*4		
		Integer		
	AMAX1	Real	Real	
	DMAX1	Double	Double	
None	MAX1	Real	Integer	
MIN	MIN0	Integer*2	Integer	
		Integer*4		
		Integer		Smallest value
	AMIN1	Real	Real	
	DMIN1	Double	Double	
None	MIN1	Real	Integer	

Integer denotes full word (eight-byte) integers.

(Continued)

Table 8-1. Intrinsic Functions *(Continued)*

Generic Name	Specific Names	Type of Argument	Type of Function	Description
MOD	MOD	Integer*2	Integer	Remaindering
		Integer*4	Integer	
		Integer	Integer	
	AMOD	Real	Real	
	DMOD	Double	Double	
None*	NEQV	Any type but character	Boolean	Nonequivalence
NINT	NINT	Real	Integer	Nearest integer
	IDNINT	Double	Integer	
None*	OR	Any type but character	Boolean	Boolean sum
PTR*	None	Any type	Boolean	Parameter address; used only when passing parameters to C or CYBIL routines
None*	RANF	None	Real	Random number generator
REAL	FLOAT	Integer*2	Real	Conversion to real
		Integer*4	Real	
		Integer	Real	
REAL	FLOAT	Integer*2	Real	Conversion to real
		Integer*4	Real	
		Integer	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*2	Integer	Transfer of sign
		Integer*4		
		Integer		
	SIGN	Real	Real	
	DSIGN	Double	Double	

Integer denotes full word (eight-byte) integers.

(Continued)

Table 8-1. Intrinsic Functions *(Continued)*

Generic Name	Specific Names	Type of Argument	Type of Function	Description
SIN	SIN DSIN CSIN	Real Double Complex	Real Double Complex	Sine (argument in radians)
None*	SIND	Real	Real	Sine (argument in (degrees)
SINH	SINH DSINH	Real Double	Real Double	Hyperbolic sine
SQRT •	SQRT DSQRT CSQRT	Real Double Complex	Real Double Complex	Square root
SUM1S	None	Integer Real Double Complex	Integer	Number of bits that are set in a word
TAN	TAN DTAN	Real Double	Real Double	Tangent (argument in radians)
None*	TAND	Real	Real	Tangent (argument in degrees)
TANH	TANH DTANH	Real Double	Real Double	Hyperbolic tangent
None*	XOR	Any type but character	Boolean	Exclusive OR

Integer denotes full word (eight-byte) integers.

Table 8-2. Mathematical Intrinsic Functions

Function	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)	a2<0, a1<0	−pi < ATAN2(a1,a2) < −pi/2
DATAN2(a1,a2)	a2<0, a1≥0	pi/2 ≤ ATAN2(a1,a2) ≤ pi
	a2=0, a1<0	ATAN2(a1,a2) = −pi/2
	a2=0, a1>0	ATAN2(a1,a2) = pi/2
	a2>0, a1<0	−pi/2 < ATAN2(a1,a2) < 0
	a2>0, a1≥0	0 ≤ ATAN2(a1,a2) < pi/2
	a2=0, a1=0 (error)	
ATANH(a)	|a| ≤ 1	

COS(a)
DCOS(a)
	|a| < 2**47	−1 ≤ COS(a) ≤ 1

COTAN(a)	|a| < 2**47	
CCOS(ar,ai)	|ar| < 2**47 |ai| < 4095*log(2)	-1≤ CCOS(x)≤ 1 x=ar+(ai)i
COSD(a)	|a| < 2**47	−1 ≤ COSD(a) ≤ 1

COSH(a)
DCOSH(a)
	|a| < 4095*log(2)	COSH(a) ≥ 1
		DCOSH(a) ≥ 1

ERF(a)	infinity ≤ a ≤ infinity	−1 ≤ ERF(a) ≤ 1
ERFC(a)	infinity ≤ a ≤ 25.923	≤ ERFC(a) ≤ 2

EXP(x)
DEXP(x)
	x < 4095*log(2)
	x ≥ -4097*log(2)	

CEXP(ar,ai)	|ar| < 4095*log(2) |ar| < -4097*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) ** 1/2 in machine range	−pi < CLOG(ai) < pi

(Continued)

Table 8-2. Mathematical Intrinsic Functions *(Continued)*

Function	Domain	Range

LOG10(a)
ALOG10(a)
DLOG10(a)
	$a > 0$	$\|LOG10(a)\| < 4095*log(2)$ base 10

SIN(a)
DSIN(a)
	$\|a\| < 2**47$	$-1 \leq SIN(a) \leq 1$

CSIN(ar,ai)	$\|ar\| < 2**47$ $\|ai\| < 4095*log(2)$	
SIND(a)	$\|a\| < 2**47$	$-1 \leq SIND(a) \leq 1$

SINH(a)
DSINH(a)
	$\|a\| < 4095*log(2)$	

SQRT(a)
DSQRT(a)
	$a \geq 0$	$SQRT(a) \geq 0$

CSQRT(ar,ai)	(ar**2+ai**2) **1/2 + $\|ar\|$ in machine range	value in right half ofplane ($ar \geq 0$)

TAN(a)
DTAN(a)
	$\|a\| < 2**47$	

TAND(a)	$\|a\| < 2**47$ a cannot be exact odd multiple of 90	
TANH(a)		$-1 \leq TANH(a) \leq 1$

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 a boolean, integer (any size), 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 boolean, real or double precision argument. The result is in radians. The result is real or double precision, depending the argument type. The specific names are ACOS and DACOS.

AIMAG

AIMAG(a) is a specific function that returns the imaginary part of a boolean or 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 boolean, real or double precision argument, the result is 0 if $|a| < 1$. If $|a| \geq 1$, 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 boolean or 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 or boolean 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 boolean or integer; 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 boolean or real; 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 boolean or integer; 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 boolean or real; 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 boolean or real; the result is real. If a2 is zero, results are undefined. The generic name is MOD.

Control Data Extension

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; the result is boolean. The result is the same as for the boolean .AND. operator.

End of Control Data Extension

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, boolean or double precision; 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 boolean, 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 boolean, 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.

Control Data Extension

ATANH

ATANH(a) is a specific function that returns the hyperbolic arctangent of a boolean or real argument. The result is real.

End of Control Data Extension

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 (a boolean argument is converted to real). The arguments must not both be zero. The specific names are ATAN2 and DATAN2.

Control Data Extension

BOOL

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 (any size), 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.

End of Control Data Extension

CABS

CABS(a) is a specific function that returns the absolute value of a boolean or 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 boolean or 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 is boolean or complex. The result is 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 (any size); 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 you specify an argument greater than 255 and compile with DC=FIXED, mod(i, 256) is used. If you specify an argument greater than 255 and compile with DC=USER, a runtime error is issued.) If, in a user-specified collating sequence, more than one character has weight i, any of the characters can be returned. User-specified collating sequences are explained in Appendix H of this manual.

CLOG

CLOG(a) is a specific function that returns the natural logarithm (logarithm base e) of the argument. The argument is boolean or complex. The result is 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 boolean, integer (any size), real, double precision, or complex. A boolean argument is treated as a bit string and is not changed.

For two arguments a1 and a2, the arguments must be of the same type (one or both can be of type boolean which is converted to real) 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.

Control Data Extension

COMPL

COMPL(a) returns a complemented value. The argument can be any type except character; the result is boolean. If the argument is not boolean, the argument is converted with BOOL(a). The result is the value of the logical operator .NOT. on a boolean value.

End of Control Data Extension

CONJG

CONJG(a) is a specific function that returns the conjugate of a boolean or 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 boolean, real, double precision, or complex argument. The argument is in radians. The result has the same type as the argument (boolean converted to real). The specific names are COS, CCOS, and DCOS.

Control Data Extension

COSD

COSD(a) is a specific function that returns the cosine of a boolean or real argument. The argument is in degrees. The result is real. COSD does not have a generic name.

End of Control Data Extension

COSH

COSH(a) is a generic function that returns the hyperbolic cosine of a boolean, real or double precision argument. The result is real or double precision, depending on the argument type. The specific names are COSH and DCOSH.

Control Data Extension

COTAN

COTAN(a) is a specific function that returns the cotangent of a boolean or real argument. COTAN first reduces a by modulo 2*pi. The argument is expressed in radians and the result is real. COTAN does not have a generic name.

End of Control Data Extension

CSIN

CSIN(a) is a specific function that returns the sine of a boolean or 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 boolean or 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 boolean or 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 boolean or 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 boolean or 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 boolean or 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 can be boolean or double precision. The result is 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 boolean, integer (any size), real, double precision, or complex. A boolean argument is treated as a bit string and is not changed. 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 boolean or 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 boolean or 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 less than a2, it returns zero. The argument can be boolean or double precision. The result is 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 can be boolean or double precision. The result is 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 unless one is of type boolean (converted to real). 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 can be boolean or double precision. The result is 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 boolean or 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 boolean or 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 can be boolean or double precision. 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 boolean or double precision; 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 can be boolean or double precision; the result is 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 can be boolean or double precision; the result is 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 boolean or 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 can be boolean or double precision; the result is double precision. The generic name is SIGN. See the SIGN description.

DSIN

DSIN(a) is a specific function that returns the sine of a boolean or 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 boolean or 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 boolean or 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 boolean or 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 boolean or double precision argument. The result is double precision. The generic name is TANH. See the TANH description.

Control Data Extension

EQV

EQV(a, ..., a) returns the equivalence of the 2 through 500 arguments. The arguments can be any type except character; the result is boolean. The result is the same as for the boolean .EQV. operator.

End of Control Data Extension

Control Data Extension

ERF

ERF(a) returns an error function result. The argument can be boolean or real; the result is real. The argument must be positive. The mathematical definition is as follows:

$$\mathrm{ERF(x)} = 2\ /\sqrt{\mathrm{pi}} \int_{0}^{x} e^{-t^2}\,dt$$

End of Control Data Extension

Control Data Extension

ERFC

ERFC(a) returns a complementary error function result. The argument can be boolean or real; the result is real. The result is 1−ERF(a). The mathematical definition of ERFC is as follows:

$$\text{ERFC}(x) = 2 / \sqrt{pi} \int_{x}^{\infty} e^{-t2}\, dt$$

End of Control Data Extension

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 (a boolean argument is converted to real). The specific names are EXP, DEXP, and CEXP.

Control Data Extension

EXTB

EXTB(a1, a2, a3) is a generic function that returns extracted bits from a1; a2 specifies the ordinal of the first bit to be extracted and a3 specifies the number of bits to be extracted. Bits are numbered from the left starting with zero.

Argument a1 can be of any type except character; however, if the argument is of type double precision or complex, only the first word is used. Arguments a2 and a3 must be of type integer and greater than or equal to zero. There is no specific name.

Argument a2 must also be less than or equal to 63, and the sum of a2 and a3 must be less than or equal to 64.

End of Control Data Extension

FLOAT

FLOAT(a) is a specific function that returns the value of the boolean or 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 a boolean or integer (any size) 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 boolean or 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 boolean or 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 boolean or double precision. The generic name is NINT. See the NINT description.

IFIX

IFIX(a) is a specific function that returns the value of the boolean or 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.

Control Data Extension

INSB

INSB(a1, a2, a3, a4) is a generic function that returns a copy of a4 with the bits from a1 inserted. The rightmost a3 bits of a1 are inserted at bit position a2 of a4. Argument a4 itself is not altered. Bits are numbered from the left starting with zero.

Arguments a1 and a4 can be of any data type except character; however, if either a1 or a4 is double precision or complex, only the first word is used. Arguments a2 and a3 must be of type integer and greater than or equal to zero. There is no specific name.

Argument a2 must also be less than or equal to 63, and the sum of a2 and a3 must be less than or equal to 64.

End of Control Data Extension

INT

INT(a) is a generic function that performs type conversion to integer. The result is integer, and the argument can be boolean, integer (any size), 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 $|a| \geq 1$, 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 $-|a1|$ if a2 is negative. The arguments can be boolean or integer; the result is 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, 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 as 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 as 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 (a boolean argument is converted to real). 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 (a boolean argument is converted to real). The argument must be greater than zero. The specific names are ALOG10 and DLOG10.

Control Data Extension

MASK

MASK(a) returns a boolean result. The argument is integer or boolean in the range 0 through 64. The result is a word of a left-justified one bits followed by (64 – a) zero bits. If argument a is less than zero or greater than 64, the result is undefined.

End of Control Data Extension

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 (a boolean argument is converted to integer). The specific names are MAX0, AMAX1, and DMAX1. All of the arguments that are not boolean must be of the same type. The result is the same type as the nonboolean arguments, unless all the arguments are boolean when the result is integer.

MAX0

MAX0(a, ..., a) is a specific function that returns the largest value from the 2 through 500 boolean or 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 boolean or 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 (a boolean argument is converted to integer). 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 boolean or 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 boolean or 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 (a boolean argument is converted to integer). If only one argument is boolean, the result is the type of the other argument. If both arguments are boolean, the result is integer. The result is a1 – (INT(a1/a2) * a2). If a2 is zero, results are undefined. The specific names are MOD, AMOD, and DMOD.

Control Data Extension

NEQV

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.

End of Control Data Extension

NINT

NINT(a) is a generic function that returns the nearest integer. The result is integer, and the argument can be boolean, 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.

Control Data Extension

OR

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.

End of Control Data Extension

Control Data Extension

PTR

PTR (a) is a generic function that returns the address of a. This function can only be used in a statement that is calling or referencing a C or CYBIL routine. The argument can be of any type and the result is boolean. The result can not be used within a FORTRAN program unit.

End of Control Data Extension

Control Data Extension

RANF

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.

End of Control Data Extension

REAL

REAL(a) is a generic function that performs type conversion and returns a real result. The argument can be boolean, integer (any size), real, double precision, or complex. A boolean argument is treated as a bit string and is not changed. 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.

Control Data Extension

SHIFT

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. If a2 is outside this range, the result is undefined.

End of Control Data Extension

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 (a boolean argument is converted to integer). 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 (a boolean argument is converted to real). The generic name is SIN. See the SIN description. The specific names are SIN, DSIN, and CSIN.

Control Data Extension

SIND

SIND(a) returns the sine of a boolean or real argument. The result is real. The argument is in degrees.

End of Control Data Extension

SINH

SINH(a) is a generic function that returns a hyperbolic sine of a boolean, 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 boolean or 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 (a boolean argument is converted to real). The argument must not be negative. The specific names are SQRT, DSQRT, and CSQRT.

Control Data Extension

SUM1S

SUM1S(a) is a generic function that returns the number of bits that are set. A set bit is one with the binary value '1'. The argument can be any type but character or logical; however, if the argument is of type double precision or complex, only the first word is used.

End of Control Data Extension

TAN

TAN(a) is a generic function that returns the tangent of a boolean, 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.

Control Data Extension

TAND

TAND(a) is a specific function that returns the tangent of a boolean or real argument. The result is real. The argument is in degrees. TAND does not have a generic name.

End of Control Data Extension

TANH

TANH(a) is a generic function that returns the hyperbolic tangent of a boolean, real, or double precision argument. The result is real or double precision, depending on the argument type. The specific names are TANH and DTANH.

Control Data Extension

XOR

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.

End of Control Data Extension

FORTRAN-Callable Subprograms 9

FORTRAN provides a number of subroutine and function calls that enable you to access other Control Data products as well as perform a variety of other operations. (The subprograms are all non-ANSI.)

FORTRAN-Callable Subprograms 9

FORTRAN provides a set of subprograms that enable you to take advantage of various NOS/VE Operating System features. The subprograms allow you to:

- Access program execution command parameters
- Access SCL variables used before or after program execution
- Execute any NOS/VE SCL command within a FORTRAN program
- Handle the NOS/VE status variable

FORTRAN also provides a library of utility routines to perform various tasks, including:

error handling

random number generation

collating sequence control

The following information provides only an introduction from a FORTRAN program perspective. For complete information on the associated NOS/VE features, refer to the SCL Language Definition manual or the SCL System Interface manual.

NOS/VE Status Subprograms

The NOS/VE status subprograms allow you to manipulate the NOS/VE status variable and parameter from within your FORTRAN program. The NOS/VE status parameter and variable (of type OST$STATUS) have a record structure that is incompatible with FORTRAN data types and forms. The status variable and parameter contain three fields of information:

NORMAL field

Contains a boolean flag (one bit) indicating if an error occured.

CONDITION field

Contains the condition code which is the binary combined value of the product indentifier and the condition number.

TEXT field

Contains information for the message template; usually not in a printable format.

Because of the different data types contained in the NOS/VE status variable, you should not try to print or access it without using the status subprograms described in this section. The NOS/VE status variable information is returned to a FORTRAN program in the form of a character string or array. If a FORTRAN character string is used, it is declared as CHARACTER*264; if an array is used, it is declared with 264 elements of CHARACTER*1. The subprograms process the FORTRAN string or array and allow you to manipulate the status variable fields through the FORTRAN string or array (called fstat in the subprogram descriptions). The subprograms and their brief descriptions are:

INTCOND

Returns the integer value of the condition code.

CONDNAM

Returns the condition name from an integer condition code or FORTRAN condition number.

UPKSTAT

Returns the value of the NORMAL field, the product identifier and condition number from the condition code, the length of the TEXT field, and the string in the TEXT field in fstat.

CONDSYM

Returns the string representation for a condition code in the form of product identifier and condition number.

NOTE

The routines GETSVAL and REDSVAR, also described in this chapter, return the status variable information from an existing SCL status variable; the NOS/VE status processing subprograms described in this section process status parameters that are returned directly to a FORTRAN program. For example, from the CREV, DELV, or SCLCMD subroutines or from a system-resident CYBIL routine.

The subprograms are described below in alphabetical order.

CONDNAM

The CONDNAM function returns the condition name from an integer representation of the CONDITION field of a NOS/VE status variable. The function has the form:

CONDNAM(condition)

condition

Integer expression specifying the value of the condition code whose condition name is to be returned as a string.

The value returned by the CONDNAM function is a character string with a length of 31. CONDNAM must be declared type CHARACTER*31 in the calling program. If the specified condition number does not exist, the string 'UNKNOWN_CONDITION' is returned. If the specified condition number does exist, the returned condition name is in the standard NOS/VE condition identifier format. Some examples of existing condition names are:

FLE$INT_FILE_SIZE_TOO_BIG

CLE$UNKNOWN_VARIABLE

MLE$CEXP_REAL_RANGE

DBE$LINE_EXTENT_ZERO

You can use the INTCOND function to first return an integer representation of the condition code. For example:

```
CHARACTER CONDTN*31, CONDNAM*31, S1*264
  :
CALL CREV ('BALANCE', 'INTEGER', LEN, 1, 1, 'JOB', S1)
CONDTN = CONDNAM(INTCOND(SI))
```

The call to the CREV subroutine specifies the creation of an SCL integer variable name BALANCE, with a scope of JOB. After execution, the variable CONDTN contains the condition name and S1 contains the NOS/VE status information. If the variable BALANCE does not already exist, it is created and the value of CONDTN is FLE$NO_ERROR. If the variable BALANCE does already exist, the value of CONDTN is CLE$VAR_ALREADY_CREATED. Condition names and their associated condition codes are listed in the Diagnostic Messages for NOS/VE manual.

To read a description of a condition name online, specify the condition name on the SCL EXPLAIN command. For example, to read about the condition CLE$UNKNOWN_VARIABLE in the online messages manual, enter

```
/explain m=messages s='cle$unknown_variable'
```

The online messages manual provides a brief description of the condition, the associated product identifier and condition number, and a recommended user action.

CONDSYM

The CONDSYM call returns the string representation of an integer condition code. The string representation is in the form of product identifier and condition number. This call has the form:

CALL CONDSYM (condition, string, len)

condition

Integer expression specifying either the condition code or FORTRAN condition number whose value is to be returned as a string.

string

Character variable to receive the string representation. A value shorter than the argument is left justified and blank filled; a value longer is truncated on the right.

len

Integer variable to receive the length of the returned string.

The value returned in string is normally of the form xxnumber, where xx is the product identifier portion of the condition code and number is the condition number portion of the condition code. If the identifier portion of the condition code is not two displayable characters, the string represents the condition code as a number of up to 12 digits.

If a FORTRAN condition number is supplied for the condition parameter, the returned product identifer is the string FL. The INTCOND function can be used first to return an integer representation of the condition code contained in the fstat or iostat parameter or the CYBIL OST$STATUS variable. The fstat parameter is described under the CREV, DELV, and SCLCMD commands. The iostat parameter is an input/output specifier and described in chapter 7.

INTCOND

The INTCOND function returns the condition code field from a NOS/VE status variable contained in a FORTRAN character variable. This function has the form:

INTCOND(fstat)

fstat

Character variable, array, or substring specifying the name of the FORTRAN variable containing the NOS/VE status information for which the condition code is to be returned.

The value returned is of type integer. If the NORMAL field of the status variable was TRUE, (indicating no error or exception), the value returned is zero. Otherwise, the value returned is the condition code specifying the exception.

You can use the CONDNAM function to return the associated condition name of a value returned by the INTCOND function. For example:

```
CHARACTER CONDTN*31, CONDNAM*31, S1*264
  :
CALL CREV ('BALANCE', 'INTEGER', LEN, 1, 1, 'JOB', S1)
CONDTN = CONDNAM(INTCOND(SI))
```

The call to the CREV subroutine specifies the creation of an SCL integer variable name BALANCE, with a scope of JOB. After execution, the variable CONDTN contains the condition name and S1 contains the NOS/VE status information. If the variable BALANCE does not already exist, it is created and the value of CONDTN is FLE$NO_ERROR. If the variable BALANCE does already exist, the value of CONDTN is CLE$VAR_ALREADY_CREATED. Condition names and their associated condition codes are listed in the Diagnostic Messages for NOS/VE manual.

UPKSTAT

The UPKSTAT call returns the information contained in a NOS/VE status variable in separate parts. This call has the form:

CALL UPKSTAT (fstat, norm, id, cond, len, text)

fstat

Character variable, array, or substring name containing the NOS/VE status information from which to retrieve information.

norm

Logical variable to receive the value of the normal field of the NOS/VE status variable. The normal field contains one of the values TRUE or FALSE.

id

Character variable to receive the value of the identification portion of the condition field. Should be declared with at least a length of 2.

cond

Integer variable to receive the condition code of the NOS/VE status variable.

len

Integer variable to receive the length (number of characters) of the data in the text field.

text

Character variable to receive the value of the TEXT field of the status variable. Should be 256 characters in length. A value shorter than the argument is left justified and blank filled; a value longer is truncated on the right.

If the value returned in the norm parameter is true, the id, cond, len, and text parameters are undefined.

The fstat parameter receives status information in one of the following ways:

- Returned in the fstat parameter on the CREV, DELV, or SCLCMD call statements
- Returned in the OST$STATUS variable from a system-resident CYBIL routine

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.

Program Execution Command 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, 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
```

The value 3 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 13.

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 (pdefs)

where pdefs is a character constant expression, whose value is of the form '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. Long parameter definitions can be made by specifying the required character constant in a PARAMETER statement, which may be continued over 19 lines, and then referencing the symbolic name of the constant in the C$ PARAM directive.

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; values specified for parameter B will be SCL variables of kind string.

The following example shows how to represent five parameters with a single C$ PARAM directive:

```
      CHARACTER * (*) P1, P2, P3, P4, P5
      PARAMETER (P1 = 'P1: BOOLEAN;')
      PARAMETER (P2 = 'P2: LIST 1..7, 2..2  RANGE OF INTEGER -3..10;')
      PARAMETER (P3 = 'P3: RANGE OF VAR OF STATUS;')
      PARAMETER (P4 = 'P4: LIST 2..4 of FILE;')
      PARAMETER (P5 = 'P5: RANGE OF INTEGER;')
C$    PARAM (P1//P2//P3//P4//P5)
```

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 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. Arguments of type integer must be full word integers, that is, typed as INTEGER*8. 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 described on the following pages 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. Allowable values are: 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, KEY, 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. Allowable values are: 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. Values shorter than the argument are left justified and blank-filled; values longer are truncated on the right. The variable must be at least 256 characters long to receive FILE parameter values.

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. Options are: 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. Options are: 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:

GETVCNT (param, setnum)

param

Character expression specifying the name of the parameter for which the number of values in the specified value set 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. Options are: 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.

The TSTPARM function must be declared type logical in the calling program.

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. Variables or arrays of type integer must be full word integers, that is, typed as INTEGER*8.

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. Must be in the range 0 through 999999.

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.

CREV

The CREV call defines a new SCL variable. This call has the form:

CALL CREV (var, knd, len, lb, ub, scope, ***fstat*****)**

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.

fstat

Character variable to receive the status resulting from the execution of the CALL CREV command; fstat must be declared with a length of 264 in the calling program. Use the NOS/VE status processing subprograms to retrieve data from the variable.

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, ***fstat*****)**

var

Character expression specifying an SCL variable name. Maximum length of an SCL variable name is CHARACTER*256.

fstat

Character variable to receive the status resulting from the execution of the CALL DELV command; fstat must be declared with a length of 264 in the calling program. Use the NOS/VE status processing subprograms to retrieve data from the variable.

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 or array for which values are to be stored. Maximum length of a variable or array 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. If ref is an array, val must be an array containing one value for each element or ref.

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

If val is an array, bool must be an array containing one value for each element of val.

SCL type boolean corresponds to FORTRAN type logical.

WRTCVAR

The WRTCVAR call stores values into the specified SCL string variable or array. This call has the form:

CALL WRTCVAR (ref, dim, len, str)

ref

Character expression specifying the name of the SCL string variable or array into which values are to be stored. Maximum length of an SCL variable or array 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. If str is an array, len must be an array containing one value for each element of str.

str

Character expression whose value is a string to be stored. If ref specifies an SCL array, str must 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 or array. This call has the form:

CALL WRTIVAR (ref, dim, val, rad)

ref

Character expression specifying the SCL integer variable or array 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 or array element 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 values to be stored in the SCL variable or array. If ref specifies an array, val must 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 must be an array containing a value for each value being stored.

WRTSVAR

The WRTSVAR call stores values into the specified SCL STATUS variable or array. This call has the form:

CALL WRTSVAR (ref, dim, norm, id, cond, text)

ref

Character expression specifying the name of the STATUS variable or array into which values are to be stored. Must be a valid SCL variable or array reference, with variable name and subscript or field name if appropriate. Maximum length of an SCL STATUS variable or array element reference is 256 characters.

dim

Integer expression specifying the dimension of 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 must 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 must 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 must 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 must 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 (text, ***fstat*****)**

text

Character expression specifying a valid SCL command line. Maximum length of a command line is 256 characters. A command line can contain multiple commands separated by semicolons.

fstat

Character variable to receive the status resulting from the execution of the SCL command; fstat must be declared with a length of 264 in the calling program. Use the NOS/VE status processing subprograms to retrieve data from the variable.

Example:

```
CALL SCLCMD ('SETFA FILE=ABC MAXIMUM_RECORD_LENGTH=500')
```

This call executes a SETFA command.

Example:

```
CHARACTER COMMAND*50, STAT*256
READ *, COMMAND
CALL SCLCMD(COMMAND,STAT)
```

This examples uses a character variable to read an SCL command from the terminal.

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. Arguments that are of type integer must be full word integers, that is, typed as INTEGER*8. The following table presents a summary of the FORTRAN utility subprograms:

Table 9-1. 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 runtime 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 in the last record read.
	LENGTHB	Returns number of bytes in the last record read.
	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 access file.
	STINDX	Specifies a subindex for a random file.

(Continued)

Table 9-1. FORTRAN Utility Subprograms *(Continued)*

Miscellaneous Input/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.
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.
	CLOCK	Returns the current clock time.
	DISPLA	Places a message in the job log.
	REMARK	Places a message in the job log.
	SSWTCH	Tests the system sense switches.
	CHGUCF	Enables and disables system conditions.
	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. The Debug utility is described in appendix K.

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 occurrences of the triplet a, b, f is 20. (The number of occurences of the triplet a, b, f may vary throughout a program, but a warning message will be issued.)

The first and last word specified in a PDUMP call must be in the same segment.

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 FORTRAN runtime error message at any time during program execution. This call has the form:

CALL SYSTEM (ernum, msg)

ernum

Error code. An integer expression whose value is in the range 0 through 9999 decimal.

msg

Error message in the form of a character constant.

The error message is issued immediately when SYSTEM is called. If the specified error number corresponds to one of the FORTRAN runtime error numbers, then the error is forced to occur and the normal FORTRAN error message header, including the error number, is written along with the specified message 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 specified is 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 condition numbers, are listed in the Diagnostic Messages for NOS/VE manual. Example:

```
CALL SYSTEM (3, 'CHECK DATA')
```

The FORTRAN error 3 header, the message CHECK DATA, and a traceback are printed immediately upon execution of the CALL SYSTEM statement.

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 FORTRAN error number for which nonstandard recovery is to be implemented. The value must be in the range 0 through 9999. Only FORTRAN runtime library error numbers are supported. Some expressions in the source program generate a reference to an external library of mathematical routines on the system called the Math Library. For math library errors, the equivalent FORTRAN error number for the error must be used.

slist

Six-element integer array containing the following error processing specifications:

element 1	1 = fatal, 0 = nonfatal.
element 2	Print frequency.
element 3	Frequency increment.
element 4	Print limit.
element 5	Recovery routine selector: 0 = Recovery routine not provided. 1 = Recovery routine provided.
element 6	Maximum traceback limit applicable to all errors. Default is 20.

recov

Optional 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. All errors that you define using the listed numbers in a SYSTEM call retain the indicated severity. However, the severity of any FORTRAN 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 element 5 of slist 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 element 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. If elements 2, 3, and 4 of slist are all specified as zero in the last SYSTEMC call executed for an error number, then the error summary for that error number will not be issued at job termination.

A user-supplied error recovery routine should not reference a variable or array of a common block.

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. If a user-supplied error recovery routine is used for input/output error processing, it should not perform input/output operations.

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:
   a. Calling routine is a main program.
   b. 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, except for errors whose last SYSTEMC call had words 2, 3, and 4 of slist set to zero.

Example:

```
EXTERNAL ERRFN
DIMENSION IRAY(6)
DATA IRAY/6*-1/
IRAY(4) = 0
IRAY(5) = 1
CALL SYSTEMC (62, IRAY, ERRFN)
  ⋮
```

These statements suppress printing of error message 62 and transfer control to a user defined 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 J.

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

Character 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$ASCII6_FOLDED collating sequence
ASCII6S	selects the OSV$ASCII6_STRICT collating sequence
COBOL6	selects the OSV$COBOL6_FOLDED collating sequence
COBOL6S	selects the OSV$COBOL6_STRICT collating sequence
DISPLAY	selects the OSV$DISPLAY64_FOLDED collating sequence
DISPLAYS	selects the OSV$DISPLAY64_STRICT collating sequence
DISPLAY63	selects the OSV$DISPLAY63_FOLDED collating sequence
DISPLAY63S	selects the OSV$DISPLAY63_STRICT collating sequence
EBCDIC	selects the OSV$EBCDIC collating sequence
EBCDIC6	selects the OSV$EBCDIC6_FOLDED collating sequence
EBCDIC6S	selects the OSV$EBCDIC6_STRICT collating sequence
INSTALL	selects the OSV$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 J.

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 for the collating sequence currently in effect.)

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 displayed 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.

CHGUCF

The CHGUCF allows you to disable or enable certain system conditions that terminate program execution. You can also use the SET_PROGRAM_ATTRIBUTE command to disable or enable these system conditions from outside your program. This call has the form:

CALL CHGUCF (flag, mode, scope, rtncode)

flag

The name of the NOS/VE User Mask Register condition flag to be changed. Options are:

'DIVIDE_FAULT' or 'DF'

'ARITHMETIC_OVERFLOW' or 'AO'

'EXPONENT_OVERFLOW' or 'EO'

'EXPONENT_UNDERFLOW' or 'EU'

'FP_LOSS_OF_SIGNIFICANCE' or 'FPLOS' or 'FLOS' or 'FP_SIGNIFICANCE'

'FP_INDEFINITE' or 'FPI' or 'FI'

'ARITHMETIC_LOSS_OF_SIGNIFICANCE' or 'ALOS' or

'ARITHMETIC_SIGNIFICANCE'

mode

Logical expression specifying the desired mode for the specified condition flag. Options are:

.TRUE.

The condition is enabled (masked ON) for traps.

.FALSE.

The condition is disabled (masked OFF) for traps.

scope

The scope of reference to be covered by the user mask register setting. Options are:

'LOCAL' ('L')

The specified flag is changed for the procedure (program or subprogram) that called CHGUCF, and all subprograms called by that program or subprogram.

'GLOBAL' ('G')

The specified flag is changed for all procedures on the stack at the time of the call to CHGUCF. All subprograms referenced by the main program, and the main program, are included.

'PREVIOUS'('P')

The specified flag is changed for both the caller of CHGUCF and any other subprograms it calls, and the previous caller (the caller of the caller of CHGUCF) and any other subprograms it calls.

rtncode

Integer variable to receive the resulting return code from the call. The returned integer is one of the following values: on, no previous traps for that condition were outstanding. 0>

Call completed normally. If the flag was to be turned on, this indicates that a previous trap for that condition was outstanding and was cleared before the flag was turned on. (This will be the condition code for PME$SYSTEM_ CONDITION.)

Both upper case letters and lower case letters are allowed in any string value used for flag or scope.

The CHGUCF subprogram changes the specified mask flag (condition bit) to the specified mode in the user mask register(s) for all stack frames within the specified scope. The scope applies both for the setting of the user mask register(s) and for the clearing of any associated outstanding trap conditions in the corresponding user condition register(s) before a condition is to be enabled (turned on). The scope parameter applies to the current and previous calling routines. The setting of the user mask register for a routine holds for that routine and also for any other routines that it calls with subsequent calls or nested calls unless one of those other procedures calls CHGUCF.

Mask and condition register bits not specified are left unchanged. For details on the functioning of traps, the user mask register and user condition register, refer to the Virtual State Hardware Reference Manual.

You must clear the mask and condition register before referencing a FORTRAN-supplied intrinsic function.

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

This chapter describes the FORTRAN command options, the FORTRAN output listings, and the command to execute a FORTRAN program.

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=list of file
BINARY_OBJECT=file
LIST=file
COMPILATION_DIRECTIVES=boolean
DEBUG_AIDS=keyword
DEFAULT_COLLATION=keyword
ERROR=file
ERROR_LEVEL=keyword
EXPRESSION_EVALUATION=list of keyword
INPUT_SOURCE_MAP=list of file
FORCED_SAVE=boolean
LIST_OPTIONS=list of keyword
MACHINE_DEPENDENT=keyword
ONE_TRIP_DO=boolean
OPTIMIZATION_LEVEL=keyword
RUNTIME_CHECKS=list of keyword
SEQUENCED_LINES=boolean
STANDARDS_DIAGNOSTICS=keyword
TARGET_MAINFRAME=keyword
TERMINATION_ERROR_LEVEL=keyword
STATUS=status variable

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_LEVEL=HIGH

FORTRAN SFILE,,,,,,,,,,,,,HIGH
```

The second command specifies the INPUT and OPTIMIZATION_LEVEL parameters positionally; the positions of the omitted parameters are indicated by thirteen 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.

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.

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_AIDS	DA	Debugging options	DEBUG_ AIDS=NONE
DEFAULT_COLLATION	DC	Collating sequence control	DC=FIXED
ERROR	E	File to receive error information	E=$ERRORS
ERROR_LEVEL	EL	Severity level of error messages to be printed	EL=W
EXPRESSION_EVALUATION	EE	Order of evaluation of expressions	EE=NONE
FORCED_SAVE	FS	Save variables and arrays in subprograms	Variables and arrays not saved
INPUT	I	Source input file(s)	I=$INPUT

(Continued)

Table 10-1. FORTRAN Command Parameters *(Continued)*

Parameter Name	Short Form	Description	Default
INPUT_SOURCE_MAP	ISM	PPE source input file	ISM=$NULL
LIST	L	Output listing file	L=$LIST (batch) L=$NULL (interactive)
LIST_OPTIONS	LO	Output listing options	LO=S
MACHINE_DEPENDENT	MD	Machine dependencies	Machine dependencies not flagged
ONE_TRIP_DO	OTD	Minimum trip count for DO loops	Zero trip loops
OPTIMIZATION_LEVEL or OPTIMIZATION	OL OPT	Compiler optimization level	OL=LOW
RUNTIME_CHECKS	RC	Runtime range checking of subscript and substring expressions	No 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
TARGET_MAINFRAME	TM	Mainframe on which the object code is to be executed	TM=C180V on a model 990; TM=C180MI on any other model
TERMINATION_ERROR_LEVEL	TEL	Severity level of errors for which STATUS returned	TEL=F

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. These parameters have one of the forms:

parameter name=file

parameter name = list of file

A file specification identifies a local or permanent file. A file specification consists of a file path (which includes the file name), a cycle reference (for permanent files), and a file position. If a list of files is allowed, separate each file name with a comma or space and enclose the list in parentheses. 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 specifications.

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

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

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 selects no 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. The binary object code will be generated into a single file even if you specified a list of files for the INPUT parameter. Options are:

Omitted

Same as BINARY_OBJECT=$LOCAL.LGO.

BINARY_OBJECT=file

Binary object code is written to the specified file.

BINARY_OBJECT=$NULL

Binary 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_AIDS (DA)

The DEBUG_AIDS parameter selects debugging options. Options are:

Omitted
Same as DEBUG_AIDS=NONE.

DEBUG_AIDS=DT
Generates line number and symbol tables for use by Debug.

DEBUG_AIDS=PC
Generates code to allow for load-time argument checking. Argument mismatch information is written to the loadmap file, regardless of the LOAD_MAP_OPTION on the EXET or SETPA commands.

DEBUG_AIDS=ALL
Selects both DEBUG_AIDS=PC and DEBUG_AIDS=DT options.

DEBUG_AIDS=NONE
No options are selected.

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
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) or ERROR_LEVEL=INFORMATIONAL (EL=I)

Lists trivial (informational) 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. Also causes relational operand expressions involving the .EQ. or .NE. operators to be evaluated using using floating-point comparisons. If not specified, integer (bit-by-bit) comparison used.

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. Also results in the generation of descriptive error messages by internal FORTRAN routines if execution errors occur. If this option is not selected, the operating system produces error messages which generally provide less information.

OVERLAPPING_STRING_MOVES (OSM)

Guarantees valid character assignment in character assignment statements of the form v=exp where the character positions being defined in v are referenced in exp.

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 OL=DEBUG or OL=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 or list of file

Source code to be compiled is contained in the specified file or files. Each file must contain a full program unit and not parts of a program unit. If more than one file is specified, the list of files is enclosed in parentheses.

INPUT_SOURCE_MAP (ISM)

The INPUT_SOURCE_MAP parameter is used when the FORTRAN source program is contained in SCU decks. The ISM parameter specifies the file containing the source map that was generated by the OUTPUT_SOURCE_MAP option on the SCU_EXPD command.

Omitted

Same as ISM=$NULL.

ISM=file

The specified file or files contains the source map. If more than one file is specified, the list of files is enclosed in parentheses.

A program compiled with an ISM file, and DA=DT or ALL, can be used in the full screen Debug utility. For more information about SCU decks, see the Source Code Utility manual.

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. The format of the compiler output listing will be a single file even if you specified a list of file references for the INPUT parameter. Options are:

Omitted

Same as LIST=$LIST (batch jobs) or LIST=$NULL (interactive jobs). (Data written to file $NULL is discarded).

LIST=file

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 parameter is not specified, you must connect the default file by using the SCL command:

```
CREATE_FILE_CONNECTION $LIST $OUTPUT
```

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$ LIST directives are not listed.

SA

Same as S, except that lines turned off by C$ LIST 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) or
MACHINE DEPENDENT=INFORMATIONAL(MD=I)

Machine dependent usages are diagnosed as trivial (informational) 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 establishes the minimum trip count for DO loops. 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 sets the minimum trip count to one. This means that all DO loops will executed at least once, even if the terminating conditions are satisfied before the loop is initially entered. This information enables the compiler to generate more efficient object code. Specifying ONE_TRIP_DO=OFF sets the minimum trip count to zero; this means that DO loops whose terminating conditions are satisfied before the loop is initially entered will not be executed. You can override the ONE_TRIP_DO parameter with the C$ DO directive. Refer to the discussion of DO loops in chapter 5.

For Better Performance

For full optimization of DO loops you should specify ONE_TRIP_DO=ON. You should check all DO loops in your program to ensure that the results will not be affected by executing all DO loops at least once.

OPTIMIZATION_LEVEL (OL, OPTIMIZATION, OPT)

The OPTIMIZATION_LEVEL parameter selects the level of optimization performed by the compiler. Options are:

Omitted

Same as OPTIMIZATION_LEVEL=LOW.

OPTIMIZATION_LEVEL=DEBUG

Object code is similar to that produced with OPTIMIZATION_LEVEL=LOW, except that it is modified for debugging use. Also automatically selects FORCED_SAVE = ON.

OPTIMIZATION_LEVEL=LOW

Minimum optimization is performed, resulting in faster compilation time but slower execution time.

OPTIMIZATION_LEVEL=HIGH

Maximum optimization is performed, resulting in slower compilation time but faster execution time.

At OPTIMIZATION_LEVEL = HIGH, storage is not allocated at load time for variables and arrays unless they are in a common block, or are saved (by a SAVE statement or FORCED_SAVE = ON compiler option), initialized in a DATA statement, or used as actual arguments. Instead, storage is allocated for them on the runtime stack during execution, only when the program unit to which they belong becomes active. This storage is then given up on execution of a RETURN or END statement in the program unit. The default runtime stack size is about 2 million bytes. If this limit is exceeded, a runtime error results, usually of the form "Tried to read/write beyond maximum segment length" and "A stack segment contains invalid frames". For programs where the number of active items allocated on the runtime stack exceeds the default limit, you can increase the runtime stack size by specifying the STACK_SIZE parameter on the EXECUTE_TASK command (as described in the SCL Object Code Management Usage Manual).

For Better Performance

For full optimization (fastest execution), specify OPTIMIZATION_LEVEL=HIGH. But remember that this option results in slower compilation.

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=NONE.

RUNTIME_CHECKS=NONE

Causes no options to be selected.

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=ALL

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)or STANDARDS_DIAGNOSTICS =INFORMATIONAL (SD=I)

Nonstandard usages are treated as trivial (informational) 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 (informational), 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 status variable must have been created by the command:

```
CREATE_VARIABLE variable KIND=STATUS
```

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 three 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.

Condition field

A unique integer value indicating the specific error that occurred. The condition field consists of a unique one to four digit error code prefixed by an identifier. 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.

TARGET_MAINFRAME (TM)

The TARGET_MAINFRAME(TM) parameter specifies the kind of mainframe that the object code is generated for. This parameter is only significant when the OPTIMIZATION_LEVEL parameter specifies HIGH. Options are:

Omitted

Same as C180_VECTOR if compilation occurs on a CYBER 180 Model 990. Same as C180_MODEL_INDEPENDENT if compilation occurs on another model of the CYBER 180.

TARGET_MAINFRAME=C180_VECTOR or C180V

The object code is generated for use on the model 990 of the CYBER 180. The model 990 has vector-processing capabilities; object code produced by this option will also execute on any CYBER 180 model, however, it would perform optimally on a model 990.

TARGET_MAINFRAME=C180_MODEL_INDEPENDENT or C180MI

The object code is generated for use on any model of the CYBER 180.

For Better Performance

Be sure to use the TARGET_MAINFRAME=C180_VECTOR option for code that is going to be executed on a model 990 of the CYBER 180.

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) or
TERMINATION_ERROR_LEVEL=INFORMATIONAL (TEL=I)
Abnormal status is returned for trivial (informational), 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 and catastrophic diagnostics are written to the output listing file.

All other parameters assume default options. (See table 10-1.)

Example:

```
FORTRAN INPUT=MYPROG OPTIMIZATION_LEVEL=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.)

Example:

```
FORTRAN I=(AFILE, BFILE, CFILE) B=BINF DA=ALL
```

This statement selects the following options:

`I=(AFILE, BFILE, CFILE)`	Source statements are read from files AFILE, BFILE and CFILE.
`B=BINF`	The binary object code is written to file BINF. The binary object code is generated from all three input files into one file.
`DA=ALL`	Allows you to use the Debug utility with your program.

Example:

```
/fortran
?       PROGRAM TEST
?       READ *, J
?       PRINT *, J/2
?       END
?*EOI
/
```

This example reads the source input from the terminal (assuming $INPUT is connected to the terminal) and then compiles it. To terminate terminal input, and begin program compilation, enter *EOI (in uppercase) after the prompt.

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. If you supplied a list of files for the INPUT parameter, the format of the compiler output listing is the same as if you had combined the files into a single file. The following paragraphs describe the output listing options.

Source Listing

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$ LIST directives (described in appendix D) to suppress the listing of selected source lines. If you select the LIST_OPTIONS=SA option, C$ LIST 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 in the compiler listing and are not affected by the LIST_OPTIONS parameter unless LIST_OPTIONS=NONE is specified. 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 (Informational 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 (LO) parameter on the FORTRAN command. The options are:

LO=A

Lists all symbolic names in the source program and their attributes (such as size, data type, relative address, and so forth).

LO=R

Lists each symbolic name in the program unit and all references to the name.

LO=(A,R)

Combines the A and R options (lists the attributes and references for each symbolic name).

LO=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. If name is used only as a statement function dummy argument, it is listed as a STF DUMMY ARGUMENT.

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 only character items. BYTES for a block containing character and noncharacter 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.

sec+off

Relative address of the label. If an address cannot be assigned, one of the following symbols appears:

UNDEFINED

Statement label is not defined.

NO REFERENCES

Label is not referenced anywhere in the program unit. (This label can be removed from the source program.)

INACTIVE

Label has been deleted by optimization.

prop

One of the following:

FORMAT

Label appears in a FORMAT statement.

DO-TERM

Label appears in a DO statement.

NON-EX

Label appears on a nonexecutable statement. (If the label is referenced, a diagnostic is issued.)

blank

Label appears in the label field of an executable statement.

def

Source line number where label is defined. *UNDEF* if not defined.

refs

Source line numbers where the label is referenced. (Appears only when R option selected.) May be suffixed by a usage symbol.

DO Loops Map

The DO loops map lists information about all DO loops in the program unit, including implied DO lists in I/O statements. (Implied DO lists in DATA statements are not listed.) The loops are listed in order of their appearence in the program unit. The map appears only if you selected the M option. The DO loop map has the following form:

```
--DO LOOPS--
  -LABEL--SECTION+OFFSET-----PROPERTIES-----INDEX---FROM---TO

   label  sec+off            prop           index   first  last
     :
```

label

Label of final statement; I/O for implied DO lists in input/output statements.

sec+off

Relative address of the first statement of the loop.

prop

One of the following symbols, describing properties of the loop:

OPEN

Loop can be reentered from outside its range.

EXIT

Loop contains references to statement labels outside its range.

XREF

Loop contains references to an external subprogram, including compiler-generated references to all library subprograms except those used to do character moves or compares.

OUTER

Loop contains nested loops.

index

Name of the DO variable.

from

Line number of the first statement of the loop.

to

Line number of the last statement of the loop.

Entry Points Map

The entry points map lists information about subprogram names appearing on ENTRY statements. This map has the following form:

```
--ENTRY POINTS--
  -NAME----SECTION+OFFSET----------ARGS-REFERENCES-

    name    sec+off                 args refs
     ⋮
```

name

Entry point name as defined in the source program.

sec+off

Relative address assigned to the entry point.

args

Number of dummy arguments in the ENTRY statement; blank if no arguments.

refs

Source line numbers of ENTRY or RETURN statements. (Produced only of the R option is selected.) May be suffixed by a usage symbol.

Procedures Map

The procedures map lists names of all functions and subroutines called from the program unit, names declared in an EXTERNAL statement, and names of intrinsic and statement functions appearing in the program unit. Implicit external references generated by certain FORTRAN statements, such as input/output statements, are not listed. The procedures map has the following form:

```
--PROCEDURES--
  -NAME----TYPE----------ARGS-CLASS-----REFERENCES

    name    type          args class     refs
     ⋮
```

name

Subroutine or function name.

type

Data type of function result; blank if class is SUBROUTINE or UNKNOWN; GENERIC for generic intrinsic functions.

args

Number of arguments; VAR if number is variable (intrinsic functions such as MAX and MIN). UNKNOWN if class is UNKNOWN.

class

One of the following:

DUMMY FUNC

Name is a dummy argument used as a function name

DUMMY SUBR

Name is a dummy argument used as a subroutine name

SUBROUTINE

Name is used as a subroutine name

FUNCTION

Name is used as an external function name

FUNC+SUBR

Name used as both a function name and subroutine name (gives a warning message)

INTRINSIC

Name is an intrinsic function name

STAT FUNC

Name is used as a statement function name

UNKNOWN

Class cannot be determined. (Appears if name is used in an EXTERNAL statement.)

refs

Line number where name is referenced. May be suffixed by a usage symbol.

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

Auxiliary 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 Object Code Management manual for a detailed description of the EXECUTE_TASK command and 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 commmand 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.

Condition field

Returns the error code. The error code is an integer value representing the combined ASCII values of a two digit condition identifier and a one through four digit condition code.

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 is the last parameter on the execution command.

Example:

```
CREATE_VARIABLE IERR KIND=STATUS
LGO STATUS=IERR
```

These statements define 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 overridden by file name specifications in OPEN statements.

Keyed-File Interface 11

Keyed-File Interface 11

The keyed-file interface is a group of subprogram calls that use the NOS/VE keyed-file interface to perform input/output operations on keyed files. A keyed file is a file whose organization allows you to access records by their key values.

The following subsection, Keyed-File Concepts, describes keyed-file structure and alternate keys in general. This information applies when using keyed files within or outside of programs. The next subsection, FORTRAN Keyed-File Concepts, describes concepts unique to the FORTRAN keyed-file interface. It is followed by individual descriptions of the keyed-file interface calls and the file information table (FIT) values.

NOTE

Type integer data in files to be processed by the keyed-file interface must be full-word integers, that is, typed explicitly or implicitly as integer*8.

Keyed-File Concepts

Keyed files are like sequential files and byte-addressable files in that the data in the files is contained in records.

A record is a collection of data that is read and written as a unit. The record could contain several fields of data, some of which have a fixed length while others vary in length. Thus, the records as a whole could have a fixed length or be variable in length.

For example, a record could contain three data items of different types: an integer, a floating point number, and a string of characters. To write a record, a program writes all three data items together as a record; when the record is later read, all three data items are delivered to the program.

The records in a sequential or byte-addressable file are stored as a simple sequence. The records in a keyed file are stored within a file structure.

Keyed-File Organizations

A file is a keyed file if its file_organization attribute is either indexed-sequential or direct access. A keyed-file organization allows you to read any record in the file directly by specifying its key value. The key value for a record is determined when the record is written to the file.

To allow you to access each record by a key value, the file organization must relate each key value to the location of the record in the file. The keyed-file interface performs all processing required to relate a key value to a record location; beyond choosing the file organization, the user does not specify how this is done. The method of relating a key value to a record location differs for each keyed-file organization as described in the following sections.

Indexed-Sequential File Organization

The indexed-sequential file organization allows content addressing of records; that is, you can directly access a record by the contents of one or more fields of data in the record. The fields of data by which a record is addressed are its key fields, and the contents of those fields are its key values.

An indexed-sequential file always has a primary key. (It can also have one or more alternate keys as described in the Alternate Keys section of this chapter.)

Each primary-key value is unique within the file; there can be no duplicate primary-key values in a file.

The indexed-sequential file organization is used only when you can assign a unique value to each record stored in the file. This unique value is usually a field of data within the record (an embedded key), although it can be a value assigned to the record and not included in the record data (a nonembedded key).

For example, the primary key for an employee file could be the employee's name. However, because two employees could have the same name, it is better to assign a unique identification number to each employee and use that number as the primary key for the file.

The indexed-sequential file organization should be used if a requirement exists to read file records both sequentially and randomly. For example, the records in an employee file could be read sequentially to produce a listing of all employees or read randomly to update individual records.

When an indexed-sequential file is read sequentially, its records are accessed in ascending order by key value. The order is kept even when new records are added to the file. For example, if an employee file is read sequentially using its primary key (the employee identification number) the records are read in ascending order by their identification number.

Indexed-Sequential File Structure

This subsection gives a general description of the indexed-sequential structure. You can use indexed-sequential files without knowing their structure. However, if you understand the indexed-sequential structure and how it grows, you can create more efficient indexed-sequential files by specifying appropriate values for structural parameters.

The internal structure of an indexed-sequential file is designed to provide both random and sequential access to the data records in the file. File space is divided into blocks, all the same size.

A block contains a block header and one of the following:

- Internal tables
- Data records (a data block)

- Index records (an index block)

Each index record points to a data block. The index record contains the location of the data block and the range of key values of the data records stored in that block.

You can display the contents of all components of an indexed-sequential file, the internal tables and index blocks as well as the data blocks, using the DISPLAY_KEYED_FILE command described in the SCL Advanced File Management Usage manual.

As you might expect, the actual internal index mechanism is complex. The simplified examples in this part, however, provide the level of detail you need to know in order to use indexed-sequential files.

To see how an index works, let's look at a very small file that contains one index block and two data blocks. As shown in figure 11-1, the index block contains two index records. Each index record points to a data block in the file.

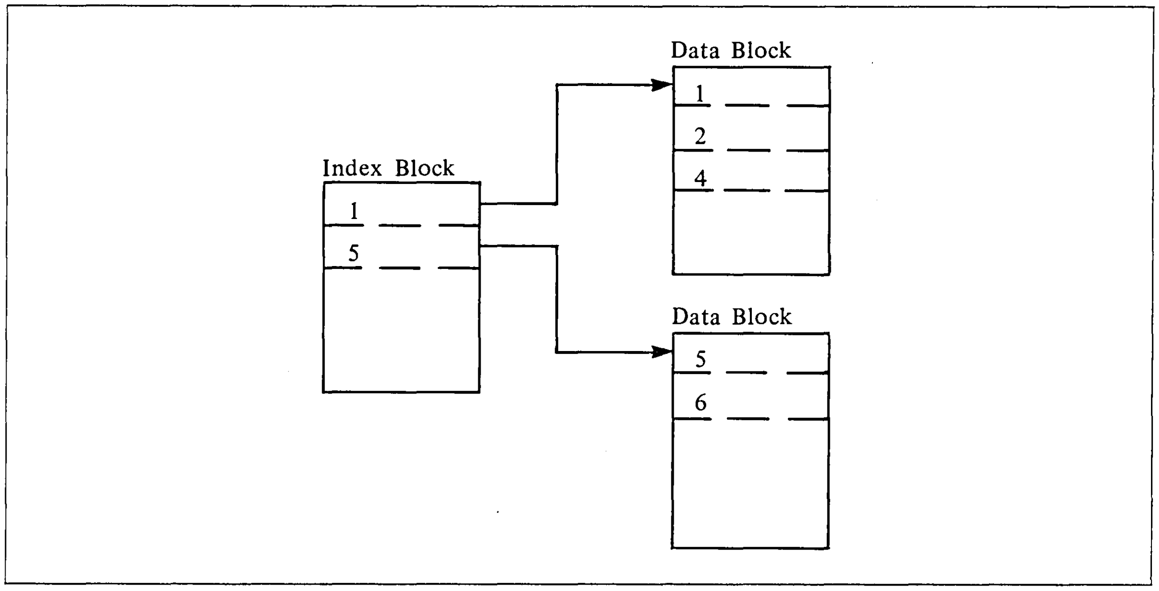


Figure 11-1. Minimal Indexed-Sequential Structure

Let's suppose you request to read randomly the record with key value 6. When the record is read, these steps are performed:

1. The index records are searched to find the index record whose range of key values includes the key value 6.
2. After the correct index record (the second one) is found, the search for the record continues with the data block pointed to by the second index record.
3. The second data block is searched for the record with key value 6. When the record is found, its data is returned to the requestor.

Next, suppose you request that all records in the file be read sequentially. These steps are performed.

1. The first index record is read to find the first data block.
2. The records from the first data block are read in order.

3. The second index record is read to find the second data block.
4. The records from the second data block are read in order.
5. The sequential read ends because there are no more index records and, so, no more data blocks to read.

This process reads the records in key-value order because both the index records and the data records are kept in key-value order.

Data-Block Split

Usually, a block has some empty space, called padding, that was left empty so that additional records could be written later to the block. Suppose, as shown in figure 11-2, that a data block has been filled, a new record is to be written, and its key value is within the range of key values of the records in the full data block. For the file structure to be maintained, the data block must be split.

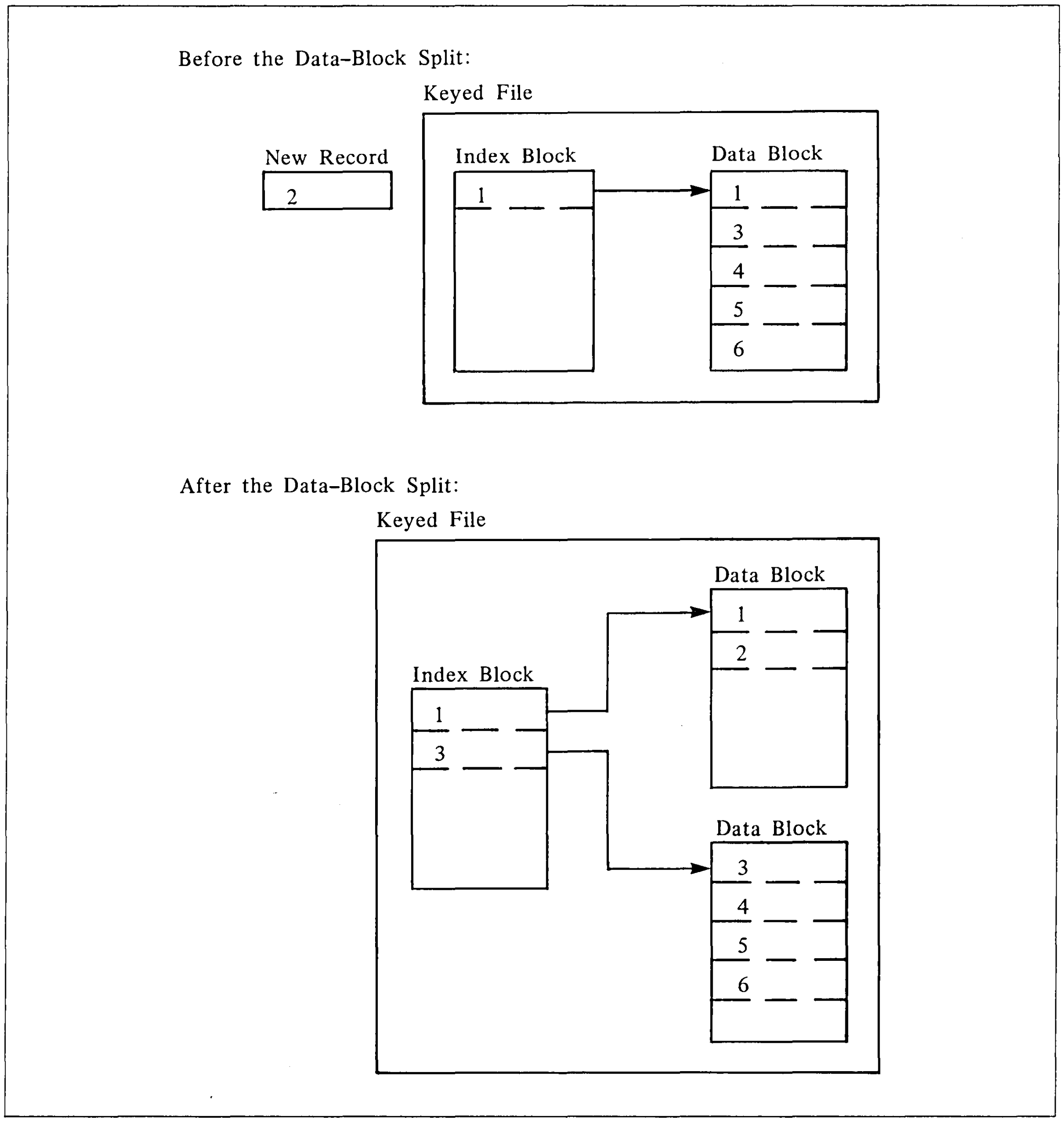


Figure 11-2. Data Block Split

When a data-block split occurs, records in the data block whose key values are less than the key value of the new record remain in the existing block. All records in the existing block that come after the new record are moved to the newly created block.

The new record is put into either the new block or the existing block, depending on the relative 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 that block.

Index Levels

As with data blocks, index blocks may be initially created with some empty space (index-block padding). However, for each new data block created due to a data-block split, another index record must be created. With the addition of many data records, the initial index block becomes full. When the index block is full, the next data-block split causes an index-block split.

As shown in figure 11-3, when the initial index block splits, it causes the creation of another index level.

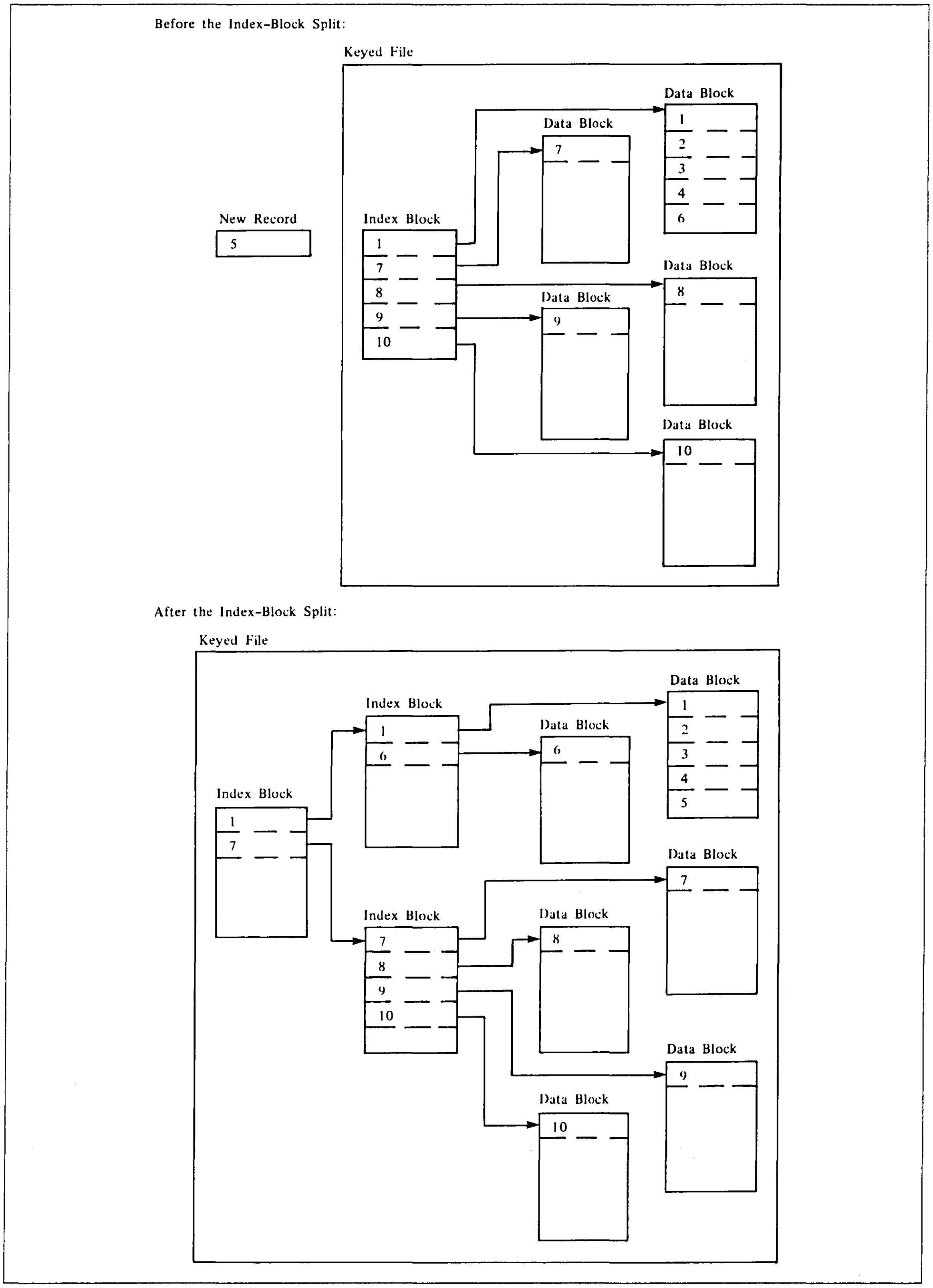


Figure 11-3. Index Block Split

The index levels are numbered from the top down as index level 0, index level 1, and so forth. Index level 0 always has only one index block; it is always the starting point for an index search.

The index block at an upper level contains an index record for each index block at the next lower level. For example, the index block at level 0 contains an index record for each index block at level 1.

A search for a data record requires an index-block search at each index level. For example, the level-0 search finds the index record that points to the appropriate level-1 index block. If the file has only two index levels, the level 1 search finds the index record that points to the appropriate data block.

As you can see, the addition of another index level increases the time required to find an individual data record.

Index levels can be added up to the index-level limit of 15 levels. This sets a limit on the number of records in the file.

The index-level limit is reached when addition of another record to the file would require creation of another index level, but 15 index levels already exist in the file. When this happens, the index-level-overflow flag is set and no more records can be added to the file.

Indexed-Sequential Primary Keys

The primary key for an indexed-sequential file is defined when the file is created. The primary-key value must be unique for each record in the file.

A primary-key definition requires specification of these attributes:

- Embedded or nonembedded key (the default is embedded)
- Key position (if the key is embedded)
- Key length
- Key type (the default is uncollated)
- Collate-table name (if the key type is collated)

A key is embedded if the key value is part of the data in the record. An embedded key value is returned as part of the record data when the record is read; a nonembedded key value is not.

The key position in the record must be specified if the key is embedded. The first byte position in a record is byte 0. If the key is nonembedded, you do not specify a key position.

You must specify the key length whether the key is embedded or nonembedded. It indicates the number of bytes in the key.

The key type describes the data in the key. These are the possible key types:

Integer key
The key value is a signed integer; it is sorted in numerical order.

Uncollated key

The key value is a string of characters; it is sorted byte-by-byte according to the ASCII collating sequence.

Collated key

The key value is a string of characters; it is sorted byte-by-byte according to a collating sequence that you specify.

If the key is a collated key, you must specify the collating sequence to be used to sort the key values. The collating sequence is specified by its name. NOS/VE provides several predefined collating sequences (listed in appendix J). You can also create your own collating sequence as described in appendix H.

Direct Access File Organization

The direct access file organization is like the indexed-sequential file organization in its use of a primary key. You define the primary key for the file when you create the file. It can be a field embedded in the record or a nonembedded value. Each primary-key value in the file must be unique; the file can contain no duplicate primary-key values.

Like an indexed-sequential file, a direct access file can have alternate keys. An alternate key for a direct access file is the same as an alternate key for an indexed-sequential file. Alternate keys are described later in this chapter.

Like indexed-sequential file records, you must specify the primary-key value when writing or deleting a direct access file record. Similarly, you must specify either a primary-key value or an alternate-key value to read a direct access file record.

Direct access and indexed-sequential files differ in the ordering of records in the file:

- When records are read sequentially from an indexed-sequential file, the records are returned in order, sorted by primary-key value.
- When records are read sequentially from a direct access file, the records are returned unordered.

In general, random record access is faster for the direct access file organization than for the indexed-sequential file organization. This is because the direct access file organization determines the location of a record directly from its primary-key value. (In indexed-sequential files, a record can be found only after a search at each index level.)

Direct Access File Structure

The direct access file structure is designed to locate each record directly by its primary-key value. The primary-key value directly specifies the file block containing the record.

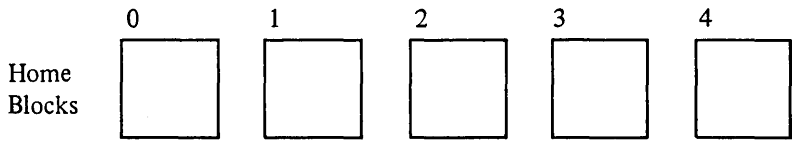
File space in a direct access file is divided into equal-size blocks. Initially, all blocks in the file are home blocks (as opposed to overflow blocks).

When a record is written to a direct access file, its primary-key value is hashed to produce the number of the home block in which the record is written. If the home block does not contain enough empty space for the new record, the record is written to an overflow block.

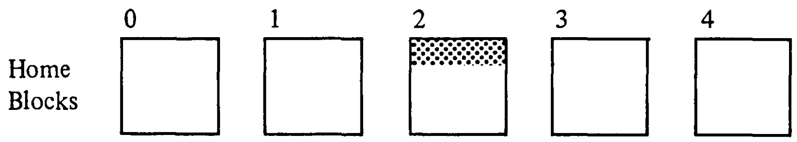
Assuming the hashing procedure produces a uniform distribution of numbers from the primary-key values in the file, the records are uniformly distributed among the home blocks of the file. Thus, each record can be found by a single search of its home block without additional searches of overflow blocks.

You specify the initial number of home blocks when you create the file. By default, a system hashing procedure is used to distribute the records among the home blocks, although you can provide another hashing procedure for the file if you like.

As an illustration of a small direct access file, suppose you define a direct access file as having five home blocks.

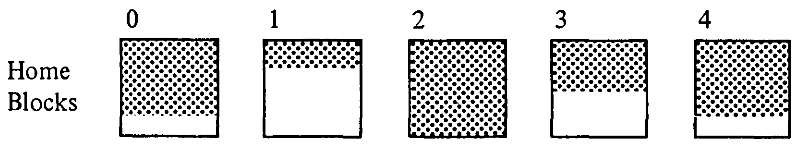


The first record written to the file has primary-key value XYZ. Assume that hashing of this primary-key value produces the block number 2. The record is then written in home block 2.

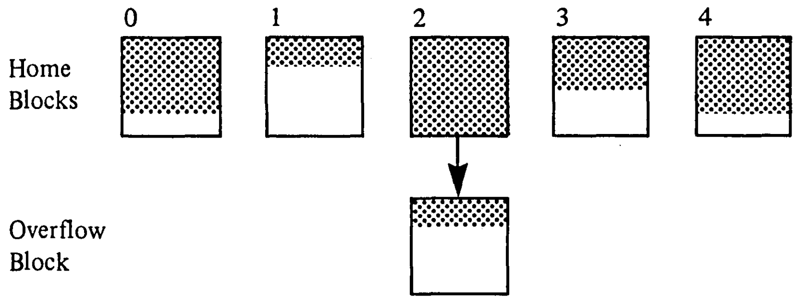


Assume you want to read the record with primary-key value XYZ. The value XYZ is hashed and, as before, produces the block number 2. The keyed-file interface searches for the record with primary-key value XYZ in home block 2. (The records in a block are ordered by primary-key value so each record can be found quickly.)

Suppose that many records have been written to the file and home block 2 has been filled.



At this point, a record is to be written with primary-key value ABC. Hashing of the value ABC produces block number 2, but there is insufficient space for the record in home block 2 so it is written in an overflow block.



Later, to read the record with primary-key value ABC, the primary-key value is hashed to produce block number 2. Home block 2 is searched for primary-key value ABC. When it is not found in the home block, the search continues in the overflow block until the record is found.

An ideal direct access file structure has these characteristics:

- Sufficient home blocks are allocated and records are uniformly distributed among the home blocks so as to avoid overflow.
- Each block contains a limited number of records so as to minimize the search time in each block.
- The number of home blocks is not so large that the file contains excessive unused space.

These characteristics are determined by the file attribute values specified when the file is created.

You must specify the initial_home_block_count and can optionally specify the max_block_length and the hashing_procedure_name attributes. (These attributes are described in CYBIL Keyed-File and Sort/Merge Interfaces, publication number 60464117.)

One other characteristic to be considered when selecting the number of home blocks is the loading factor. The loading factor is the percentage of block space used. To allow for less-than-uniform distribution of records in the home blocks, the loading factor should be no greater than 90%.

You can use the following equations to determine the minimum home_block count for a given loading factor if the number of bytes of data in the file and the block size are known.

If the file has fixed-length records, reduce the block size by 39 bytes, as follows:

$$\text{home\_block\_count} = \frac{\text{record\_count x fixed\_record\_length}}{\text{loading\_factor x (block\_size - 39)}}$$

If the file has variable-length records, reduce the block size by 36 bytes and use the average record length plus 3 as the record length, as follows:

$$\text{home\_block\_count} = \frac{\text{record\_count x (average\_record\_length + 3)}}{\text{loading\_factor x (block\_size - 36)}}$$

To illustrate, suppose the direct access file is to contain 10,000 80-byte records (80,000 bytes of record data). Using a block size of 4096 bytes and a loading factor of 90%, the equation appears as follows:

$$\text{home\_block\_count} = \frac{10000 \text{ x } 80}{.90 \text{ x } (4096\text{-}39)}$$

The equation gives 22 blocks as the minimum home block count for the file. However, it is recommended that the home block count be a prime number so 23 would be a better home block count for the file in this example.

Hashing Procedure

The system provides a default hashing procedure named AMP$SYSTEM_HASHING_PROCEDURE. However, if desired, you may specify your own hashing procedure that produces a uniform distribution of numbers from the primary-key values in your file.

The system executes the hashing procedure each time a record is requested by key value from the direct access file. The hashing procedure is not stored with the file so the system must be able to load the procedure each time the direct access file is opened.

NOTE

Although any ring_attributes value is valid for the object library containing the hashing procedure, in a production environment, you should store the hashing procedure in a ring 4 object library. This improves performance because otherwise hashing procedures is loaded by an asynchronous tasks. (Ring 4 object libraries are maintained usually by site personnel.)

A hashing procedure receives a primary-key value as its input and produces an integer as its output. It must always produce the same output from a given input.

A hashing procedure is written in the CYBIL language. For information on how to write a hashing procedure, see the CYBIL Keyed-File and Sort/Merge Interfaces manual.

The system divides the value it receives from the hashing procedure by the number of home blocks and uses the remainder as the home block number. For example, if the number of blocks is 97, it divides the hashed value by 97 and uses the remainder (an integer from 0 through 96) as the home block number. A more uniform distribution of records can be expected if the number of home blocks is a prime number.

Direct Access Primary Keys

In general, the primary key of a direct access file has the same characteristics as the primary key of an indexed-sequential file. You specify whether the primary key is embedded or nonembedded, its position (if the key is embedded), and the key length. However, a key_type attribute value specified for a direct access file is ignored; the key_type attribute for a direct access file is always uncollated.

Unlike an indexed-sequential file, sequential access calls to a direct access file while the primary-key is selected do not return the file records sorted by primary-key value. The calls return records according to their physical location in the direct access file. Records within each block are ordered according to the default ASCII collating sequence, but the blocks are not ordered by primary-key values.

Direct access file records can be accessed in order if one or more alternate keys are defined for the file. The alternate index keeps the alternate-key values in sorted order. Sequential access calls while an alternate key is selected return records in the order provided by the alternate index.

If appropriate, you could define an alternate key for the same field as an embedded primary key. In this way, you could access direct access file records in primary-key value order.

Alternate Keys

A record within a keyed file can always be accessed by its primary-key value. An alternate key provides an additional way to access records.

An alternate key defines a value in the data record by which the record can be accessed. An alternate key is defined as a field or group of fields in the record.

Although a program can use alternate keys to read records or to position a file, alternate keys cannot be used to write, replace, or delete records. The primary-key value must be used to identify a record to be written, replaced, or deleted.

Alternate-Key Characteristics

Alternate-key fields can overlap each other and the primary key. For example, the primary-key field could be bytes 0 through 9 and two alternate-key fields bytes 0 through 19 and bytes 4 through 14.

Unlike a primary-key value, one alternate-key value can be associated with several records in a file. This is because an alternate-key value need not be unique. The same alternate-key value can occur in several records. For example, the same job title can be associated with many names as follows:

Data Records:	Hanson	Computer Programmer
	Jones	Computer Programmer
	Smith	Computer Programmer

Alternate Index:	**Alternate-Key Value**	**Primary-Key Values**
	Computer Programmer	Hanson
		Jones
		Smith

A record can contain more than one alternate-key value if the alternate key is defined as a field that repeats in the record; thus, a single record could contain several alternate-key values. For example, the license numbers of several cars owned by one person as follows:

Data Record: `R. Petty   1 LB AU   2ASM451   ELK 592`

Alternate Index:

Alternate-Key Value	**Primary-Key Values**
1 LB AU	R. Petty
2ASM451	R. Petty
ELK 592	R. Petty

The Alternate Index

The index for the primary key was described earlier in this chapter. Each alternate key defined for a file has its own index.

An alternate index contains index records, each of which associates an alternate-key value with the primary-key values of the records containing that alternate-key value. The list of primary-key values associated with an alternate-key value is the key list for that alternate-key value.

When you select an alternate key and then specify an alternate-key value, the system searches for the value in the alternate index. If it finds the alternate-key value, it uses the primary-key values in the key list for the alternate-key value to access the data records.

When one or more alternate keys are defined for a file, file updates require more time because the alternate indexes must also be updated. Alternate keys should be used only when the additional record access capability offsets the cost of increased time spent for file updates.

Alternate-Key Definition

The attributes of an alternate key are specified by its alternate-key definition.

These attributes are required to define an alternate key:

- Key name
- Key position
- Key length

An alternate key has a name so that it can be selected for use. The alternate-key position and length define the alternate-key field within the record.

These optional attributes define how the alternate key is processed:

- Key type
- Collate table name (if the key type is collated)
- Duplicate key values
- Null suppression

- Sparse-key control
- Repeating groups
- Concatenated key
- Variable-length key

The key type of an alternate key determines the order of the alternate-key values in the alternate index, and therefore, the order in which records are accessed sequentially when you use the alternate key. The key types for an alternate key are the same as the key types for the primary key as described earlier in this chapter.

If the key type is collated, you can explicitly specify a collation table for the alternate key or use, as the default, the collation table for the primary key (if one has been specified).

Duplicate Key Values

By default, duplicate values for an alternate key are not allowed. However, if you want to allow duplicate key values, you can specify whether the records having the same alternate-key value are accessed in primary-key-value order or in first-in-first-out order.

In a key list ordered by primary key, the primary-key values are stored in sorted order according to the primary-key type. New values are added to the key list so that the primary-key-value order is kept.

In a key list ordered first-in, first-out, the primary-key values are stored in the key list in the order the values are added to the key list, instead of in primary-key-value order. New values are always added to the end of the key list.

For Better Performance

When alternate-key values are frequently duplicated in a file, the key lists should be ordered by primary-key value. First-in, first-out ordering of key lists requires that delete and replace operations sequentially search the key list to find the primary-key value of the updated record; a sorted key list provides faster access to a primary-key value.

For example, suppose you write three records to the file in this order:

```
McDarrels          Hamburgers
Burger Duke        Hamburgers
Willys             Hamburgers
```

The following shows the resulting key list in primary-key order and in first-in-first-out order:

	Key Lists	
Alternate Key Value	**Ordered by Primary Key**	**First In First Out**
Hamburgers	Burger Duke	McDarrels
	McDarrels	Burger Duke
	Willys	Willys

Duplicate-Key Value Error Processing

If duplicate values are not allowed and a duplicate is found in a record about to be written to the file, the record is not written to the file and a nonfatal error (status AA2100) is returned.

A nonfatal error (status AA2865) also occurs if a duplicate value is found while a new alternate index is being created. However, the record containing the duplicate value cannot be discarded, as it is already in the file. Subsequent processing depends on whether incrementing the nonfatal-error count causes the count to exceed the nonfatal-error limit as set by the user.

- If the nonfatal-error limit is not exceeded, the apply operation redefines the alternate key to allow duplicates, ordered by primary-key value, discards the partially built index, and builds the redefined index.
- If the nonfatal-error limit is reached, the apply operation returns AA2870 and removes all alternate indexes it has created. (Deleted indexes are not restored.)

In either case, a message describing the action taken is written to the $ERRORS file.

Null Suppression

By default, if an alternate-key field contains a null value, the null value is stored as the alternate-key value for the record. The null_suppression attribute allows you to exclude null values from an alternate index.

Null suppression excludes any record with a null alternate-key value from the alternate index. Null suppression can save space, access time, and update time because the index is smaller when null alternate-key values are excluded. (Null suppression does not remove the null value from the data record.)

The null value depends on the key type as follows:

Key Type	Null Value
Integer	Zero
Uncollated	Spaces
Collated	Spaces (before collation)

If null suppression is not specified, records containing a null value in the alternate-key field are indexed by the null value. The records can later be accessed by specifying the null value as the alternate-key value.

For example, suppose the spouse's name is defined as an alternate key to a membership file. Unmarried members would have a null value for the alternate-key field. Therefore, the key list for the null value lists all unmarried members. The following shows the alternate index with and without null suppression:

Without Null Suppression

Spouse's Name	Member's ID
	1626736 8273648
Diana Simmons	4872672
Mark Ramsey	7726184
Shelly Gable	2673651

With Null Suppression

Spouse's Name	Member's ID
Diana Simmons	4872672
Mark Ramsey	2673651
Shelly Gable	7726184

Sparse-Key Control

You can use sparse-key control to create an alternate index that includes or excludes records depending on the character in a specific position in the record.

For example, suppose a student file has a one-character code indicating the student's class. To get a mailing list for juniors and seniors only, you could define an alternate index controlled by the class code.

To specify sparse-key control, you specify three values:

Value	Example
Sparse-key control position	Position of the class code in the record
Sparse-key control characters	Junior and senior class code characters
Sparse-key control effect (Indicates whether the alternate-key value should be included or excluded if the sparse-key character matches)	Included if the class code indicates a junior or senior record

Assume that the sparse-key control position is the first character after the name field and that the junior and senior class codes are 3 and 4. If the following records are copied to the file, the first three records are included in the alternate index, but not the last record.

```
Louis Skolnik    4
Gilbert Sullivan 4
Elliot Wermzer   3
Judy Manhasset   2
```

The sparse-key control position must be within the minimum record length. If you specify sparse-key control for an alternate key, the alternate-key field or fields need not be within the minimum record length.

A nonfatal (trivial) error (status AA2875) is returned if both of these conditions are true for a record:

- The character at the sparse_key_control_position indicates that the record should be included in the alternate index.
- The record has no alternate-key value because the record is too short to contain the entire alternate-key value.

When an apply or write operation detects this error, it does not include the record in the alternate index. (A write operation does write the record to the file.)

Concatenated Keys

A concatenated key is an alternate key formed from several fields, or pieces, in the record. A concatenated key can comprise up to 64 pieces.

The concatenated pieces can be noncontiguous and can be concatenated in any order. Each piece can be a different key type. All collated-key pieces use the same collation table.

The RMKDEF call cannot create a concatenated key; to create a concatenated key in a FORTRAN program, you must use the SCLCMD call to execute the CREATE_ALTERNATE_INDEXES utility. (The CREATE_ALTERNATE_INDEXES utility is described in the SCL Advanced File Management Usage manual.)

The first piece you specify is the leftmost piece of the key. You specify it the same as you specify a nonconcatenated key. The pieces to be concatenated to the leftmost field are defined by individual ADD_PIECE subcommands. The subcommand order specifies the order of the concatenated pieces.

A concatenated key can use sparse-key control and/or null suppression. A concatenated key is considered to have a null value if the values in all fields of the key are null (before collation for collated keys).

For example, suppose you decide to define an alternate key consisting of the initials of the member's name. The first piece of the key value would be the first letter of the member's first name, the second piece would be the first letter of the member's middle name, and the third piece would be the first letter of the member's last name. Consider this data record:

0	20	40
Kennedy	John	Fitzgerald

The alternate-key value is JFK, assuming the concatenated-key pieces are defined as:

First piece:	`Key_Position=20, Key_Length=1`
Second piece:	`Key_Position=40, Key_Length=1`
Third piece:	`Key_Position=0, Key_Length=1`

Repeating Groups

The repeating-groups attribute allows a data record to contain more than one value for the same alternate key. This allows a primary-key value to be associated with more than one alternate-key value.

To specify an alternate-key field within a repeating group:

1. Specify the first alternate-key field by its key position, key length, and key type. All subsequent alternate-key fields have the same length and type as the first.

2. Specify repeating groups for the alternate key by specifying the repeating group length, that is, the distance from the beginning of the first instance of the alternate key to the beginning of the second instance of the alternate key in the record.

3. Specify the repeating-group count, that is, how many times the alternate-key field repeats in the record.

You can specify that the repeating group repeats a fixed number of times or that it repeats until the end of the record.

- If the alternate-key field repeats a fixed number of times, all alternate-key fields must be within the minimum record length.

- If the alternate-key field repeats to the end of the record, the minimum record length imposes no restriction. The system stores as many alternate-key values as the record length allows.

Repeating groups cannot be used with concatenated keys or when duplicate-key values are allowed and ordered first-in, first-out.

For example, suppose each record in a membership file lists the sports the member enjoys and his years of experience as follows (columns are counted from zero):

Field:	Sports and Sports Experience
Columns:	Variable number of 2-field pairs beginning at column 75
	The Sports field is 10 characters followed by a 2-digit Sports Experience field
Type:	ASCII characters

You could define an alternate key for the Sports values (without the Sports-Experience values) as follows:

CREATE_KEY_DEFINITION parameters:

```
Key_Position=75, Key_Length=10, Key_Type=uncollated, Repeating_Group_Length=12,
Repeating_Group_Count=repeat_to_end_record,
Duplicate_Key_Values=ordered_by_primary_key
```

RMKDEF call:

```
CALL RMKDEF(fit, 0, 75, 10, 0, 'UNCOLLATED', 'ORDERED_BY_PRIMARY_KEY', 12, 0)
```

The key list for an alternate-key value would list the identification numbers of all members that enjoy that sport.

The following shows the primary keys for three records and their contents from column 75 to the end of the record:

Primary Key	Record Contents Beginning at Column 75
1662876	`Volleyball02Running   03Basketball02`
6166287	`Bicycling 10Volleyball01`
0027840	`Running   15Running   15Running   15`

If these were the only records in the file, the alternate index would appear as follows:

Alternate Key Value	Primary Key Values
Basketball	1662876
Bicycling	6166287
Running	0027840 1662876
Volleyball	1662876 6166287

Notice that the key type is the default (uncollated) and the duplicate-key values specification is ordered_by_primary_key. Thus, each key list is sorted according to the default ASCII collating sequence.

Notice also, as shown by the Running key list, each primary-key value is listed only once in a key list, regardless of the number of times the alternate-key value occurs in the record.

Variable-Length Key

A variable-length alternate key is an alternate key whose values vary in length. Its alternate-key definition specifies its starting position, its maximum length, and its set of delimiter characters.

The end of a variable-length key value is marked by a delimiter character, the end of the key field, or the end of the record, whichever is found first starting at the key_ position.

By defining the key as a variable-length key, you can use the following values as alternate keys:

- The first value beginning at a certain position of each record.
- The last field in a variable-length record.
- All data in a variable-length record.

By defining the key as a variable-length key with the repeating groups attribute, you can use the following values as alternate keys:

- A value found anywhere in a fixed-length field (if all other characters in the field are in the set of delimiter characters for the alternate key).
- Each value in a sequence of values, separated by one or more consecutive delimiter characters. The sequence of values can be within:
  - A fixed-length field
  - A variable-length field at the end of the record
  - The entire record

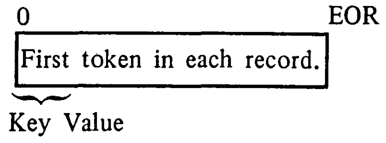
For Better Performance

Define a key as a variable-length key only when necessary. The requirement to scan the key field for delimiter characters adds processing time when the alternate index is built and when the file is updated.

The following examples each specify a variable-length alternate key.

Example 1:

The alternate key is to be the first sequence of up to 80 non-blank characters in each record.



To define the alternate key, specify the key position as 0, the key length as 80, and the variable-length key attribute with the blank character as the delimiter, as follows:

CREATE_KEY_DEFINITION parameters:

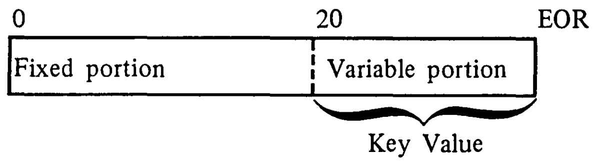
```
Key_Position=0, Key_Length=80, Variable_Length_Key=' '
```

RMKDEF Call:

```
CALL RMKDEF(fit,0,0,80,0,0,0,0,0,0,0,0,0,' ')
```

Example 2:

Assume that each record consists of a required 20-byte portion followed by an optional variable-length portion of up to 120 bytes.



To define the variable-length portion as the alternate key, specify the key position as 20, the key length as 120, and the variable-length key attribute with an empty delimiter set.

The statements to define the key are the same as for example 1 except for the following:

CREATE_KEY_DEFINITION parameters:

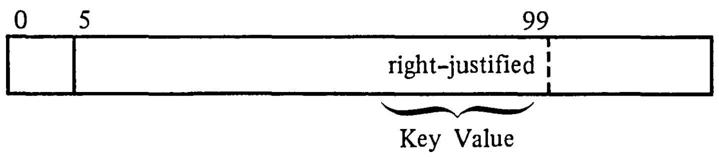
```
Key_Position=20, Key_Length=120, Variable_Length_Key=''
```

RMKDEF Call:

```
CALL RMKDEF(fit,0,20,120,0,0,0,0,0,0,0,0,0,'')
```

Example 3:

Assume a 100-byte field at byte 5 contains one value that is to be used as the alternate key. The value is from 0 through 100 bytes long, right-justified and blank-filled within the field.



To define the alternate key, specify the key position as 5, the key length as 100, the variable-length key attribute with the blank character as the delimiter, and the repeating_groups attribute.

The repeating_groups attribute is required because the value is right-justified in the field; thus, the search for the value must not end at the first delimiter; it should continue to the end of the field. For a repeating variable-length key, the repeating_group_length value can be any integer greater than zero; the repeating_group_count is the length of the alternate-key field.

CREATE_KEY_DEFINITION parameters:

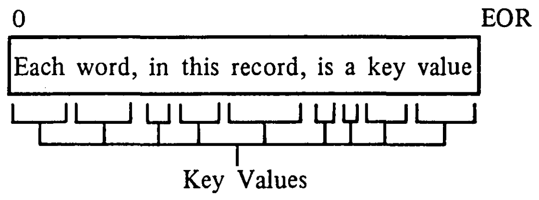
```
Key_Position=5, Key_Length=100, Variable_Length_Key=' ',
Repeating_Group_Length=1, Repeating_Group_Count=100
```

RMKDEF Call:

```
CALL RMKDEF(fit,0,5,100,0,0,0,1,100,0,0,0,0,' ')
```

Example 4:

Each string of letters in the data is to be defined as a value for the alternate key.



To define the alternate key, specify the key position as 0, the key length as the maximum record length (80), the variable-length key attribute, and the repeating_groups attribute. Notice that the delimiter set is defined as all characters except the letters.

CREATE_KEY_DEFINITION commands:

```
create_variable, key delimiters, kind=(string,76),..
  value=' 1234567890-=!@#$%^&*()_+[]`{}";''\:"!,./<>?'..
//$CHAR(000)//$CHAR(001)//$CHAR(002)//$CHAR(003)..
//$CHAR(004)//$CHAR(005)//$CHAR(006)//$CHAR(007)..
//$CHAR(008)//$CHAR(009)//$CHAR(010)//$CHAR(011)..
//$CHAR(012)//$CHAR(013)//$CHAR(014)//$CHAR(015)..
//$CHAR(016)//$CHAR(017)//$CHAR(018)//$CHAR(019)..
//$CHAR(020)//$CHAR(021)//$CHAR(022)//$CHAR(023)..
//$CHAR(024)//$CHAR(025)//$CHAR(026)//$CHAR(027)..
//$CHAR(028)//$CHAR(029)//$CHAR(030)//$CHAR(031)..
//$CHAR(127)

create_key_definition, key_name=words, ..
  key_position=0, key_length=80, ..
  variable_length_key=key_delimiters, ..
  repeating_group_length=1, ..
  repeating_group_count=repeat_to_end_of_record
```

RMKDEF Call:

```
      value=' 1234567890-=!@#$%^&*()_+[]`{}";''\:"!,./<>?'
     + //$CHAR(000)//$CHAR(001)//$CHAR(002)//$CHAR(003)
     + //$CHAR(004)//$CHAR(005)//$CHAR(006)//$CHAR(007)
     + //$CHAR(008)//$CHAR(009)//$CHAR(010)//$CHAR(011)
     + //$CHAR(012)//$CHAR(013)//$CHAR(014)//$CHAR(015)
     + //$CHAR(016)//$CHAR(017)//$CHAR(018)//$CHAR(019)
     + //$CHAR(020)//$CHAR(021)//$CHAR(022)//$CHAR(023)
     + //$CHAR(024)//$CHAR(025)//$CHAR(026)//$CHAR(027)
     + //$CHAR(028)//$CHAR(029)//$CHAR(030)//$CHAR(031)
     + //$CHAR(127)
      CALL RMKDEF(fit,0,0,80,0,0,0,1,0,0,0,0,0,value)
```

Attributes Incompatible With Variable-Length Keys

The following alternate-key attributes are not supported for variable-length keys:

- Integer key type
- Ordering duplicate-key values chronologically (First_In_First_Out)
- Concatenation
- Null suppression
- Sparse-key control

Using a Variable-Length Key

Using a variable-length alternate key differs from using a fixed-length key in the following ways:

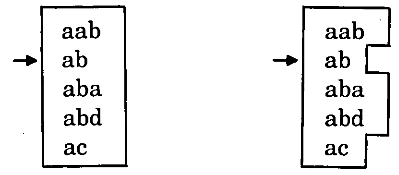
- On a call using a variable-length key, you must specify the length of the key value as well as its location. The length of the key value is specified using the appropriate major-key length parameter.
- When a call returns a variable-length key value, it returns the value padded with delimiter characters to the full key length. (It pads using the lowest character in the delimiter set.)
- The key value specified on the call is compared with the full key value stored in the index, not only the leftmost bytes.

Key value comparison is illustrated by the following example that contrasts the use of a variable-length key value with the use of an equivalent major-key value for a fixed-length key. The key value used is the leftmost two bytes ('ab'):

File Position in the Alternate Index

Parameter Specifications **Fixed-Length** **Variable-Length**

Key Value: 'abb'
Key_Relation: 'Equal'
Major_Key_Length: 2



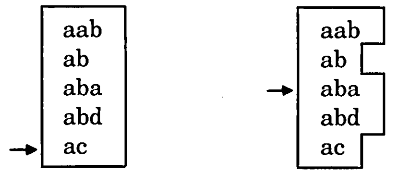
As shown, when the Key_Relation is 'Equal', the positioning is the same.

However, the positioning can differ if the Key_Relation is 'Greater_Than':

File Position in the Alternate Index

Parameter Specifications **Fixed-Length** **Variable-Length**

Key_Value: 'abb'
Key_Relation: 'Greater_Than'
Major_Key_Length: 2



The file positioning differs because:

- The two-byte major-key value is compared with the leftmost two bytes of the fixed-length alternate-key values. So, the file is positioned at the first key value whose leftmost two bytes are greater than 'ab', that is, 'ac'.
- The two-byte variable-length key value is compared with the full variable-length alternate-key value, not just the leftmost two bytes. So, the file is positioned at the first key value greater than 'ab', that is, 'aba'.

Nested Files

A nested file is a file structure defined within a NOS/VE file cycle. It is recognized and used by the keyed-file interface; it is not recognized or used by the NOS/VE file system.

The keyed-file interface provides nested files so as to extend the NOS/VE limit on the number of files a task can use. All nested files defined in a file share the same memory segment. This provides effective memory use when the nested files are much smaller than the segment size limit (2^{32} bytes).

All Nested files in a file share the same NOS/VE catalog entry. Thus, if one nested file is damaged, the entire file is damaged and requires recovery.

The keyed-file interface creates the initial nested file (named $MAIN_FILE) when it creates the keyed file. It uses $MAIN_FILE as the default nested file; other nested files are used only when explicitly selected.

No FORTRAN keyed-file interface call exists to create a nested file. However, a FORTRAN program can create a nested file (other than $MAIN_FILE) by calling the CYBIL subprogram AMP$CREATE_NESTED_FILE or by calling the SCL command COPY_KEYED_FILE to copy an existing nested file or by calling the CREATE_KEYED_FILE utility. (The AMP$CREATE_NESTED_FILE call is described in the CYBIL Keyed-File and Sort/Merge Interfaces manual. The COPY_KEYED_FILE command and the CREATE_KEYED_FILE utility are described in the SCL Advanced File Management manual.)

A FORTRAN program selects a nested file by storing its name in the FIT using the keyword $NESTED_FILE_NAME (or $NFN). To re-select the default nested file, it stores the name $MAIN_FILE.

Each alternate-key definition applies to only one nested file. To define an alternate key for a nested file other than the default nested file ($MAIN_FILE), you first select the nested file and then define the alternate key. Similarly, to select an alternate key for a nested file other than the default nested file ($MAIN_FILE), you first select the nested file and then select the alternate key.

A task can perform operations only on the currently selected nested file. However, the file position, key selection, and locks for a nested file are not lost when another nested file is selected. For example, consider this sequence of events:

1. A task is issuing GETN calls while NESTED_FILE_1 and ALTERNATE_KEY_1 are selected.

2. The task selects and uses NESTED_FILE_2.

3. The task selects NESTED_FILE_1 again. It can continue reading records sequentially from the file position at which it stopped reading when it selected NESTED_FILE_2. The same key, ALTERNATE_KEY_1, remains selected.

FORTRAN Keyed-File Interface Concepts

This subsection describes how the keyed-file interface described in this manual differs from the other NOS/VE keyed-file interfaces (such as the SCL keyed-file utilities described in the SCL Advanced File Management Usage manual).

If you have used CYBER 170 Advanced Access Methods Version 2 (AAM 2), you may want to read about the differences between NOS/VE AAM 2 and the NOS/VE keyed-file interface. These differences are described in appendix C.

Although it is called the FORTRAN keyed-file interface in this manual, the interface can be used by other languages (such as COBOL) that use the standard calling sequence.

NOTE

Do not use more than one I/O method to process the same file. In particular, do not process the same file using both language statements and keyed-file interface calls.

NOTE

When a program written in a language other than FORTRAN or COBOL uses FORTRAN keyed-file interface calls, you must add the following object library to the program library list before executing the program:

```
$LOCAL.AAF$4DD_LIBRARY
```

For example, the following SET_PROGRAM_ATTRIBUTES command adds the object library to the program library list.

```
set_program_attributes, add_libraries=$local.aaf$4dd_library
```

For more information about the program library list, see the SCL Object Code Management manual.

File Information Table

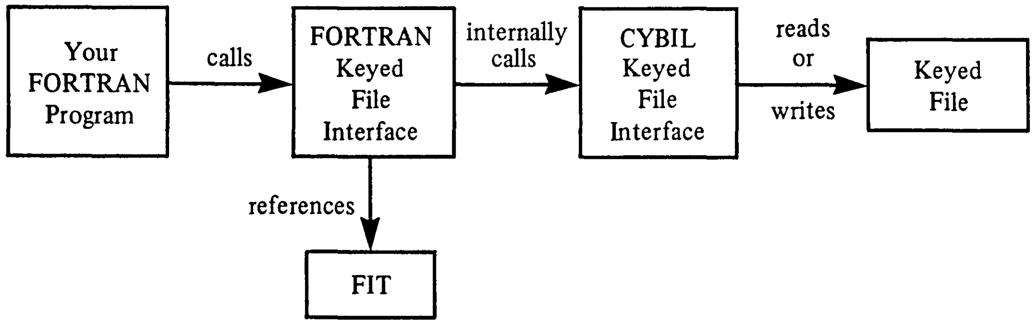
The FORTRAN keyed-file interface references values in a file information table (FIT) to determine how to process a keyed file.

To use a keyed file in your FORTRAN program, you first call the FILEIS or FILEDA subprogram to create a FIT for the file. (FILEIS for an indexed-sequential file; FILEDA for a direct-access file.) (NOS/VE allocates system space for the table; your program does not reserve space for it.)

The FILEIS or FILEDA call stores a pointer to the FIT in a variable you specify on the call. Each subsequent keyed-file interface call for the file specifies the FIT pointer variable as its first parameter.

You can set FIT values using STOREF calls and fetch FIT values using IFETCH calls. FIT values are described in detail under FIT Values later in this section.

This figure illustrates how your program can access keyed-file data.



Keyed-File Interface Error Processing

When a keyed-file interface call (other than the FILEIS or FILEDA call) detects an error, it performs these steps:

1. Sets the $ERROR_STATUS value in the FIT to the status condition code of the error.

2. Sets the fatal/nonfatal (FNF) flag in the FIT to indicate whether the severity of the error is fatal or nonfatal.

3. Writes the error message to the $ERRORS file (if indicated by the $MESSAGE_CONTROL value).

   (If the status severity is warning or informational, the keyed-file interface performs only step 3, writing the message.)

4. For a nonfatal error, it increments the $ERROR_COUNT value in the FIT and compares the $ERROR_COUNT value and the $ERROR_LIMIT value.

   If it finds that the $ERROR_COUNT value is equal to the $ERROR_LIMIT value, it changes the $ERROR_STATUS value to the fatal error code AA3255 (error limit reached) and processes the new error (starting at step 2).

5. If an error-exit procedure is specified in the FIT, it calls the procedure.

   The error-exit procedure should fetch the FNF flag to determine if the error is fatal. If the error is fatal, it should close the file because further file processing is not allowed after a fatal error. (Any calls [except CLOSEM or FLUSHM] issued after a fatal error cause a catastrophic error.)

A FORTRAN program can specify an error-exit procedure by these methods:

- By specifying the error-exit procedure as the $ERROR_EXIT_NAME value before the file is opened.
- By specifying the error-exit procedure as the $ERROR_EXIT_PROCEDURE value (before or after the file is opened).
- By specifying the error-exit procedure as a parameter on a keyed-file interface call.

If the error-exit procedure is specified by the $ERROR_EXIT_NAME value, it becomes effective only when the file is opened. Otherwise, the error-exit procedure becomes effective when it is specified.

If no error-exit procedure has been specified, the keyed-file interface does not call an error-exit procedure when it detects an error. It stores the $ERROR_STATUS value in the FIT, but the program must check the $ERROR_STATUS value after each call.

To check for an error, the program calls IFETCH to check the $ERROR_STATUS value. If IFETCH returns a nonzero value, it indicates that the call did not complete successfully, and the program should take the appropriate action.

The error-exit procedure or the program can fetch the FNF value from the FIT to determine if the error severity was fatal or nonfatal. It can also use the $ERROR_STATUS value to determine the exact status condition returned.

In one instance, the keyed-file interface clears the $ERROR_STATUS value when it returns from an error-exit procedure.

If a call specifies a working storage area, key area, or primary-key area that is not in a common block, the keyed-file interface detects the error and begins the error processing steps described earlier.

It writes an error message to the $ERRORS file (if requested by the $MESSAGE_CONTROL value) and calls the error-exit procedure (if one is specified in the FIT). If the error-exit procedure fetches the $ERROR_STATUS value, IFETCH returns the value AA2535.

However, unlike other errors, when it finishes processing this error, the keyed-file interface clears the $ERROR_STATUS value so that the get or put operation can complete.

Creating a Keyed File

A FORTRAN program to create a keyed file must perform these steps (using the indicated subprogram call):

1. Create a FIT containing appropriate file attribute values (FILEIS or FILEDA and STOREF)

2. Open the file (OPENM)

3. Optionally, write records to the file (PUT)

4. Close the file (CLOSEM)

The calls listed in parentheses are described individually under Keyed-File Interface Calls.

You specify the keyed-file attribute values before opening the new keyed file. The file attributes can be specified by one or more of the following:

- The FILEIS or FILEDA call that creates the FIT for the file

- One or more STOREF calls after the FILEIS or FILEDA call

- One or more SET_FILE_ATTRIBUTE commands executed before the program creating the file is executed. (Values specified by a SET_FILE_ATTRIBUTE command override values specified by FILEIS, FILEDA, and STOREF calls.)

If you do not specify a keyed-file attribute by one of these means, a default value is used when the file is opened.

Keyed-File Attributes

You can specify keyed-file attributes as FIT values. The individual FIT value descriptions are at the end of the keyed-file interface section. The keyed-file attributes are as follows:

- File organization attribute:

  $FILE_ORGANIZATION (required)

- Record attributes:

  $RECORD_TYPE (default, undefined [U])
  $MAXIMUM_RECORD_LENGTH (required)
  $MINIMUM_RECORD_LENGTH (recommended if the record length is variable)

- Primary-key attributes:

  $EMBEDDED_KEY (default, embedded)
  $KEY_LENGTH (required)
  $KEY_POSITION (default, 0, the leftmost byte)
  $KEY_TYPE (default, uncollated)
  $COLLATE_TABLE_NAME (required if the key type is collated)

- File structure attributes:

  $RECORD_LIMIT
  $MAXIMUM_BLOCK_LENGTH

- Indexed-sequential structure attributes:
  $DATA_PADDING (default, 0%)
  $INDEX_PADDING (default, 0%)
- Direct access structure attributes:
  $INITIAL_HOME_BLOCK_COUNT
  $HASHING_PROCEDURE_NAME
- Block-length guideline attributes (specify instead of $MAXIMUM_BLOCK_LENGTH)
  $AVERAGE_RECORD_LENGTH
  $ESTIMATED_RECORD_COUNT
  $INDEX_LEVELS
  $RECORDS_PER_BLOCK
- Processing attributes:
  $COMPRESSION_PROCEDURE_NAME
  $ERROR_LIMIT (default 0, no limit)
  $LOCK_EXPIRATION_TIME (default, 60,000 milliseconds)
  $MESSAGE_CONTROL (default, only fatal and catastrophic error messages)
- Recovery attributes:
  $FORCED_WRITE (default, unforced)
  $LOG_RESIDENCE (default, none)
  $LOGGING_OPTIONS (default, none)

The keyed-file attributes are described in the SCL Advanced File Management Usage manual. The complete SET_FILE_ATTRIBUTES command description is in the SCL Quick Reference manual.

NOTE

Besides the required keyed-file attributes, a FORTRAN program must also set the $LOCAL_FILE_NAME value in the FIT. If the $LOCAL_FILE_NAME value has not been specified, the OPENM call returns a fatal error.

Using an Existing Keyed File

A FORTRAN program to process an existing keyed file must perform these steps (using the indicated subprogram call):

1. Create a FIT containing appropriate values (FILEIS or FILEDA and STOREF)

2. Open the file (OPENM)

3. Perform the intended operations on the file (described next)

4. Close the file (CLOSEM)

Only temporary file attributes can be specified for an existing keyed file. Preserved file attributes are stored with the file and copied to the FIT by the OPENM call.

The calls listed in parentheses are described individually under Keyed-File Interface Calls.

These operations can be performed on an open keyed file:

- Fetch and store FIT values (IFETCH, STOREF)
- Position the file (REWND, SKIP, STARTM)
- Read records (GET, GETN)
- Write records (PUT, PUTREP)
- Replace records (REPLC, PUTREP)
- Delete records (DLTE)
- Flush modified file blocks to disk (FLUSHM)
- Request locks (LOCKF, LOCKK)
- Clear locks (UNLOCKF, UNLOCKK)
- Create alternate keys (RMKDEF)
- Select keys and nested files (STOREF)
- Fetch alternate key information (KLCOUNT, KEYLIST, KLSPACE)
- Build and use result sets to read records (RSBUILD, RSCLEAR, RSCLOSE, RSCOMB, RSDLTE, RSGETN, RSINFO, RSOPEN, RSPUT, RSREWND, RSSKIP, and RSSTART)

Alternate Key Creation

The recommended method for creating alternate keys is to use the SCL utility CREATE_ALTERNATE_INDEXES. In general, using the utility is easier and more efficient than writing a program especially when creating more than one alternate key.

You can execute the SCL utility from a FORTRAN program using the SCLCMD call. The SCLCMD call is described in chapter 9. CREATE_ALTERNATE_INDEXES is described in the SCL Advanced File Management Usage manual.

For compatibility with FORTRAN 5, NOS/VE FORTRAN also supports the RMKDEF call to create an alternate key in your program. Its processing is compatible with the CYBER 170 AAM RMKDEF call.

The RMKDEF call both defines the alternate key and applies the definition to the keyed file to build the alternate index.

The RMKDEF call can be issued for a keyed file that has been created before or during program execution. Both a FILEIS (or FILEDA) call and an OPENM call must be executed before the RMKDEF call. The RMKDEF call uses the FIT pointer returned by the FILEIS (or FILEDA) call.

If the file contains data at the time of the RMKDEF call, the RMKDEF call builds the alternate index. If the file contains no data, the RMKDEF call does not build the alternate index; the index is built as data is later written to the file.

The alternate key created by the RMKDEF call remains as part of the keyed file for the life of the file or until the alternate key is explicitly deleted. You can delete an alternate key using the SCL utility CREATE_ALTERNATE_INDEXES.

Alternate-Key Use

You can use an alternate key to position or read a keyed file. (Calls to write to a keyed file must specify primary-key values, not alternate-key values.)

While an alternate key is selected, the file is positioned and records are read in the logical record order defined by the alternate index. For example, each GETN call reads the next record in alternate-key order, instead of in primary-key order.

Selecting a Key

To indicate that the key values on subsequent STARTM, GET, and GETN calls are alternate-key values, you must call STOREF to select the alternate key. The key selection takes effect when the next START, REWND, OR GET (but not GETN) call is issued.

The STOREF call can specify a key by its name or by its position and length in the record. (Selection by name is recommended; selection by position and length is provided for CYBER 170 AAM 2 compatibility.)

Key Selection by Name

The STOREF call can select a key by storing the key name in the FIT.

For example, the following STOREF call selects alternate key ALTERNATE_567_9_250.

```
   CALL STOREF(fit,'$KEY_NAME','ALTERNATE_567_9_250')
```

To change the key selection, you call STOREF again, specifying another alternate key or the primary key. The primary key name is $PRIMARY_KEY. For example, the following call selects the primary key:

```
   CALL STOREF(fit,'$KEY_NAME','$PRIMARY_KEY')
```

Selection by key name is the only way to select a nonembedded primary key.

Key Selection by Position and Length

A NOS/VE FORTRAN program can also select an alternate key using the same keywords used by a CYBER 170 program: STOREF calls can select a key by storing its position (using the RKW and RKP keywords) and its length (using the KL keyword). To determine the key position, the RKW value (default, 0) is multiplied by 10 and then added to the RKP value.

For example, selecting alternate key ALTERNATE_567_9_250 by position and length requires either the three calls on the left or the two calls on the right:

```
   CALL STOREF(fit,'RKW',567)     CALL STOREF(fit,'RKP',567*10+9)
   CALL STOREF(fit,'RKP',9)       CALL STOREF(fit,'KL',250)
   CALL STOREF(fit,'KL',250)
```

Besides alternate keys, STOREF calls can also specify an embedded primary key by its position and length. The keywords used are the same as for alternate-key selection.

Specifying an Alternate-Key Value

You can specify an alternate-key value either at the location specified by the $KEY_ ADDRESS (ka) value or in the working storage area.

If you specify the value in the working storage area, you must store the value in the alternate-key position in the working storage area. If the alternate key is a concatenated key, each piece must be stored in its field in the record.

For example, suppose you define your working storage area as an 80-integer array named WSA. If the alternate-key field is the fifth integer (that is, the alternate-key field begins at byte 40 [counting from zero] and is 8 bytes long), you could store the integer alternate-key value 1374 as follows:

```
WSA(5)=1374
```

The file-position values returned, and their meanings, differ when using an alternate key, instead of the primary key, as follows:

FP Value	Meaning
1	The file is positioned at the beginning of the alternate index. (It is positioned to read the record with the lowest alternate-key value.)
8	The file is positioned at the end of the key list for the current alternate-key value. (It is positioned to read the first record having the next alternate-key value.)
16	The file is positioned at the end of a record, but not at the end of the key list. (It is positioned to read the next record having the current alternate-key value.)
64	The file is positioned at the end of the alternate index. (It cannot read a record at this position.)

When reading a file sequentially, the program should call IFETCH to fetch the file position and then check the returned value after each get call.

To get all records having the same alternate-key value, the program issues GETN calls until a file position of 8 (end-of-key-list) is returned.

When a GET or GETN call returns a file position of 64, it has positioned the file at its end-of-information and no GETN calls should be issued until the file is repositioned.

A GETN call issued after a call that positions the file at the end-of-information is an attempt to read beyond the end-of-information. It returns a trivial error ($ERROR_ STATUS value AA2635).

Key Values Returned

You can fetch both the alternate-key value and the primary-key value from the FIT while an alternate key is selected.

- A GETN call issued while an alternate key is selected returns the alternate-key value of the record read in the key area, instead of the primary-key value.
- A GETN (or GET or STARTM) call issued while an alternate key is selected can return the primary-key value of the record read in a primary-key area.

Before a call can return a value in a primary-key area, the program must store in the FIT the location of the primary-key area.

For example, this call specifies the variable PRIKEY as the primary-key area:

```
CALL STOREF(fit,'$PRIMARY_KEY_ADDRESS',prikey)
```

NOTE

Like the key area and the working-storage area, the primary-key area should be in a common block.

The primary-key area is used only while an alternate key is selected; no value is returned in the primary-key area while the primary key is selected.

Collated Key Values

If the key type of the key is COLLATED, the key value returned may no longer be the key value input with the record. This can occur if the collation table assigns the same collation weight to more than one character code.

The process is as follows:

1. Each character of a collated key value is stored in the index as the lowest character code having the same collating weight.
2. When the key value is returned, the key value is decollated to its original form. However, if more than one character code is collated as the same value, the value returned is the lowest character code with the same collation weight.

Because of this process, your program may not be able to fetch a nonembedded primary-key value in its original form. (It can always fetch an alternate-key or embedded primary-key value in its original form from the record data.)

For example, if lowercase letters are collated as equal to the corresponding uppercase letters (each lowercase letter is given the same collating weight as the corresponding uppercase letter), the alternate-key value is returned using only uppercase letters.

As another example, consider the OSV$xxxx collation tables predefined by NOS/VE. These collation tables assign collation weight 0 to all unprintable characters and to the space character. Thus, all unprintable characters and all space characters are returned as the lowest character code value with collation weight 0, that is, the unprintable NUL character (00 hexadecimal).

Fetching Information From the Alternate Index

Your program can fetch information from the alternate index using the KLCOUNT, KEYLIST, and KLSPACE calls.

- The KLCOUNT call returns the number of primary-key values for a range of alternate-key values in the alternate index.
- The KEYLIST call returns the actual primary-key values for a range of alternate-key values.
- The KLSPACE call returns the alternate-index block count for a range of alternate-key values.

These calls differ from the other keyed-file-interface calls in these ways:

- Values must be specified for all parameters. (The valid values are listed in the parameter descriptions.)
- The only values that these calls update in the FIT are the file position, the last operation, and the error status. The calls do not use FIT values as default parameter values.

Keyed File Sharing

A permanent keyed file can be shared; a temporary keyed file cannot be shared. A keyed file is shared when multiple concurrent instances of open of the file exist and more than one instance of open could be changing the file.

The possibility of sharing determines whether NOS/VE must safeguard the keyed-file structure for multiple users:

- While more than one instance of open could be changing the file, NOS/VE performs internal locking operations to maintain the integrity of the file structure.
- While only one instance of open can be changing the file, NOS/VE does not perform internal locking; the overhead required to maintain file integrity is not needed, resulting in better file access performance.

File sharing is controlled by the set of share modes in effect for the file. A keyed file cannot be shared when the global share mode set for the file is empty, that is, when the job has attached the file for exclusive access.

The default share mode set depends on the access mode set for the open. If the access mode set includes append, modify, or shorten access, the default share mode set is empty (exclusive access); otherwise, the default share mode set is read and execute.

For Better Performance

For better performance when using a keyed file, check that the share modes allowed are no more than those required. If possible, allow no sharing of the file.

In general, when the file can be shared (the Global_Share_Modes value is not none) and either the Access_Modes or the Global_Share_Modes include shorten or append access, locking is needed. The following examples show two situations in which locking is not needed and a third situation in which it is needed.

1. When reading a keyed file, it is recommended that you request modify access so that read statistics can be recorded in the file. Because modify is one of the write access modes, no other instances of open can access the file while you read it (if you do not explicitly specify Share_Modes).

   In this case, because no sharing is allowed, no locking is performed and performance is at its best.

2. Next, to allow other users to read the keyed file and maintain accurate read statistics, you explicitly specify the Share_Modes as read and modify.

   In this case, sharing is allowed, but the file data cannot be changed. So again, no locking is performed and performance is at its best.

3. Suppose that the permit applicable to the attach allows all access modes to the file, but requires that shorten and append share modes be allowed. You choose to request all access modes and allow all share modes.

   In this situation, other instances of attach, as well as this one, can write, replace, and delete records. Because of the potential for file sharing, NOS/VE uses internal locks as needed to maintain the integrity of the file structure. A program using the file in a shared situation such as this may choose to use locks to disallow changes to data it is currently using. Record deletions and replacements require locking of the primary-key value of the record.

The reasons for using locks and the means of doing so are described in detail in the following pages.

Locks

Keyed-file sharing is coordinated through the use of locks. A lock is a mechanism by which a task can restrict use of a keyed file or individual primary-key values in keyed files. The lock is owned by a particular instance of open for the file.

The part of the NOS/VE system software that manages locks is called the lock manager. In general, lock processing follows this pattern:

1. The lock manager receives a request for a lock on a nested file or record.
2. The lock manager determines whether the lock can be granted.
   a. If no conflicting lock exists, the lock manager grants the lock and notifies the requesting task.
   b. If a conflicting lock exists, the lock manager checks if the request specified waiting.
      1) If the request specified no waiting, the lock manager notifies the task requesting the lock that the record or file is currently locked.
      2) If the request specified waiting, the task is suspended until either:
         a) The lock is available (assuming no potential deadlock as described later under Lock Deadlock), or
         b) The timeout period elapses (default value, 60 seconds).

The lock manager also processes requests to clear locks and keeps track of locks that have expired (as described under Lock Expiration and Clearing).

NOTE

In general, when the Locks discussion describes two or more tasks requesting locks, the two or more tasks could actually be the same task with two or more instances of open of the same file. This is because a lock belongs to a particular instance of open and one task could be requesting locks for more than one instance of open.

Reasons for Locks

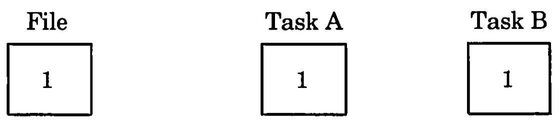
Locks are recommended for effective sharing of a keyed file. In fact, when more than one instance of open exists for a keyed file, NOS/VE requires that a task lock the record before it can replace or delete the record.

Lock use ensures that:

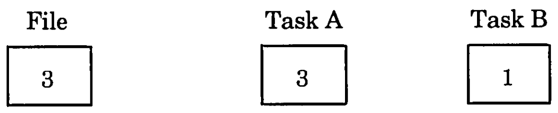
- Requests are processed in the sequence in which requests are issued.
- The operation is performed on the most up-to-date version.

To illustrate the need for locks, the following sequence of events describes two tasks using the same nested file without locks.

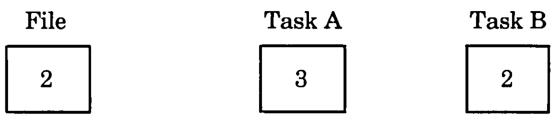
1. Two tasks both read the same record containing the value 1.



2. One task adds 2 to the value and replaces the record, containing the value 3, in the file.



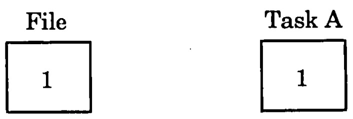
3. The other task adds 1 to the value and replaces the record, containing the value 2, in the file.



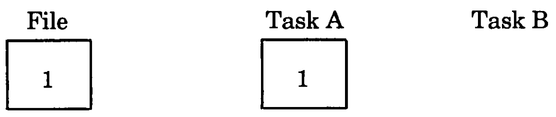
The work of one of the tasks has been overwritten.

Next, consider the alternative in which locks are used.

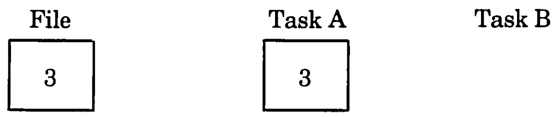
1. A task locks and reads a record.



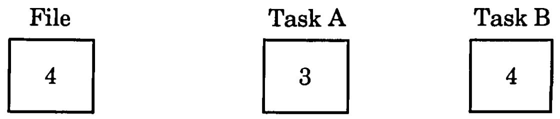
2. A second task attempts to lock and read the record but cannot because the record is already locked. It waits until the record is unlocked.



3. The first task adds 2 to the value, and replaces the record containing the value 3, in the file. It then unlocks the record.



4. The second task can now lock and read the record. It adds 1 to the value, and replaces the record, containing the value 4, in the file.



Lock Intents

Each lock has a lock intent. The lock intent indicates why the task is requesting the lock.

When more than one instance of open exists for a keyed file, only the owner of an Exclusive_Access or Preserve_Access_and_Content lock on the record (or the file) can replace or delete the record. However, the replace or delete operation does not take place until no unexpired Preserve_Content locks exist for the record.

Lock intents for file locks are described later under File Locks. The following lists describe the lock intents for record locks.

Exclusive_Access

- Used when the task intends to issue write or delete requests for the locked primary-key value. The instance of open must have shorten or append access to the file.
- Denies all requests by other tasks to read, write, update, or delete the record or lock its primary-key value.

Preserve_Access_and_Content

- Used when the task might issue write or delete requests for the locked primary-key value. Only one Preserve_Access_and_Content lock is allowed at a time for a key value.
- Allows positioning and read requests by other tasks, but denies their attempts to write, replace, or delete using the locked key value.
- Allows Preserve_Content lock requests by other tasks, but denies their requests for an Exclusive_Access or Preserve_Access_and_Content lock on the primary-key value.
- The owner of the Preserve_Access_and_Content lock can request a write, replace, or delete operation, but:
  - The write, replace, or delete operation does not begin until the conditions for an Exclusive_Access lock are met:
    - All read operations in progress for the record have completed.
    - All Preserve_Content locks for the record have expired or been cleared.

  - No read operations for the record can begin until the write, replace, or delete operation completes.

Preserve_Content

- Used when the task does not intend to issue write, replace, or delete requests for the locked primary-key value.
- If more than one instance of open exists, a Preseve_Content lock prevents all update attempts, including those of the lock owner. However, if the Preserve_Content lock owner is the only existing instance of open, the lock does not prevent updates.
- Allows positioning and read requests by other tasks, but denies their write, replace, and delete requests.
- Allows Preserve_Content and Preserve_Access_and_Content locks by other tasks, but denies their Exclusive_Access lock requests.

  Multiple Preserve_Content locks are allowed at a time, but only one Preserve_Access_and_Content lock. Thus, multiple tasks can be reading the record, but only one task can be waiting to write, replace, or delete the record.

Lock Renewal and Lock_Intent Changing

The owner of a lock can renew the lock by issuing a lock request without an intervening unlock request. The lock renewal restarts the expiration time for the lock.

The lock renewal can also change the lock_intent from Preserve_Access_and_Content to Exclusive_Access and vice versa.

An instance-of-open owning a Preserve_Content key lock or file lock cannot be granted an Exclusive_Access or Preserve_Access_and_Content file lock until it unlocks its Preserve_Content lock.

Depending on the lock_intents, a request for a lock that you already hold may result in an error. To see the possible outcomes, see Lock Conflict Tables at the end of this locking discussion.

File Locks

Your program should request a file lock when it needs locks on many key values at the same time. A file lock is a lock on all primary-key values for a nested file.

In general, the rules for using file locks are the same as those for locks on individual primary-key values.

The effect of the lock intent of a file lock is as follows:

- Exclusive_Access

  Used when the nested file is to be updated.

  Allows access to records in the nested file only by the instance of open holding the file lock; all requests by other instances of open are denied including all lock requests.

- Preserve_Access_and_Content

  Used when the instance of open intends to read records in the nested file and may update records later. It allows the holder to do updates, but prevents all other instances of open from updating.

  Allows all instances of open to read the file and allows Preserve_Content locks for records in the file or the file as a whole, but denies all Exclusive_Access and Preserve_Access_and_Content locks (except a file lock for the nested file by the same instance of open).

- Preserve_Content

  Used to prevent file updates if the file is shared. (The lock owner can update the file if no other instance of open exists.)

  Allows any number of Preserve_Content locks and one Preserve_Access_and_Content lock for each primary-key value and for the file as a whole, but denies all Exclusive_Access lock requests.

For further details on the file and key-value locks that can co-exist, see Lock Conflict Tables.

A file lock is required when your program needs more than 1024 locks at a time because 1024 is the maximum number of locks allowed for an instance of open. An attempt to exceed this limit returns the nonfatal $ERROR_STATUS value AA2115.

The number of locks allowed also depends on the FILE_LIMIT attribute value. The lock manager tracks all locks for a file in another file called the lock file (named $SYSTEM.AAM.AAF$LOCK_FILE). The lock file size cannot exceed 90% of the FILE_LIMIT value and, if an operation would cause the lock file to be more than 50% full, the operation is not allowed to begin and the fatal $ERROR_STATUS value AA6010 is returned. (A second system file, $SYSTEM.AAM.DEPENDENCY_FILE, is also required for locking.)

Waiting for a Lock

When a conflicting lock exists, but no deadlock, a call requesting a lock waits for the lock if the $WAIT_FOR_LOCK value in the FIT is TRUE.

A lock request waits until the lock is available or the lock timeout period has passed. If the lock request times out, the call returns the $ERROR_STATUS value AA2055.

The default timeout period is 60 seconds. However, each task can specify how long it waits for a lock by creating and initializing an SCL integer variable named AAV$RESOLVE_TIME_LIMIT. The value assigned to the variable is the new lock timeout period in seconds (any integer greater than 1). Do not set the lock timeout period so that it is longer than the LOCK_EXPIRATION_TIME attribute value (default, 60 seconds).

For example, the following call executes the SCL command CREATE_VARIABLE to change the timeout period to 45 seconds:

```
call sclcmd ('create_variable, name=AA$RESOLVE_TIME_LIMIT, kind=integer,
value=45')
```

Lock Expiration and Clearing

An expired lock and a cleared lock are not the same:

- A cleared lock no longer exists; the lock manager has discarded it.
- An expired lock is no longer effective in preventing access by other tasks. However, an expired lock prevents file access by its owner (except IFETCH and STOREF calls and an UNLOCKF or UNLOCKK call that clears the expired lock). This is done so that the owner of the lock is notified of its expiration.

A lock is cleared when one of these events occurs:

- The task with the lock issues an unlock request for the lock.
- The task closes the instance of open to which the lock applies.
- The request for the record lock specified automatic unlock, and the task issues any request for the instance of open (other than an IFETCH or STOREF call).

  In general, the automatic unlock occurs when the request is issued. The exception is for an update request for the locked record for which the lock is kept until the update operation completes.

  For example, if a task issues a lock on record 1 and then issues a request to replace record 1, the lock manager automatically clears the lock on record 1 after the replace operation. Similarly, if a task issues a lock on record 1 and then issues a request to position the file at record 2, the lock manager automatically clears the lock on record 1, before positioning the file at record 2.

How a Lock Expires

A lock expires when the following sequence of events occurs:

1. Its expiration time has passed since the lock was granted.
2. Another task issues a request specifying waiting that would be denied if the lock was effective. (The request is granted.)

The number of milliseconds in the lock expiration time is specified by the file attribute, LOCK_EXPIRATION_TIME. The default value is 60,000 milliseconds (60 seconds). To set an unlimited expiration time so that locks do not expire, set the attribute value to 0.

An expired lock is no longer effective in preventing access to the file or record by other tasks. However, it does prevent the task holding the expired lock from accessing records in the file.

The task holding the expired lock is prevented from accessing any record in the file until it clears the expired lock. This notifies the task that a lock has expired.

For example, consider the following sequence of events:

1. Task 1 requests a Preserve_Access_and_Content lock on record 1 in nested file 1 without automatic unlock. The lock is granted.
2. The expiration time passes.

3. Task 1 reads record 1 from nested file 1. The read request restarts the expiration time count.

   (The lock has not yet expired because no other task has issued a request for the record that a Preserve_Access_and_Content lock should prevent. The lock is not unlocked because automatic unlock was not requested for the lock.)

4. The expiration time passes again.

5. Task 2 requests a Preserve_Content lock on record 1 in nested file 1. (The Task 1 lock does not expire because a Preserve_Access_and_Content lock does not prevent Preserve_Content locks.)

6. Task 3 requests, with waiting, a Preserve_Access_and_Content lock on record 1 in nested file 1. (The Task 1 lock expires because a Preserve_Access_and_Content lock should prevent additional Preserve_Access_and_Content locks.)

7. Task 1 attempts to read record 2 in nested file 1, but instead the request terminates with a nonfatal error, notifying Task 1 that it has an expired lock. Task 1 must clear the expired lock before it can successfully request any record in nested file 1.

A task is notified of lock expiration for the currently selected nested file only. The expiration of locks in a previously selected nested file does not affect the task unless it re-selects the nested file and attempts a file operation.

Expired Lock Conditions

These are the nonfatal $ERROR_STATUS values returned for an expired lock:

AA2820

The operation failed due to a leftover expired lock in the nested file.

AA2790

A key value could not be automatically unlocked due to an expired lock.

AA2805

The key value could not be locked due to an expired lock.

AA2810

A lock with a time limit could not be changed to a lock with no time limit due to an expired lock.

AA2815

The first primary-key value in the key list for an alternate-key value could not be locked due to an expired lock. This status can be returned only if the alternate key allows duplicate values, ordered by primary key, and while the task is waiting for the lock, another task inserts a primary-key value at the beginning of the key list.

Lock Deadlock

A deadlock is a situation in which two or more tasks need a lock already held by another task in the group of tasks. For example, the following situation is a deadlock:

- Task 1 has a lock on record 1 and needs a lock on record 2.
- Task 2 has a lock on record 2 and needs a lock on record 3.
- Task 3 has a lock on record 3 and needs a lock on record 1.

If none of the tasks releases the lock it holds, none of the tasks can complete.

A deadlock can occur either when tasks are waiting for a lock or when tasks are each repeatedly requesting a lock. The lock manager can detect the deadlock when the tasks are actually waiting for a lock; it cannot detect a deadlock when tasks are repeatedly requesting locks.

When the lock manager receives a lock request indicating that the task wants to wait until the lock is available, it checks for a possible deadlock. To do so, it checks whether other tasks are waiting for locks held by the requesting task. If it detects a potential deadlock, it terminates the request with a nonfatal error.

If the deadlock is with another task, it returns error AA2040. If the deadlock is a self-deadlock (the requesting task already has the requested lock), it returns error AA2045.

To prevent a deadlock that the lock manager cannot detect, a task should limit the number of times it repeatedly requests a lock without waiting. After a fixed number of attempts, it should do one of the following:

- Issue a lock request with waiting in which case the lock manager can notify it that a potential deadlock exists.
- Assume that a potential deadlock exists and clear the locks it holds.

Lock Conflict Tables

The outcome of a request for a lock that has already been granted depends on:

- The lock intents of the existing and requested locks.
- Whether the request is from the same instance of open holding the lock.
- Whether the conflicting locks are key-value locks or file locks.

This table gives the outcomes when the requested and existing locks are either both key-value locks or both file locks.

When the requested and existing locks are either both key-value locks or both file locks:

If this lock exists:		**And a lock with this intent is requested for the same key or file:**		
Lock Intent	**Instance of Open**	**Preserve_ Content**	**Preserve_ Access_and Content**	**Exclusive_Access**
Preserve_ Content	Same Open Another Open	Renews Grants	Rejects Grants	Rejects Depends
Preserve_ Access_ and_ Content	Same Open Another Open	Rejects Grants	Renews Depends	Renews Depends
Exclusive_ Access	Same Open Another Open	Rejects Depends	Renews Depends	Renews Depends

Legend:

Grants the lock.
Renews the lock, restarting its lock expiration time and changing the lock intent if requested.
Rejects the request (nonfatal status AA2080)
Self-Deadlock returns the nonfatal status AA2045.
Depends as follows:

- No waiting requested: Returns nonfatal status AA2075
- Waiting requested: Grants the lock unless:
  - Opens belong to the same task: Returns nonfatal status AA2045
  - Opens belong to different tasks: Grants the lock unless:
    - Deadlock detected (nonfatal status AA2040)
    - Timeout period elapses (nonfatal status AA2055)

This table gives the outcomes when the existing and requested locks are not the same kind of lock (file locks or key-value locks).

When the existing and requested locks are not the same kind of lock (file locks or key-value locks).

If this lock exists:		**And a lock with this intent is requested for the same key or file:**		
Lock Intent	**Instance of Open**	**Preserve_ Content**	**Preserve_ Access_and Content**	**Exclusive_Access**
Preserve_ Content	Same Open	Grants	Grants	Self-Deadlock
	Another Open	Grants	Grants	Depends
Preserve_ Access_ and_ Content	Same Open	Grants	Self-Deadlock	Self-Deadlock
	Another Open	Grants	Depends	Depends
Exclusive_ Access	Same Open	Self-Deadlock	Self-Deadlock	Self-Deadlock
	Another Open	Depends	Depends	Depends

Legend:

Grants the lock.
Renews the lock, restarting its lock expiration time and changing the lock intent if requested.
Rejects the request (nonfatal status AA2080)
Self-Deadlock returns the nonfatal status AA2045.
Depends as follows:

- No waiting requested: Returns nonfatal status AA2075
- Waiting requested: Grants the lock unless:
  - Opens belong to the same task: Returns nonfatal status AA2045
  - Opens belong to different tasks: Grants the lock unless:
    - Deadlock detected (nonfatal status AA2040)
    - Timeout period elapses (nonfatal status AA2055)

Result Sets

A result set is a set of primary-key values. It provides a means of reading a logical set of records from a keyed file.

A result set begins as a list of primary-key values, retrieved using a key-value range for the currently selected key. (The range is specified in the same way as the range on an KLCOUNT call.) However, unlike a simple key list, a result set can be combined and modified.

A result set can be combined with other result sets (using the logical operations AND, OR, and XOR). It can be modified by adding key values to and deleting key values from the result set. Also, the result set can be used to read either the set of records in the result set or (for indexed-sequential files) all records not in the result set.

The following is a general outline of the steps by which a program creates and uses a result set to read a set of records:

1. Open the keyed file. If the result set is to be for a nested file other than the default nested file ($MAIN_FILE), select the nested file (that is, store the FIT value for $NESTED_FILE_NAME).

2. Open the result set file by calling RSOPEN. The result_ set_ID returned by the call is used by subsequent calls to reference the open result set.

3. Clear the result set using the call RSCLEAR if the existing result set in the file is to be discarded.

4. Add records to and remove records from the set by calling:

   RSBUILD

   Gets the set of records from the keyed file within the range of key values specified on the call and combines the new set with an existing result set.

   (If an alternate key is to be used, it must be stored as the $KEY_NAME FIT value before the RSBUILD call. This is required for a direct-access file because a key-value range cannot be specified for a direct-access primary key because the primary-key values are not ordered in a direct-access file.)

   RSCOMB

   Combines two existing result sets.

   RSPUT

   Adds a single primary-key value to a result set.

   RSDLTE

   Deletes a single primary-key value from a result set.

5. Select the primary key by storing the value $PRIMARY_KEY as the $KEY_NAME FIT value if an alternate key is currently selected.

6. Read records from the keyed file by calling RSGETN to either:

   - Read the records that are in the result set.

   - Read the records that are not in the result set (indexed-sequential files only).

7. Fetch information about the result set at any time while the result set is open by calling RSINFO.

8. Reposition the result set (if appropriate) using the following calls:

   RSREWND

   Position the result set at its beginning.

   RSSTART

   Position the result set at the specified record.

   RSSKIP

   Position the result set forward or backward a specified number of key values. This can be done only after the result set position has been established by a get, rewind, or start result set call.

9. Close the result set by calling RSCLOSE.

10. Close the keyed file.

Result Set Validity

A result set can only be used to read the file for which it was built. The global file name and the currently selected nested file are stored in the result set when it is first opened.

A result set cannot be used with a copy of the original data file or another cycle of the file or another nested file in the data file. For example, if a result set is built for a temporary file, the result set cannot be used to read a permanent copy of the file. This is becuase the permanent copy has a different global file name.

Result sets to be combined must apply to the same nested file and file cycle. However, more than one key for the nested file can be used to build a result set. For example, the result set could be started while the primary key is selected and then added to after selection of an alternate key.

The correctness of a result set is ensured only until the nested file is updated. At that time, key values of records referenced by the result set could be changed. When writing a program that uses result sets, you must determine whether the result set must be correct when it is used to read records. If correctness is not required, updates to the data file can continue while result sets are built and used.

Keeping the Result Set Correct

If the program using a result set requires that the result set be correct, it must ensure that the data file is not updated from the time the result set is begun through the time that its use has been completed. How this is done depends on whether the result set is created and used within a single instance of open, within a single job, or across jobs.

When the result set is created and used within a single instance of open, updates can be prevented by calling LOCKF before beginning to create the result set. The LOCKF call should request a Preserve_Content file lock to allow the nested file to be read, but not updated. The lock should be held until all use of the result set has been completed.

When the result set is created and used within a single job, data file updates can be prevented by attaching the data file so that the specified access modes and share modes do not include append or shorten access modes. This prevents updating of the file while it is attached to the job.

When the result set is to be created and used across jobs, data file updates can be prevented by creating a permit for the data file that applies to all users (a public permit) that omits the append and shorten permissions. Also, to be used across jobs, the result set file must be a file in a permanent file catalog.

Recovering from Result Set Read Errors

If the file could be updated between the time the result set is built and the time it is used, the program should check for possible errors returned by RSGETN calls. Calls to read a record could fail because a primary-key value is locked or because the record for the primary-key value has been deleted. Because the errors are nonfatal, they do not terminate the program so the sequence of reads can continue.

To recover from a lock conflict while the file is shared, the program could re-try reading the record. The retry method used depends on the result_set_not parameter value.

To retry a get call (result_set_not is 'NO'), call RSSKIP to move the result set back one value and then re-try the read. Or, the program could call RSINFO to get the previous_key value. The program could then call GET using the previous_key value. With $GET_AND_LOCK and $WAIT_FOR_LOCK set, the GET call waits for the record until it can read it.

To retry a get_not call (result_set_not is 'YES'), no repositioning is necessary. The program can re-try the read by calling RSGETN again.

Result Set Files

Result sets reside in sequential files, called result set files. The RSOPEN call specifies the result set file. If the specified file does not exist, RSOPEN creates the file.

The first RSOPEN call for a result set stores file attribute values that identify the file as a result set file. RSOPEN calls for an existing result set check that the specified file is a result set file.

NOTE

To preserve the integrity of the result set, do not perform any operations except result set operations on result set files.

Combining Result Sets

The RSBUILD and RSCOMB calls can combine result sets.

- An RSCOMB call combines two existing result sets.
- An RSBUILD call combines an existing result set (called its source result set) with a range of key values from the data file.

Combination Operations

Result sets can be combined by one of these three operations (as specified by the logical_operation parameter on the call):

Logical AND (0)

The combined result set is the intersection of the result sets. It contains only those key values that belong to both of the sets.

Logical OR (1)

The combined result set is the union of the result sets. It contains all key values from both result sets.

Logical XOR (2)

The combined result set is the union of the result sets without the intersection of the result sets. It contains all key values from each of the result sets that do not belong also to the other result set.

Placement of the Combined Result Set

On the RSBUILD and RSCOMB calls, an existing result set to be combined is specified as a source result set. The combined result set can overwrite the source result set or be written to another result set file called the target result set.

The placement of the combined result set is determined by the value of the new_result_placement parameter on the call. The parameter can specify one of these values:

Result in Source (0)

The combined result set overwrites the source result set. For RSCOMB, the result set overwritten is always the second of the source result sets specified. Use this value only when the source result set is no longer needed. It can also be used by an RSBUILD call when the source and target result sets are the same. (The source and target result sets cannot be the same for an RSCOMB call.)

Result in Target (1)

The combined result set is written to the target result set. Use this value when the source_result_set is to be saved for later use. It is also used on the initial RSBUILD call for a new result set.

Result in Fastest Place (2)

The placement of the combined result set is chosen to provide the fastest performance. The location chosen is returned in the variable specified by the actual_result_set_ placement parameter on the call. Use this value when the source result set is no longer needed and the source and target result sets differ.

Adding Or Deleting Key Values

The RSPUT and RSDLTE calls add or delete one primary-key value in the result set. These calls are for specifying isolated primary-key values, instead of the range of key values specified by an RSBUILD call.

For Better Performance

In cases where several scattered primary-key values are to be added or deleted in a result set and the result set is large, calls to directly add or delete individual key values are not the most efficient method of producing the target result set.

It is more efficient to form a temporary result set containing the individual primary-key values and combine the temporary result set with the source result set to form the target result set.

If possible, put the primary-key values into the result set in ascending order. This builds the result set more efficiently.

To add several individual primary-key values to a large result set:

1. Call RSPUT to put each primary-key value to be added into a temporary result set.
2. Combine the result sets using an logical OR (1) operation.

To delete several individual primary-key values from a large result set:

1. Call RSPUT to put each primary-key value to be deleted into a temporary result set.
2. Call RSCOMB specifying the original result set as the first_source_result_set and the temporary result set as the second_source_result_set. Combine the result sets using an logical XOR (2) operation; specify result in source (0) to overwrite the temporary result set.
3. Combine the temporary result set created by step 2 with the original result set using an logical AND (0) operation. (This step is required only when one or more of the records to be deleted may not have been in the original result set.)

Keyed-File Interface Calls

These are the keyed-file interface calls.

Call	Purpose
CLOSEM	Closes an open file
DLTE	Deletes a record
FILEDA	Creates a FIT for a direct access file
FILEIS	Creates a FIT for an indexed-sequential file
FLUSHM	Copies the file in memory to disk
GET	Reads a record by its key value
GETN	Reads the next record in sequential order
IFETCH	Fetches a FIT value
KEYLIST	Fetches primary-key values from an alternate index
KLCOUNT	Fetches the number of primary-key values within a range in the alternate index
KLSPACE	Fetches the number of alternate-index blocks that contain the specified alternate-key value range
LOCKF	Locks a file
LOCKK	Locks a primary-key value
OPENM	Opens a keyed file
PUT	Writes a record
PUTREP	Writes or replaces a record
REPLC	Replaces a record
REWND	Positions the file at the lowest key value
RMKDEF	Creates an alternate key
RSOPEN	Opens a result set
RSCLOSE	Closes an open result set
RSCLEAR	Clears a result set
RSBUILD	Builds a result set
RSCOMB	Combines two result sets
RSDLTE	Deletes a key value from a result set
RSPUT	Adds a key value to a result set
RSGETN	Uses the result set to get a record from the data file
RSINFO	Returns information about the result set
RSREWND	Positions the result set at its beginning
RSSKIP	Positions the result set forward or backward
RSSTART	Positions the result set at a key value

Call	Purpose
SKIP	Repositions the file forward or backward a number of records
STARTM	Positions the file by a key value
STOREF	Stores a value in the FIT
UNLOCKF	Clears a file lock
UNLOCKK	Clears either a single primary-key value lock or all locks for the instance of open

If you are migrating a CYBER 170 FORTRAN program that uses GETNR or SEEKF calls, see appendix C for more information.

Keyed-File Interface Calls Introduction

Each keyed-file interface call description lists the parameters for the call (with the parameter position in parentheses). The parameters must be specified in the indicated order.

Standard FORTRAN requires that all parameters be explicitly specified on a call. However, the keyed-file interface allows you to omit parameters whose values have been specified on previous calls. (The FIT pointer must always be specified.)

NOTE

You cannot omit any parameter values on KEYLIST, KLCOUNT, KLSPACE, and STOREF calls.

To omit a parameter between two specified parameters, specify a zero (0) in the parameter position. (Thus, to actually specify zero as a parameter value, you must store zero in a variable and specify the variable name on the call.)

Except for KEYLIST, KLCOUNT, KLSPACE, and STOREF calls, a zero value on a call causes the corresponding FIT value to be used. If the corresponding value in the FIT is also zero, the default parameter value is used.

Unless indicated otherwise in the call description, a parameter value specified on a call is stored in the FIT so that it becomes the default value for subsequent calls.

No type checking is performed on the values passed by a call. Passing an improper value could result in an internal routine detecting a computational fault such as arithmetic overflow. To find the line that caused the error, use Debug to trace back the call chain.

Each call description includes a list of values that could be returned to the $ERROR_ STATUS FIT value by the call. For information on decyphering the $ERROR_STATUS value, see the $ERROR_STATUS description later in this chapter.

CLOSEM Call

Purpose Closes an open keyed file.

Format **CALL CLOSEM (fit, cf)**

Parameters **(1) fit**

Variable containing the FIT pointer returned by the call that created the FIT.

(2) cf

Close flag handling:

'R'

Rewind the file (default).

'N'

Do not rewind or detach the file.

'U' or 'RET'

Detach the file. (The file is no longer accessible from the $LOCAL catalog.)

(No default value is stored in the FIT for this parameter.)

Remarks

- When a program finishes processing a keyed file, it should immediately call CLOSEM to close the file. Close processing copies any data or index blocks in memory to the mass storage file, updates internal tables, and writes statistics to the $ERRORS file (if requested by the $MESSAGE_CONTROL value). It also clears all locks for the instance of open.
- An attempt to close a file that is not open returns a nonfatal error ($ERROR_STATUS value AA2500).
- All files are closed at task termination. This is true whether the task terminates normally or abnormally.
- A CLOSEM call does not discard the FIT. The same FIT pointer variable can be specified on a subsequent OPENM call to open the same file again.

Examples This call closes and detaches a keyed file, preventing its further use in the program. The IFETCH call checks that the CLOSEM call completed successfully.

```
CALL CLOSEM (fit, 'U')
CALL IFETCH ('$ERROR_STATUS', status)
IF (status .NE. 0) CALL errept
```

DLTE Call

Purpose Removes a record from a keyed file.

Format **CALL DLTE (fit, ka, kp, 0, ex)**

Parameters **(1) fit**

Variable containing the FIT pointer returned by the FILEIS or FILEDA call that created the FIT.

(2) ka

Location of the primary-key value of the record to be deleted.

NOTE

The key area should be in a common block. If it is not, your program could execute incorrectly after being compiled with high optimization.

---

(3) kp

For FORTRAN 5 compatibility. New programs should set this parameter to zero.

(4) 0

Reserved position for unused parameter.

(5) ex

Name of the error-exit procedure.

$ERROR_STATUS	Nonfatal:	AA2000 -- key_not_found
		AA2085 -- key_not_already_locked
		AA2605 -- key_required
		AA2615 -- non_embedded_key_not_given
		AA2650 -- not_enough_permission
	Fatal:	AA3250 -- file_is_ruined
		AA3430 -- file_at_file_limit

Remarks

- A DLTE call requires append, shorten, and modify access to the file. Otherwise, DLTE returns a nonfatal error ($ERROR_STATUS value AA2650).

- If the file could be shared (more than one instance of open could be changing the file at the same time), a record can be deleted only if the instance of open has a Preserve_Access_and_Content or Exclusive_Access lock on the primary-key value. An invalid attempt returns error status AA2085.

  A task can lock a primary-key value by calling LOCKK, GET, or GETN. To read about locks, see the earlier subsection Keyed-File Sharing.

- You cannot delete a record by specifying its alternate-key value. You must specify its primary-key value. The key value specified on a DLTE call is processed as a primary-key value even if an alternate key is currently selected. A DLTE call deletes the primary-key value from all alternate indexes that reference it.

- DLTE searches for the primary-key value only in the nested file currently selected.

- If DLTE cannot find a record with the specified primary-key value, it returns a nonfatal error ($ERROR_STATUS value AA2000).

- A DLTE call does not change the file position or change the currently selected key or nested file.

For Better Performance

When deleting a sequence of records, it is most efficient to delete the records in order from the highest primary-key value to the lowest primary-key value. By working backwards, you can avoid relocation of records to be subsequently deleted.

- 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.

Examples

The DLTE call deletes the record with primary-key value ABCD. The IFETCH call checks that the DLTE call completed succesfully.

```
key = 'ABCD'
CALL DLTE (fit, key)
CALL IFETCH (fit,'$ERROR_STATUS',status)
IF (status .NE. 0) CALL rerreprt
```

FILEDA Call

Purpose

Creates a file information table (FIT) for a direct access file and, optionally, initializes FIT values.

Format

CALL FILEDA (fit, keyword, value, ..., keyword, value)

Parameters

(1) fit

Integer variable in which the FIT pointer is returned.

(2) keyword

Character expression specifying a FIT keyword (must be followed by an allowable value for the attribute). The keyword must be a character expression (for example, '$KEY_LENGTH').

(3) value

FIT value to be stored for the preceding keyword. The applicable values are listed in the individual keyword description.

Remarks

- The FILEDA call must be the first call for a direct access file because it creates the FIT for the file and sets the $FILE_ORGANIZATION value to DIRECT_ACCESS.

  All other calls for the file must specify the FIT pointer variable returned by the FILEDA call.

- Except for the $FILE_ORGANIZATION value, FILEDA call processing is the same as FILEIS call processing.

Examples

This call creates a FIT for an existing direct access file named MY_DA_FILE. It stores two FIT values: the local file name and the access modes.

```
  CALL FILEDA( fitptr, '$LFN', 'my_da_file',
 +          '$ACCESS_MODE',   'READ,MODIFY')
```

This call creates a FIT for a new direct access file, specifying the minimum required attributes:

```
  CALL FILEDA( fitptr, '$LFN', 'my_da_file',
 +          '$INITIAL_HOME_BLOCK_COUNT', 23,
 +          '$KEY_LENGTH',               15,
 +          '$MAXIMUM_RECORD_LENGTH',    80,
 +          '$MINIMUM_RECORD_LENGTH',    15)
```

FILEIS Call

Purpose Creates a file information table (FIT) for an indexed-sequential file and, optionally, initializes FIT values.

Format **CALL FILEIS (fit,keyword,value,...,keyword,value)**

Parameters **(1) fit**

Integer variable in which the FIT pointer is returned.

(2) keyword

Character expression specifying a FIT keyword (must be followed by an allowable value for the attribute). The keyword must be a character expression (for example, '$KEY_LENGTH').

(3) value

FIT value to be stored for the preceding keyword. The applicable values are listed in the individual keyword description.

Remarks

- The FILEIS call must be the first call for an indexed-sequential file because it creates the FIT for the file and initializes the $FILE_ORGANIZATION value to INDEXED_SEQUENTIAL. All subsequent keyed-file interface calls must specify the variable containing the FIT pointer returned by the FILEIS call.
- The FILEIS call can specify any number of keyword,value pairs in any order. You can change FIT values specified by the FILEIS call using STOREF calls.
- FILEIS returns a nonfatal error ($ERROR_STATUS value AA2510) when it does not recognize a specified keyword. It also returns a nonfatal error ($ERROR_STATUS value AA2505) if a specified value is outside of the range applicable for the parameter.
- The FILEIS call associates the FIT with a local file name using the $LOCAL_FILE_NAME keyword. The old/new (ON) value indicates whether the file is a new or existing file.
- File attribute values specified by SET_FILE_ATTRIBUTE commands before program execution override corresponding attribute values specified by FILEIS calls.
- Attribute values in the FIT are checked for validity and consistency when the file is opened.

Examples This call creates a FIT for an existing indexed-sequential file named MY_IS_FILE. It stores two FIT values: the local file name and the access modes.

```
CALL FILEIS(fitptr, '$LFN', 'my_is_file','$ACCESS_MODE','READ,MODIFY')
```

This call creates a FIT for a new indexed-sequential file, specifying the minimum required attributes:

```
 CALL FILEIS( fitptr, '$LFN', 'my_new_is_file',
+           '$KEY_LENGTH',                15,
+           '$MAXIMUM_RECORD_LENGTH',     80,
+           '$MINIMUM_RECORD_LENGTH',     15)
```

FLUSHM Call

Purpose Writes all modified blocks to mass storage.

Format **CALL FLUSHM (fit)**

Parameters **(1) fit**

Variable containing the FIT pointer returned by the call FILEIS or FILEDA that created the FIT.

Remarks

- A FLUSHM call requires append, shorten, or modify access to the file. Otherwise, it returns a nonfatal error ($ERROR_STATUS value AA2650).
- A FLUSHM call ensures that the disk copy of the file contains the latest changes to the file. FLUSHM does not reposition or close the file. Blocks in memory are not disturbed.
- If the $FORCED_WRITE value in the FIT is TRUE, a data or index block is copied to disk immediately after the block is changed. However, a FLUSHM call also copies all internal file tables to disk, providing a complete backup copy.

Examples The STOREF call specifies the error-exit procedure to be called if the FLUSHM call detects an error. (The program has previously declared the ERREXIT subprogram as EXTERNAL.)

```
CALL STOREF (fit,'$EEP',errexit)
CALL FLUSHM (fit)
```

GET Call

Purpose Reads a record by its key value from an open keyed file.

Format **CALL GET (fit, wsa, ka, kp, mkl, 0, ex)**

Parameters

(1) fit

Variable containing the FIT pointer returned by the FILEIS or FILEDA call that created the FIT.

(2) wsa

Working storage area (location to which the record data is copied).

NOTE

The working storage area and the key area should be in common blocks. If they are not, your program could execute incorrectly after being compiled with high optimization.

(3) ka

Location of the key value of the record to be read.

(4) kp

For CYBER 170 compatibility. New programs should set this parameter to zero.

(5) mkl

Major key length (in bytes); defaults to zero. It is reset to zero after the call.

When using a variable_length alternate key, a nonzero mkl value is required because it specifies the key-value length.

The parameter is ignored if the file is a direct access file and its primary key is currently selected.

(6) 0

Reserved position for unused parameter.

(7) ex

Error-exit procedure name.

$ERROR_STATUS	Nonfatal:	AA2000 -- key_not_found
		AA2010 -- record_longer_than_wsa
		AA2035 -- primary_key_locked
		AA2040 -- key_deadlock
		AA2045 -- key_self_deadlock
		AA2055 -- key_timeout
		AA2075 -- key_found_lock_no_wait
		AA2080 -- key_already_locked
		AA2115 -- too_many_keylocks
		AA2120 -- no_exclusive_if_read_only
		AA2615 -- non_embedded_key_not_given
		AA2620 -- wsa_not_found
		AA2640 -- major_key_too_long
		AA2650 -- not_enough_permission
		AA2715 -- no_auto_unloc_pc
		AA2805 -- key_expired_lock_exists
	Fatal:	AA3250 -- file_is_ruined
		AA3430 -- file_at_file_limit
		AA3435 -- lock_file_crowded

Remarks

- A GET call requires at least read access to the file. Otherwise, it returns a nonfatal error ($ERROR_STATUS value AA2650.) To update file statistics, it also requires modify access.
- A GET call requires that a working storage area be specified on the call or in the FIT. If no working storage area is specified, it returns a nonfatal error ($ERROR_STATUS value AA2620).
- GET searches for the specified key value in the currently selected nested file only.
- GET uses the primary or alternate key specified by the $KEY_NAME value in the FIT. The $KEY_NAME value is initially set to the primary key ($PRIMARY_KEY).
- When the primary key is selected, a ka value must be specified on the call or in the FIT.
- When an alternate key is selected and the ka values on the call and in the FIT are both zero, GET assumes that the alternate key value is in the working storage area at the position of the alternate key in the record.

  For example, if the alternate key is bytes 5 through 10 of the record, GET uses the contents of bytes 5 through 10 of the working storage area as the alternate-key value.
- The meaning of the mkl value depends on whether the selected key is fixed-length or variable-length.
  - For a fixed-length key, a nonzero mkl value specifies that GET is to search for the key using a major key. This means that, starting from the left of the key value, only mkl bytes of the key values are compared.

  - For a variable-length key, the mkl value specifies the key-value length. The key value is compared with the full key values stored in the index.

  A major key length value specified on a call is not stored in the FIT. A $MAJOR_KEY_LENGTH value specified in the FIT is cleared after it is used, so the program must specify a major key length value for each call that is to use a major key or a variable-length key value. (Major-key use is valid only while either the primary key of a direct access file or any alternate key is selected.) For more information, see the $MAJOR_KEY_LENGTH FIT value description.

- If an alternate key has been selected and the key is a concatenated key, the values for the pieces of the key must be assembled in the key area or the working storage area.

  In the key area, the pieces must be concatenated in the order defined for the alternate key.

  In the working storage area, the pieces must be stored in their fields in the record.

  For example, suppose the first piece of the alternate key is the third byte of the record and the second piece of the alternate key is the first byte of the record. To get the record whose first byte is an A and whose third byte is an *, either:

  - store A in the first byte of the working storage area and * in the third byte, or

  - store *A in the key area.

- GET searches for the first key value that satisfies the relation specified by the $KEY_RELATION value in the FIT.

  - If the relation is EQUAL_KEY and an equal key value does not exist in the file, GET returns a nonfatal error ($ERROR_STATUS value AA2000). The file is left positioned to read the next record (the record that would follow the specified record if it existed).

  - If the $KEY_RELATION value is GREATER_OR_EQUAL_KEY or GREATER_KEY and no key value in the file satisfies the relation, the data-exit (DX) procedure is called, if one is specified in the FIT. The file is left positioned at the end of information.

- If the $GET_AND_LOCK value in the FIT is -1 (YES), the GET call requests a lock on the primary-key value of the record to be read. The lock request uses the $AUTOMATIC_UNLOCK, $LOCK_INTENT, and $WAIT_FOR_LOCK values in the FIT. To read about locks, see Keyed-File Sharing earlier in this section.

  When an alternate key is selected, the GET call requests a lock on the first primary-key value in the key list only.

  If the GET call fails for any reason, it terminates without a lock on the primary-key value.

- The GET call reads data from the record until it reaches the end of the record or it has read the number of bytes specified as the working storage length in the FIT. (GET does not overwrite space following the working storage area with excess data.)

If the record being read is longer than the working storage length, GET returns a nonfatal error ($ERROR_STATUS value AA2010).

- A successful GET call sets the record length value in the FIT to the actual length of the record. The record length value is not defined for an unsuccessful GET call.

- File positioning by a GET call differs depending on the file organization and the selected key:

  For a direct access file with its primary key selected, the following statements are true:

  - GET does not change the file position used by GETN calls.

  - The only file position GET returns is end-of-record (16).

  - The only calls that can reposition the file are REWND and GETN. (STARTM is not valid.)

  - The major_key_length (mlk) and $KEY_RELATION values are not used.

  A GET call for a direct access file with an alternate key selected is processed the same as a call to an indexed-sequential file with an alternate key selected.

- For an alternate key or an indexed-sequential file, the following statements are true:

  - At completion of a successful GET call, the file is positioned to read the record with the next highest key value. (The file position returned can be end-of-record [16] or, for an alternate key, end-of-key-list [8]).

  - An unsuccessful GET call returns a file position of 64 (end-of-information) in these cases:

    - The specified $KEY_RELATION was GREATER_THAN_OR_EQUAL and the key value was greater than all key values in the file.

    - The specified $KEY_RELATION was GREATER_THAN and the key value is the greatest in the file.

- A GET call that requests an unavailable lock leaves the file positioned to read the requested record.

- The program should call IFETCH to return the file position after a successful GET call.

  When the file position value returned is 64 (end-of-information), the file is positioned at the end of the file and no GETN calls should be issued before file repositioning.

- GET can return the primary-key value of a record it found using an alternate-key value. If the $PRIMARY_KEY_ADDRESS value in the FIT is nonzero, GET returns the primary-key value in the $PRIMARY_KEY_ADDRESS location.

Examples This sequence of calls reads a record by major key value.

```
C Gets the first record whose key value begins with AB.
 key1 = 'ABCD'
 CALL GET (fit, record1, key1, 0, 2, 0, errexit)

C Gets the current file position and calls subroutine NOREC
C if no key value in the file begins with AB.  (The file
C would be left positioned at its end-of-information.)

 IF (IFETCH (fit, '$FILE_POSITION') .EQ. 64) THEN
   CALL norec
 ELSE

C Fetches the record length of record read and passes the
C record and its length to subroutine PROCDTA.
   CALL IFETCH (fit, 'RL', recleng)
   CALL procdta (record1, recleng)
 ENDIF
```

GETN Call

Purpose Reads the next record at the current file position.

Format **CALL GETN (fit,wsa,ka,ex)**

Parameters **(1) fit**

Variable containing the FIT pointer returned by the call that created the FIT.

(2) wsa

Working storage area (location to which the record data is copied).

NOTE

The working storage area and the key area should be in common blocks. If they are not, your program could execute incorrectly after being compiled with high optimization.

(3) ka

Variable in which GETN returns the key value of the record.

For a variable-length alternate key, the key value is written to the variable followed by padding characters up to the maximum key length. The padding character used is the lowest character in the key-delimiter set.

For example, if the variable is 80 bytes long, the key value is 12 bytes, and the maximum key length is 31 bytes, the call first writes the 12-byte key value and then 19 padding characters. The GETN call does not write to the last 49 bytes.

(4) ex

Error-exit procedure name.

$ERROR_STATUS	Nonfatal:	AA2010 -- record_longer_than_wsa
		AA2035 -- primary_key_locked
		AA2040 -- key_deadlock
		AA2045 -- key_self_deadlock
		AA2055 -- key_timeout
		AA2075 -- key_found_lock_no_wait
		AA2080 -- key_already_locked
		AA2115 -- too_many_keylocks
		AA2120 -- no_exclusive_if_read_only
		AA2615 -- non_embedded_key_not_given
		AA2620 -- wsa_not_found
		AA2635 -- cant_position_beyond_bound
		AA2640 -- major_key_too_long
		AA2650 -- not_enough_permission
		AA2665 -- cant_da_getn_after_put
		AA2715 -- no_auto_unloc_pc
		AA2790 -- da_getn_lost_file_lock
		AA2805 -- key_expired_lock_exists
		AA2880 -- cant_da_getn_if_shared

Remarks

- A GETN call requires at least read access to the file. (Otherwise, it returns a nonfatal error, $ERROR_STATUS value AA2650). To update file statistics, it also requires modify access.

- A GETN call requires that a working storage area be specified on the call or in the FIT. If no working storage area is specified, it returns a nonfatal error ($ERROR_STATUS value AA2620).

- If the $GET_AND_LOCK value in the FIT is -1 (YES), GETN requests a lock on the primary-key value of the record to be read. The lock request uses the $AUTOMATIC_UNLOCK, $LOCK_INTENT, and $WAIT_FOR_LOCK values in the FIT. To read about locks, see Keyed-File Sharing earlier in this section.

  If the GETN call fails for any reason, it terminates without a lock on the primary-key value.

- The GETN call reads data from the record until it reaches the end of the record or it has read the number of bytes specified as the working-storage length in the FIT. (GETN cannot copy more data than the working storage area length.)

  If the record being read is longer than the working-storage length, GETN returns a nonfatal error ($ERROR_STATUS value AA2010).

- A successful GETN call sets the record-length value in the FIT to the actual length of the record. The record-length value is not defined for an unsuccessful GETN call.

- When an alternate key is selected, GETN calls return records in the key-value order provided by the alternate index.

  When the primary key of an indexed-sequential file is selected, GETN returns records in the key-value order provided by the primary index.

  However, no index exists for the primary key of a direct access file so GETN does not return records in key-value order. It returns records in physical order by their location in the file.

  A GETN call that requests an unavailable lock leaves the file positioned to read the requested record.

- When a GETN call reads a record from the file, it returns a $FILE_POSITION value of 16 (or 8 if an alternate key is selected).

  After the GETN call that reads the last record in the file, the next GETN call returns a $FILE_POSITION of 64 (end-of-information). It returns an $ERROR_STATUS of 0 (no error), but no data or key values.

  A GETN call issued after a $FILE_POSITION value of 64 is returned, and before the file is repositioned, is an attempt to read beyond the end-of-information. The call returns a nonfatal error ($ERROR_STATUS value AA2635).

- The key value returned to the ka location is the value of the currently selected key. If the selected key is an alternate key, the value returned is the alternate-key value.

  The length of the value returned is the key_length specified when the key was created. A variable-length alternate-key value is padded to its right with delimiter characters up to the maximum length for the key. (The padding character is the lowest character in the key-delimiter set.)

- GETN can also return the primary-key value when an alternate key is selected. If the $PRIMARY_KEY_ADDRESS value in the FIT is nonzero, GETN returns the primary-key value in the $PRIMARY_KEY_ADDRESS location.

Examples

This sequence of calls reads all records whose alternate key value is ABC into a very long character variable named WSA.

```
      CALL STOREF (fit, '$KEY_NAME', 'ALT1')
      key = 'ABC'
      n = 1
      CALL GET (fit, wsa(n), key, 0, 0, errexit)
      IF (IFETCH(fit, '$FILE_POSITION') .EQ. 8) THEN
        CONTINUE
      ELSEIF (IFETCH(fit, '$FILE_POSITION') .EQ. 16) THEN
 10     n = n + IFETCH(fit,'$RECORD_LENGTH')
        CALL GETN (fit, wsa(n), 0, 0)
        IF (IFETCH(fit, '$FILE_POSITION') .EQ. 16) GO TO 10
      ELSE
        CALL nodata
      ENDIF
      n = n + IFETCH(fit, 'RL')
      CALL procdta (wsa, n)
```

IFETCH Call

Purpose Retrieves a FIT field value.

NOTE

IFETCH can be called as a function or as a subroutine.

Format **IFETCH (fit, keyword)**

CALL IFETCH (fit, keyword, variable)

Parameters **(1) fit**

Variable containing the FIT pointer returned by the FILEIS or FILEDA call that created the FIT.

(2) keyword

Character expression specifying the FIT value to be fetched (such as, '$FILE_POSITION').

The keyword can be specified using uppercase and/or lowercase letters. (The keywords are listed in the FIT value descriptions later in this section.)

(3) variable

Variable to receive the FIT value.

Remarks

- Before a FIT is used to open a file, the only values that IFETCH can fetch from the FIT are those that have been stored in the FIT by the FILEIS or FILEDA call that created the FIT or by a STOREF call.

- While the file is open, IFETCH can fetch any value from the FIT. However, after the file is closed, IFETCH can only fetch certain values. The following is a list of the values that it can fetch.

$AUTOMATIC_UNLOCK	$LOCAL_FILE_NAME
DX (data exit routine)	$LOCK_INTENT
$ERROR_STATUS	$MAJOR_KEY_LENGTH
$ERROR_EXIT_PROCEDURE	OC (opened/closed flag)
FNF (fatal/nonfatal flag)	ON (old/new flag)
$FILE_ORGANIZATION	$PRIMARY_KEY_ADDRESS
$FILE_POSITION	RKW (relative key word)
$GET_AND_LOCK	RL (record length)
$KEY_ADDRESS	$WAIT_FOR_LOCK
$KEY_POSITION	$WORKING_STORAGE_ADDRESS
$KEY_RELATION	$WORKING_STORAGE_LENGTH
$LAST_OPERATION	

- IFETCH always returns an 8-byte value. In most cases, the value is an integer number, although it may be a boolean value or the first 8 characters of a name. The value returned for each keyword is described later in the individual description of the FIT value.

- IFETCH returns a boolean FALSE (or NO) value as the integer 0; it returns a boolean TRUE (or YES) value as the integer -1.

- IFETCH returns a name value as the first 8 characters of the name, left-justified, with blank padding. (It does not return the full 31 characters of the SCL name.) The name is returned using uppercase letters, even if the program specified the name using lowercase letters.

- Fetching an address, that is, fetching the $KEY_ADDRESS, $WORKING_STORAGE_ADDRESS, or $PRIMARY_KEY_ADDRESS value, is not recommended because the program cannot use the value returned.

Examples

This call fetches the error status value. If no error-exit procedure has been specified, the program should check for a nonzero error status value after each keyed-file interface call.

```
IF (IFETCH (fit,'$ERROR_STATUS') .NE. 0)  CALL errprog
```

KEYLIST Call

Purpose

Fetches primary-key values from the alternate index, beginning at the current position.

NOTE

You must specify values for all KEYLIST parameters. KEYLIST does not use FIT values as default values.

Format

CALL KEYLIST (fit, high_key, major_high_key, high_key_relation, working_storage_area, working_storage_length, end_of_primary_key_list, transferred_byte_count, transferred_key_count, filpos, condition_code)

Parameters

(1) fit

Name of the variable containing the file information table (FIT) pointer.

(2) high_key

Alternate-key value at which the range ends. The value must be valid for the key type (integer for an integer key, character for a collated or uncollated key).

(3) major_high_key

For a fixed-length key, a nonzero value indicates that the high end of the range is to be found by a major-key search. The specified value is the number of leftmost bytes of the high_key value to be used as the major key. A zero value indicates that the full high_key value is to be used.

For a variable-length alternate key, a nonzero value is required because it specifies the length of the key value.

(4) high_key_relation

Indicates when KEYLIST is to stop fetching key values.

'GREATER_KEY' or 'GK' or 'GT'

Stop at the lowest alternate-key value greater than the high_key value.

'EQUAL_KEY' or 'EK' or 'EQ' or 'GREATER_OR_EQUAL_KEY' or 'GOEK' or 'GE'

Stop at the lowest alternate-key value greater than or equal to the high_key value.

'HIGHEST_KEY' or 'HK'

Stop at the end of the alternate index. (The high_key and major_high_key values are ignored when 'HIGHEST_KEY' is specified.)

(5) working_storage_area

Variable in which the primary-key values are returned.

(6) working_storage_length

Number of bytes in the working storage area.

(7) end_of_primary_key_list

Integer variable in which KEYLIST returns a value indicating whether the working storage area was long enough to contain all values in the requested range.

0	KEYLIST could not return all values in the requested range.
1	KEYLIST returned all values in the requested range.

(8) transferred_byte_count

Integer variable which receives the total length, in bytes, of the primary-key values KEYLIST returned in the working storage area.

(9) transferred_key_count

Integer variable which receives the number of primary-key values KEYLIST fetched.

(10) filpos

Integer variable in which the file position at completion of the KEYLIST call is returned.

Value	Meaning
8	The file is positioned at the end of a key list (positioned to fetch the first value in the next list).
16	The file is positioned at the end of a record, but not at the end of a key list (positioned to fetch the next value in the same key list).
64	The file is positioned at the end of the alternate index. (It cannot fetch any more values at this position.)

(11) condition_code

Integer variable in which the error status value is returned. A zero value returned indicates successful completion.

Some nonfatal-error values that could be returned are:

AA2650

You must have at least read permission to the file.

AA2755

The high end of the range must be above the current position of the file.

AA2760

The KEYLIST call is valid only if an alternate key is currently selected.

For information on decyphering the condition code, see the ERROR_ STATUS description later in this section.

Remarks

- The program must select an alternate key before issuing a KEYLIST call.

- The high_key parameter value specifies the upper bound of the range of keys to be returned. The high_key_relation parameter indicates whether the primary-key values for the high_key value itself are returned.

  For example, suppose the high_key value is SMITH.

  - If you specify 'GREATER_KEY' as the high_key_relation value, KEYLIST returns the primary-key values for SMITH.

  - If you specify 'EQUAL_KEY' as the high_key_relation, KEYLIST does not return the primary-key values for SMITH. (It stops fetching values at the SMITH alternate-key value.)

- A major key consists of the leftmost bytes of a key. For a fixed-length key, a nonzero major_high_key parameter specifies the number of bytes of the high_key value KEYLIST to use as a major key. A major key search compares only the leftmost bytes of the key values on the call and in the index.

  For example, suppose the high_key value is ABCDEF and the major_high_key parameter value is 2. The major key used is AB. KEYLIST returns primary-key values until it finds an alternate-key value beginning with the characters AB or higher. (Whether it returns the primary-key values for the AB value depends on the high_key_relation parameter value.)

  (Major_key use is invalid when the primary key of a direct access file is selected.)

- The KEYLIST call could return the same primary-key value more than once if the primary-key value is associated with more than one alternate-key value. This is possible if the repeating-groups attribute is defined for the alternate key.

- KEYLIST returns primary-key values until it reaches the end of the specified range or until it cannot fit another value into the working storage area. By checking the end_of_primary_key_list value, the program can determine if all requested values were returned and, if not, call KEYLIST again to fetch the rest of the values.

- KEYLIST repositions the file as it fetches key values. At completion of the call, the file is positioned at the end of the last key value returned and positioned to continue fetching values at that point if KEYLIST is called again.

Examples

These calls fetch all primary-key values in the alternate index. The STOREF call selects alternate key ALT_KEY_1 and positions the file at the beginning of the alternate index. The subroutine KEYPROC processes the key values fetched. The KEYLIST call is repeated until all primary-key values are fetched.

```
      CALL STOREF (fit, '$KEY_NAME', 'ALT_KEY_1')

   10 CALL KEYLIST(fit, 0, 0, 'HIGHEST_KEY', wsa, LEN(wsa),
     +             keyend, length, keycnt, filpos, ccode)

      IF (ccode .NE. 0) THEN
         CALL errprog
      ELSE
         CALL keyproc(wsa, LEN(wsa), length, keycnt)
      ENDIF

      IF (keyend .EQ. 0) GO TO 10
```

The STARTM call positions the alternate index at alternate-key value ABCD. The KEYLIST call then fetches the primary-key values for that alternate-key value.

```
      keyval='ABCD'
      CALL STARTM(fit, keyval)
      CALL KEYLIST(fit, keyval, 0, 'GT', bigaray, LEN(bigaray),
     +             keyend, length, keycnt, filpos, ccode)
      IF (ccode .NE. 0) CALL errprog
```

KLCOUNT Call

Purpose

Counts the number of primary-key values associated with the specified range of alternate-key values in the alternate index.

NOTE

You must specify values for all KLCOUNT parameters. KLCOUNT does not use FIT values.

Format

CALL KLCOUNT (fit, low_key, major_low_key, low_key_relation, high_key, major_high_key, high_key_relation, list_count_limit, list_count, condition_code)

Parameters

(1) fit

Name of the variable containing the file information table (FIT) pointer.

(2) low_key

Alternate-key value at which the range begins. The value must be valid for the key type (integer·for an integer key, characters for a collated or uncollated key).

(3) major_low_key

For a fixed-length key, a nonzero value indicates that the low end of the range is to be found by a major_key search. The specified value is the number of leftmost bytes of the low_key value to be used as the major key. A zero value indicates that the full low_key value is to be used.

For a variable-length alternate key, a nonzero value is required because it specifies the length of the key value.

(4) low_key_relation

Indicates where KLCOUNT is to start counting primary-key values.

'GREATER_KEY' or 'GK' or 'GT'

Start at the lowest alternate-key value greater than the low_key value.

'EQUAL_KEY' or 'EK' or 'EQ' or 'GREATER_OR_EQUAL_KEY' or 'GOEK' or 'GE'

Start at the lowest alternate-key greater than or equal to the low-key value.

'LOWEST_KEY' or 'LK'

Start counting at the beginning of the alternate index. (The low_key and major_low_key values are ignored when 'LOWEST_KEY' is specified.)

(5) high_key

Alternate-key value at which the range ends. The value must be valid for the key type (integer for an integer key, character for a collated or uncollated key).

(6) major_high_key

For a fixed-length key, a nonzero value indicates that the high end of the range is to be found by a major-key search. The specified value is the number of leftmost bytes of the high_key value to be used as the major key. A zero value indicates that the full high_key value is to be used.

For a variable-length alternate key, a nonzero value is required because it specifies the length of the key value.

(7) high_key_relation

Indicates when KLCOUNT is to stop counting primary-key values.

'GREATER_KEY' or 'GK' or 'GT'

Stop at the lowest alternate-key value greater than the high-key value.

'EQUAL_KEY' or 'EK' or 'EQ' or 'GREATER_OR_EQUAL_KEY' or 'GOEK' or 'GE'

Stop at the lowest alternate-key value greater than or equal to the high-key value.

'HIGHEST_KEY' or 'HK'

Stop at the end of the alternate index. (The high_key and major_high_key values are ignored when 'HIGHEST_KEY' is specified.)

(8) list_count_limit

Maximum number of primary-key values counted. If you specify zero for the parameter, no limit is set.

(9) list_count

Integer variable in which the primary-key value count is returned.

(10) condition_code

Integer variable in which the error status value is returned. A zero value returned indicates successful completion.

To determine the meaning of a nonzero condition code, see the Diagnostics Messages for NOS/VE manual.

Some of the nonfatal-error condition codes that could be returned are:

AA2650

You must have at least read permission to the file.

AA2750

The high end of the range must be above the low end.

AA2760

The KLCOUNT call is valid only if an alternate key is currently selected.

For information on decyphering the condition_code, see the $ERROR_STATUS description later in this section.

Remarks

- The program must select an alternate key before issuing a KLCOUNT call.
- The low_key and high_key parameter values specify the lower and upper bounds, respectively, of the range to be counted.
- The low_key_relation and high_key_relation parameters indicate whether the primary-key values for the low_key and high_key values, respectively, are included in the count.

  For example, suppose the low_key value is JONES and the high_key value is SMITH.

  - If you specify 'GREATER_KEY' as the low_key_relation value, KLCOUNT does not count the primary-key values for JONES.
  - If you specify 'EQUAL_KEY' as the low_key_relation value, KLCOUNT counts the primary-key values for JONES.
  - If you specify 'GREATER_KEY' as the high_key_relation value, KLCOUNT counts the primary-key values for SMITH.
  - If you specify 'EQUAL_KEY' as the low_key_relation value, KLCOUNT does not count the primary-key values for SMITH.
- A major key consists of the leftmost bytes of a key. For a fixed-length key, a nonzero major_high_key or the major_low_key parameter specifies the number of bytes of the high_key or low_key value, respectively, that KLCOUNT is to use as a major key. A major key serach compares only the leftmost bytes of the key values on the call and in the index.

  For example, suppose the low_key value is ABCDEF. If the major_low_key parameter value is 2, the major key used is AB. KLCOUNT would then search for the lowest alternate-key value whose first two characters are greater than or equal to AB.
- The KLCOUNT call could count the same primary-key value more than once if the primary-key value is associated with more than one alternate-key value. This is possible if the repeating-groups attribute is defined for the alternate key.
- KLCOUNT returns the value 0 as the list count if it cannot find either the low_key or high_key values in the alternate index.

  For example, if the low_key and high_key values are both A and A is not an alternate-key value in the index, KLCOUNT returns 0 as the list count.
- The list_count_limit parameter can minimize the processing required for the call.

  For example, if you call KLCOUNT to determine whether the number of primary-key values is 0, 1, or more than 1, you should set the list_count_limit to 2.

Examples These calls return the number of primary-key values for alternate key ALT_KEY_1 in the integer variable KEYCNT. The completion code is returned in the integer variable CCODE.

```
CALL STOREF(fit, '$KEY_NAME', 'ALT_KEY_1')
CALL KLCOUNT(fit, 0, 0, 'LOWEST_KEY',
+                 0, 0, 'HIGHEST_KEY', 0, keycnt, ccode)
IF (ccode .NE. 0) CALL errprog
```

These calls return the number of primary-key values associated with alternate-key values that begin with 'C' (the major-key value).

```
CALL STOREF(fit, '$KEY_NAME', 'ALT_KEY_1')
CALL KLCOUNT(fit, 'C', 1, 'EQ', 'C', 1, 'GT', 0,
+            keycnt, ccode)
IF ccode .NE. 0 CALL errprog
```

KLSPACE Call

Purpose

Returns the number of alternate-index blocks that contain the specified range of alternate-key values.

NOTE

You must specify values for all KLSPACE parameters. KLSPACE does not use FIT values as default values.

Format

CALL KLSPACE (fit, low_key, major_low_key, low_key_relation, high_key, major_high_key, high_key_relation, block_count, block_space, condition_code)

Parameters

(1) fit

Name of the variable containing the file information table (FIT) pointer.

(2) low_key

Alternate-key value at which the range begins. The value must be valid for the key type (integer for an integer key, characters for a collated or uncollated key).

(3) major_low_key

For a fixed-length key, a nonzero value indicates that the low end of the range is to be found by a major-key search. The specified value is the number of leftmost bytes of the low_key value to be used as the major key. A zero value indicates that the full low_key value is to be used.

For a variable-length alternate key, a nonzero value is required because it specifies the length of the key value.

(4) low_key_relation

Indicates whether the low_key value is included in the range.

'GREATER_KEY' or 'GK' or 'GT'

Exclude the low_key value from the range.

'EQUAL_KEY' or 'EK' or 'EQ' or 'GREATER_OR_EQUAL_KEY' or 'GOEK' or 'GE'

Include the low_key value in the range.

'LOWEST_KEY' or 'LK'

The range starts at the beginning of the alternate index. (The low_key and major_low_key values are ignored when 'LOWEST_KEY' is specified.)

(5) high_key

Alternate-key value at which the range ends. The value must be valid for the key type (integer for an integer key, character for a collated or uncollated key).

(6) major_high_key

For a fixed-length key, a nonzero value indicates that the high end of the range is to be found by a major-key search. The specified value is the number of leftmost bytes of the high_key value to be used as the major key. A zero value indicates that the full high_key value is to be used.

For a variable-length alternate key, a nonzero value is required because it specifies the length of the key value.

(7) high_key_relation

Indicates where the range ends in relation to the highest value in the range.

'GREATER_KEY' or 'GK' or 'GT'

Include the high_key value in the range.*

'EQUAL_KEY' or 'EK' or 'EQ' or 'GREATER_OR_EQUAL_KEY' or 'GOEK' or 'GE'

Exclude the high_key value from the range.

'HIGHEST_KEY' or 'HK' or 'HIGHEST_KEY'

The range ends at the end of the alternate index. (The high_key and major_high_key values are ignored when 'HIGHEST_KEY' is specified.)

(8) block_count

Integer variable in which the block count is returned.

(9) block_space

Integer variable in which the combined length of the blocks (in bytes) is returned (the block count multiplied by the block size).

(10) condition_code

Integer variable in which the error status value is returned. A zero value returned indicates successful completion.

You can look up the meaning of any nonzero condition code in the Diagnostic Messages manual.

Some of the nonfatal-error codes that could be returned are:

AA2650

You must have at least read permission to the file.

AA2750

The high end of the range must be above the low end.

AA2760

The KLSPACE call is valid only if an alternate key is currently selected.

For information on decyphering the $ERROR_STATUS value, see the $ERROR_STATUS description later in this section.

Remarks

- An alternate key must be the currently selected key when KLSPACE is called. If the primary key is currently selected, KLSPACE returns the condition code AA2760.

- The low_key, major_low_key, low_key_relation, high_key, major_high_key, and high_key_relation parameters specify the range of alternate-key values. Their use on a KLSPACE call is the same as on a KLCOUNT call. For details, see the Remarks in the KLCOUNT call description.

- A KLSPACE call does not actually find the specified alternate-key values in the alternate index. Rather, it searches the index to determine the number of blocks at the lowest level that would contain the specified range of alternate-key values.

  (An alternate index is an indexed-sequential structure with one or more index levels. The lowest level of blocks actually contain the alternate-key values and their corresponding primary-key values.)

- KLSPACE returns a value even if the specified low_key and high_key values are not in the alternate index. It returns the number of blocks that would contain the range if the values existed in the index.

- An accurate primary-key value count (such as that returned by KLCOUNT) cannot be derived from the block count that KLSPACE returns. The block counts for ranges containing the same number of primary-key values could differ because the ranges can span blocks.

  For example, suppose a range contains only one alternate-key value. If the record for the alternate-key value spans two blocks, the block count returned is 2, not 1.

- Because a KLSPACE call is faster than a KLCOUNT call, it can be used for a quick comparison of the relative lengths of primary-key lists (see the KLSPACE Example).

- The block_length value that KLSPACE returns can be used when comparing primary-key lists for files with different block sizes. (Larger blocks require longer searches.)

Examples

Assume that a program is to find a set of records in response to this query:

> Find the Jones on Madison Avenue with more than two dependents.

Assume that Jones is a value for alternate key ALT_KEY_1 and Madison Avenue is a value for alternate key ALT_KEY_2. The number of dependents is not an alternate key so the program must read the data records to find that information.

The program could read the set of records for either Jones or Madison Avenue. To minimize the number of records read, the program first issues KLSPACE calls to compare the two primary-key value lists.

The following call sequence gets the block count values, compares them, and then stores the alternate-key name and value to be used.

```
 CALL STOREF(fit, '$KEY_NAME', 'ALT_KEY_1')

 CALL KLSPACE(fit, 'Jones', 0, 'EQ', 'Jones',
+ 0, 'GT', blkcnt1, blklen, ccode)
 IF ccode .NE. 0 CALL errprog

 CALL STOREF(fit, '$KEY_NAME', 'ALT_KEY_2')

 CALL KLSPACE(fit, 'Madison Avenue', 0, 'EQ',
+ 'Madison Avenue', 0, 'GT', blkcnt2, blklen, ccode)
 IF ccode .NE. 0 CALL errprog

 IF (blkcnt1 .GE. blkcnt2) THEN
   keyval='Madison Avenue'
 ELSE
   keyval='Jones'
   CALL STOREF(fit, '$KEY_NAME', 'ALT_KEY_1')
 END IF
```

LOCKF Call

Purpose Requests a file lock.

Format **CALL LOCKF (fit, wfl, li)**

Parameters **(1) fit**

Variable containing the FIT pointer. It specifies the instance of open to be locked.

(2) wfl

Specifies whether the task waits if the lock is not immediately available.

'YES'

Task waits until either the lock is available or a time period (default value, 60 seconds) has passed. When the time period has passed, LOCKF returns the nonfatal $ERROR_STATUS value AA2055.

'NO'

LOCKF terminates, returning nonfatal $ERROR_STATUS value AA2075, indicating that the lock is unavailable.

If 0 is specified as the wfl value on the call, the FIT value $WAIT_FOR_LOCK is used. The default $WAIT_FOR_LOCK value is YES.

(The task does not wait if a deadlock exists; a deadlock with another task returns status AA2040; a deadlock within the same task returns status AA2045.)

(3) li

Lock intent. The string can be specified using uppercase and/or lowercase letters. For more information, see Lock Intents earlier in this section.

'Exclusive_Access' or 'EA'

Exclusive_Access

'Preserve_Access_and_Content' or 'PAC'

Preserve_Access_and_Content

'Preserve_Content' or 'PC'

Preserve_Content

If the li value on the call is 0, the FIT value $LOCK_INTENT is used. Its default value is Preserve_Access_and_Content.

$ERROR_STATUS	Nonfatal:	AA2055 -- key_timeout
	Fatal:	AA3435 -- lock_file_crowded
		AA6001 -- bad_resolve_time_limit

Remarks

- The lock applies to the current nested file only (as specified by the $NESTED_FILE_NAME value).

- You can change the maximum waiting period for the lock (used if wfl is YES). The default value is 60 seconds.

  To change the waiting period, create an SCL integer variable named AAV$RESOLVE_TIME_LIMIT and assign it the waiting period value in seconds (any integer greater than 1). (The timeout period should not exceed the LOCK_EXPIRATION_TIME attribute value.)

  For example, this call executes an SCL command that sets the waiting period at 45 seconds.

  ```
     CALL SCLCMD ('create_variable, name=AAV$RESOLVE_TIME_LIMIT,
    + kind=integer, value=45')
  ```

  Be aware of the scope of the AAV$RESOLVE_TIME_LIMIT variable. The default scope is LOCAL. If the time limit change should apply to all tasks in the job, specify SCOPE=JOB on the CREATE_VARIABLE command.

- Assuming the LOCK_EXPIRATION_TIME file attribute is nonzero, the lock could expire. LOCKF returns the nonfatal $ERROR_STATUS value AA2805 if the expired lock prevents granting of the requested lock. To read about lock expiration, see Lock Expiration and Clearing earlier in this section.

- File locks cannot be automatically unlocked. To clear a single file lock, call UNLOCKF. To clear all file locks for an instance of open, call CLOSEM or UNLOCKK with the 'ALL' option.

Examples

This call requests a file lock. Its wfl and li values are supplied by the $WAIT_FOR_LOCK and $LOCK_INTENT FIT values (default values YES and Preserve_Access_Content, respectively).

```
CALL LOCKF (fit)
```

LOCKK Call

Purpose Requests a lock on a primary-key value.

Format CALL LOCKK (fit, ka, wfl, au, li)

Parameters **(1) fit**

Variable containing the FIT pointer returned by the FILEIS or FILEDA call that created the FIT.

(2) ka

Key area (location containing the primary-key value to be locked).

NOTE

The key area should be in a common block. If it is not, your program could execute incorrectly after being compiled with high optimization.

(3) wfl

Indicates whether the task waits if another task has a conflicting lock on the primary-key value and no deadlock exists.

'YES'

Task waits until either the lock is available or the wait time period (default value, 60 seconds) has passed. When the time period has passed, LOCKK returns nonfatal $ERROR_STATUS value AA2055.

'NO'

LOCKK terminates, returning nonfatal $ERROR_STATUS value AA2075, indicating that the lock is unavailable.

If 0 is specified as the wfl value on the call, the FIT value $WAIT_FOR_LOCK is used. The default $WAIT_FOR_LOCK value is YES.

(4) au

Indicates whether automatic unlock is used for this lock.

'YES'

Automatic unlock is used. (The lock is cleared when the task issues a request for another record or at completion of a write or delete request specifying the locked key value.)

'NO'

Automatic unlock is not used.

If the au value on the call is 0, the FIT value $AUTOMATIC_UNLOCK is used. Its default value is YES.

NOTE

Automatic unlock cannot be used with Preserve_Content lock intent.

(5) li

Lock intent. The string can be specified using uppercase and/or lowercase letters. For more information, see Lock Intents earlier in this section.

'Exclusive_Access' or 'EA'	Exclusive_Access
'Preserve_Access_and_Content' or 'PAC'	Preserve_Access_and_Content
'Preserve_Content' or 'PC'	Preserve_Content

If the li value on the call is 0, the FIT value $LOCK_INTENT is used. Its default value is Preserve_Access_and_Content.

$ERROR_STATUS	Nonfatal:	AA2035 -- primary_key_locked
		AA2040 -- key_deadlock
		AA2045 -- key_self_deadlock
		AA2055 -- key_timeout
		AA2075 -- key_found_lock_no_wait
		AA2080 -- key_already_locked
		AA2115 -- too_many_keylocks
		AA2120 -- no_exclusive_if_read_only
		AA2715 -- no_auto_unloc_pc
		AA2805 -- key_expired_lock_exists
	Fatal:	AA3435 -- lock_file_crowded
		AA6001 -- bad_resolve_time_limit

Remarks

- LOCKK only locks primary-key values. Even if an alternate key is currently selected, the key value in the specified key area is assumed to be a primary-key value.

- A LOCKK call can reserve a presently unused primary-key value for subsequent use by the task.

- A LOCKK call does not verify that the key value is valid, nor does it check whether the key value is already in the file. The key value is verified by a subsequent call that uses the key value.

- Assuming the LOCK_EXPIRATION_TIME file attribute is nonzero, the lock could expire. LOCKK returns nonfatal $ERROR_STATUS value AA2805 if the expired lock prevents granting of the requested lock. To read about lock expiration, see Lock Expiration and Clearing earlier in this section.

- You can change the maximum waiting period for the lock (used if wfl is YES). (The default value is 60 seconds.)

  To change the waiting period, create an SCL integer variable named AAV$RESOLVE_TIME_LIMIT and assign it the waiting period value in seconds (any positive integer).

  For example, this call executes an SCL command that sets the waiting period at 45 seconds.

```
    CALL SCLCMD ('create_variable, name=AAV$RESOLVE_TIME_LIMIT,
   +  kind=integer value=45')
```

Be aware of the scope of the AAV$RESOLVE_TIME_LIMIT variable. The default scope is LOCAL. If the time limit change should apply to all tasks in the job, specify SCOPE=JOB on the CREATE_VARIABLE command.

- LOCKK returns a nonfatal error if the requested lock could cause a deadlock. A potential deadlock can be detected only if the wfl value for the call is YES.

  If the deadlock is with another task, it returns error AA2040. If the deadlock is a self-deadlock (the requesting task already has the requested lock), it returns error AA2045.

  To clear the deadlock situation, the task should clear its locks. It can then request the locks again.

- Besides the automatic unlock, a task can unlock a key value by calling UNLOCKK or by closing the instance of open.

Examples

This call requests a lock on a key value. Its ka, wfl, au, and li values are supplied by these FIT values, respectively: $KEY_ADDRESS (no default), $WAIT_FOR_LOCK (default YES), $AUTOMATIC_UNLOCK (default YES), and $LOCK_INTENT (default Preserve_Access_and_Content).

```
CALL LOCKK(fit)
```

This call requests a lock on the key value in variable KEY1. The next call writes the record. The lock is automatically unlocked at completion of the write request.

```
CALL LOCKK(fit, key1, 'YES', 'YES', 'Exclusive_Access')
CALL PUTREP(fit, array1, 15, key1)
```

OPENM Call

Purpose Opens a keyed file.

Format **CALL OPENM (fit,pd,of)**

Parameters **(1) fit**

Variable containing the FIT pointer returned by the FILEIS call.

(2) pd

Type of processing:

'INPUT'

Open file for reading only (file statistics are not kept).

'OUTPUT'

Open file for writing only.

'I-O' or 'IO'

Open file for reading and writing.

'NEW'

A new file is being created; sets the $ACCESS_MODE (PD) FIT value to 'OUTPUT' and the old/new (ON) FIT value to 'NEW'.

If the call specifies 0 as the pd parameter value, the $ACCESS_MODE value in the FIT is used.

(3) of

File positioning when the file is opened:

'R'

Rewind the file (position the file to read the record with the lowest key value). This is the default if the $OPEN_POSITION value in the FIT is zero.

'E'

Position the file after the record with the highest key value. (A GETN call at this position would return end-of-information (EOI) status.)

If the call specifies 0 as the of parameter value, the $OPEN_POSITION value in the FIT is used.

Remarks

- The OPENM call to open a keyed file must precede all other keyed-file interface calls except FILEDA, FILEIS, IFETCH, and STOREF calls.
- When opening an existing file, the old/new (ON) value in the FIT or on the call must be 'OLD'. If the ON value is 'NEW', OPENM returns a fatal error ($ERROR_STATUS value AA3030).

  Similarly, when opening a new file, the old/new (ON) value in the FIT or on the call must be 'NEW'. If the ON value is 'OLD', OPENM returns a fatal error ($ERROR_STATUS value AA3040).
- The access modes requested when the file is opened determine the processing allowed on the file. For example, if you specify 'INPUT' on the OPENM call, you cannot call PUT to write a record to the file.

- An existing file must be attached with the appropriate usage mode set for the type of processing (read permission for 'INPUT', write permissions for 'OUTPUT', or read and write permissions for 'I-O').
- If zero is specified as the pd parameter on the call, the $ACCESS_MODE value in the FIT is used. If the program has not stored an $ACCESS_MODE value in the FIT, the file is opened for read access only ('INPUT').
- Multiple instances of open are allowed for a file. Each instance of open must have its own FIT. So before the program attempts to open an already open file, it must call FILEIS or FILEDA to create another FIT.
- An OPENM call performs these steps:
  1. OPENM checks the old/new (ON) flag in the FIT to determine if the file is a new file or an existing file.
     a. If the file is a new file, OPENM creates the file in the $LOCAL catalog using the file name specified by the $LOCAL_FILE_NAME value in the FIT.
     b. If the file is an existing file, OPENM searches for the file in the $LOCAL catalog using the file name specified by the $LOCAL_FILE_NAME in the FIT.
  2. OPENM initializes file attribute values in the FIT as follows:
     a. If the file is an existing file, OPENM verifies attribute values stored in the FIT against the corresponding attribute values preserved with the file. If the program has not stored a FIT value for a preserved attribute, OPENM copies the attribute value preserved with the file to the FIT.
     b. If SET_FILE_ATTRIBUTES commands specified one or more attribute values for the file before the program began, OPENM overwrites the corresponding values in the FIT. (Only temporary attribute values can be specified for an existing file.)
  3. OPENM checks that the FIT contains appropriate values for the keyed-file organization. It also checks that the values are consistent.
  4. OPENM positions the file according to the $OPEN_POSITION value.
  5. OPENM loads the collation-table module if the $KEY_TYPE value is COLLATED. (The entry point name used is the $COLLATE_TABLE_NAME FIT value.)
  6. It also loads the error-exit procedure if a value has been stored for $ERROR_EXIT_NAME.
  7. OPENM sets the open/closed (OC) flag in the FIT to open.

     The following is a list of some $ERROR_STATUS values that OPENM can return. For information on decyphering the value, see the $ERROR_STATUS description later in this chapter.

- Fatal errors:

  AA3245

  To open the file, you must have access to the file.

  AA3030

  An existing file cannot be opened as a new file.

  AA3310

  The hashing procedure for the direct-access file cannot be loaded.

  AA3370

  The compression procedure for the file cannot be loaded.

- Fatal errors when opening a new file:

  AA3210

  The maximum record length is 0 or undefined.

  AA3215

  The primary-key type is integer, but the key length is greater than 8.

  AA3220

  The key length cannot be greater than the maximum record length.

  AA3230

  The primary key must be within the minimum record length.

  AA3240

  The index padding attribute value is too large.

  AA3265

  The primary-key type is collated, but no collation table was specified.

  AA3275

  A nonzero key-length value must be specified.

- Nonfatal errors:

  160155

  When a file is opened with append access only, its open position must be at its EOI.

  160225

  To open the file, at least one access mode must be granted.

- Nonfatal errors when opening a new file only:

  160100

  The file cannot be created because the job has reached its local file limit already.

  160115

  When the record type is F (fixed-length records), the maximum record length cannot exceed the maximum block length.

  160150

  Opening a new file requires append access.

  160200

  The specified collation table could not be loaded.

  160205

  The specified error-exit procedure could not be loaded.

- Nonfatal errors when opening an existing file only:

  160095

  Preserved attribute values stored in the FIT do not match the corresponding preserved attribute values stored with the file.

  160120

  One or more of the requested access modes are not in the set of access modes granted when the file was attached.

  161016

  The local file name specified in the FIT is not known to the job.

PUT Call

Purpose Writes a record to a keyed file.

Format **CALL PUT (fit, wsa, rl, ka, kp, 0, ex)**

Parameters

(1) fit

Variable containing the FIT pointer returned by the FILEIS or FILEDA call that created the FIT.

(2) wsa

Working-storage area (location from which data is copied to the file).

NOTE

The working storage area and the key area should be in common blocks. If they are not, your program could execute incorrectly after being compiled with high optimization.

(3) rl

Record length in bytes (used only if the record type is variable length; ignored if the record type is fixed-length).

(4) ka

Key area (location containing the primary key value of the new record). This parameter is ignored for files with embedded keys.

(5) kp

For CYBER 170 compatibility. New programs should set this parameter to zero.

(6) 0

Reserved position for an unused parameter.

(7) ex

Error-exit procedure name.

$ERROR_STATUS	Nonfatal:	AA2005 -- key_already_exists
		AA2015 -- file_at_user_record_limit
		AA2020 -- file_full_no_puts_or_wraps
		AA2075 -- key_found_lock_no_wait
		AA2100 -- duplicate_alternate_key
		AA2605 -- key_required
		AA2615 -- non_embedded_key_not_given
		AA2650 -- not_enough_permission
		AA2865 -- sparse_key_beyond_eor
		AA2975 -- missing_key_delimiter
	Fatal:	AA3250 -- file_is_ruined
		AA3430 -- file_at_file_limit

Remarks

- A PUT call requires at least append access as indicated by the $ACCESS_MODE value in the FIT. If alternate keys are defined in the file, a PUT call requires append, shorten, and modify access in order to update the alternate indexes. If the file was opened without the required access, the PUT call returns the nonfatal $ERROR_STATUS value AA2650.

- Before the program calls PUT, it must store the record data in the working-storage area. If the primary key is nonembedded, it must also store the key value in the key area.

- The specified primary-key value must not already exist in the file. If it does, the PUT call returns a nonfatal error ($ERROR_STATUS value AA2000).

- You always specify a primary-key value on a PUT call, not an alternate-key value, even if an alternate key is currently selected.

- The PUT call updates each alternate index that is to include the new record. If the new record contains an alternate-key value that duplicates a value already in the alternate index and the alternate key does not allow duplicates, the PUT call returns a nonfatal error ($ERROR_STATUS value AA2100).

- If the file has fixed-length records, the record length (rl) value on the call (and in the FIT) is ignored. The length of the record written is always the fixed record length for the file.

  A warning message is issued for the first PUT, PUTREP, or REPLCE call whose rl value differs from the fixed record length for the file. The warning is given because, although excess data is truncated, insufficient data is not padded so garbage could be written as the last part of the record.

  For Better Performance

  When writing to an indexed-sequential file, the program should write records in order by ascending key values. This results in faster execution and a more efficient file structure. Your program could write the records to a sequential file and then call Sort/Merge to sort and write the records to an indexed-sequential file.

- A PUT call returns an error when it cannot write the record because the file has reached a limit. The $ERROR_STATUS value indicates the limit reached as follows:

  - Nonfatal errors:

    AA2015

    The number of records in the file has reached the $RECORD_LIMIT value.

    AA2020

    The record cannot be written because it would require addition of another index level to the indexed-sequential file and the file already has 15 index levels.

  - Fatal error:

    AA2655

    The number of bytes of file space has reached the FILE_LIMIT value. (The file structure is ruined.) It may be possible to re-create the file using the COPY_KEYED_FILE command.

- A PUT call does not reposition the file.
- When the file could be shared (more than one instance of open could be changing the file at the same time), the task should either:
  - Call LOCKK to lock the key value before it calls PUT.
  - Be prepared to process the $ERROR_STATUS value AA2075 returned if the key value is locked by another task.

PUTREP Call

Purpose Replaces an existing record or writes a new record to a keyed file.

Format **CALL PUTREP (fit, wsa, rl, ka, kp, 0, ex)**

Parameters

(1) fit

Variable containing the FIT pointer returned by the FILEIS or FILEDA call that created the FIT.

(2) wsa

Working storage area (location from which data is copied).

NOTE

The working storage area and the key area should be in common blocks. If they are not, your program could execute incorrectly after being compiled with high optimization.

(3) rl

Record length in bytes (used only if the record type is variable length; ignored if the record type is fixed-length).

(4) ka

Key area (variable containing the key value of the record to be written or replaced).

(5) kp

For CYBER 170 compatibility only. New programs should set this parameter to zero.

(6) 0

Reserved position for an unused parameter.

(7) ex

Error-exit procedure name.

$ERROR_STATUS	Nonfatal:	AA2015 -- file_at_user_record_limit
		AA2020 -- file_full_no_puts_or_reps
		AA2075 -- key_found_lock_no_wait
		AA2100 -- duplicate_alternate_key
		AA2605 -- key_required
		AA2615 -- non_embedded_key_not_given
		AA2650 -- not_enough_permission
		AA2865 -- sparse_key_beyond_eor
		AA2975 -- missing_key_delimiter
	Fatal:	AA3250 -- file_is_ruined
		AA3430 -- file_at_file_limit

Remarks

- A PUTREP call requires at least append and shorten access as indicated by the $ACCESS_MODE value in the FIT. If alternate keys are defined in the file, a PUTREP call requires append, shorten, and modify access in order to update the alternate indexes. If the file was opened without the required access, the call returns $ERROR_STATUS value AA2650.

- You always specify a primary-key value on a PUTREP call, not an alternate-key value, even if an alternate key is currently selected.

- The PUTREP call updates each alternate index that is to include the new record. If the new record contains an alternate-key value that duplicates a value already in the alternate index and the alternate key does not allow duplicates, the PUTREP call returns a trivial error ($ERROR_STATUS value AA2100).

- PUTREP executes a put request if the specified primary key does not match any existing primary key. It executes a replace request if a matching primary key is found in the file.

- If the file has variable-length (U or V) records, the length of the record written is the record length (rl) value specified on the call (or, if omitted, the $WORKING_STORAGE_LENGTH value in the FIT).

  For a file with variable-length (U or V) records, the new record need not be the same length as the existing record; however, the new record length must be within the minimum and maximum record lengths for the file.

- If the file has fixed-length (F) records, the record length (rl) value on the call is ignored. The fixed record length is always the length of each record written to the file.

  A warning message is issued for the first PUT, PUTREP, or REPLCE call whose rl value differs from the fixed record length for the file. The warning is given because, although excess data is truncated, insufficient data is not padded so garbage could be written at the end of the record.

- A PUTREP call does not reposition the file.

- Unlike a REPLC call, a PUTREP call does not require the task to own a Preserve_Access_and_Content or Exclusive_Access lock on the record.

  However, when the file is shared (more than one instance of open could exist), the task should either:

  - Call LOCKK to lock the key value before it calls PUTREP, or

  - Be prepared to process the abnormal status code AA2075 returned if the key value is locked by another task.

REPLC Call

Purpose Replaces an existing record in a keyed file.

Format **CALL REPLC (fit, wsa, rl, ka, kp, 0, ex)**

Parameters

(1) fit

Variable containing the FIT pointer returned by the FILEIS or FILEDA call that created the FIT.

(2) wsa

Working storage area (variable from which data is copied).

NOTE

The working storage area and the key area should be in common blocks. If they are not, your program could execute incorrectly after being compiled with high optimization.

---

(3) rl

Record length in bytes (used only if the record type is U or V; ignored if the record type is F).

(4) ka

Key area (variable containing the primary key value of the record to be replaced).

(5) kp

For CYBER 170 compatibility only. New programs should set this parameter to zero.

(6) 0

Reserved position for an unused parameter.

(7) ex

Error-exit procedure name.

$ERROR_STATUS	Nonfatal:	AA2000 -- key_not_found
		AA2020 -- file_full_no_puts_or_reps
		AA2085 -- key_not_already_locked
		AA2100 -- duplicate_alternate_key
		AA2605 -- key_required
		AA2615 -- non_embedded_key_not_given
		AA2650 -- not_enough_permission
		AA2865 -- sparse_key_beyond_eor
		AA2975 -- missing_key_delimiter
	Fatal:	AA3250 -- file_is_ruined
		AA3430 -- file_at_file_limit

Remarks

- A REPLC call requires at least append and shorten access as indicated by the $ACCESS_MODE value in the FIT. If alternate keys are defined in the file, a REPLC call requires append, shorten, and modify access in order to update the alternate indexes. If the file was opened without the required access, the call returns nonfatal $ERROR_STATUS value AA2650.

- If the file could be shared (more than one instance of open could be changing the file at the same time), a record can be replaced only by the owner of a Preserve_Access_and_Content or Exclusive_Access lock on the record.

  The task should lock the primary-key value by calling GET, GETN, or LOCKK before the REPLC call.

  To read about locks, see Keyed File Sharing earlier in this section.

- A REPLC call always specifies a primary-key value, not an alternate-key value, even if an alternate key is currently selected.

- The new record must have the same primary-key value as the record being replaced. If REPLC cannot find a record with a matching primary-key value, it returns a nonfatal error ($ERROR_STATUS value AA2005).

- The REPLC call updates each alternate index that is to include the new record.

  If the new record contains an alternate-key value that duplicates a value already in the alternate index and the alternate key does not allow duplicates, the REPLC call returns a nonfatal error ($ERROR_STATUS value AA2100).

- A REPLC call does not reposition the file.

- If the record type for the file is U or V, the record length is the $WORKING_STORAGE_LENGTH (wsl) value in the FIT.

  For a file with variable (U or V) records, the new record need not be the same length as the existing record; however, the new record length must be within the minimum and maximum record lengths for the file.

- If the file has fixed-length (F) records, the record length (rl) value on the call is ignored; the fixed record length for the file is always used.

  A warning message is issued for the first PUT, PUTREP, or REPLC call whose rl value differs from the fixed record length for the file. The warning is given because, although excess data is truncated, insufficient data is not padded.

REWND Call

Purpose Rewinds the file.

Format **CALL REWND (fit)**

Parameters **(1) fit**

Variable containing the FIT pointer returned by the FILEIS or FILEDA call that created the FIT.

Remarks

- When the primary key is selected, REWND positions an indexed-sequential file at its lowest primary-key value and a direct access file at the beginning of its first block.
- If the currently selected key is an alternate key, REWND positions the file to read the record with the lowest value for that alternate key.
- The $FILE_POSITION value after a successful REWND call is always 16 (end-of-record). It is not 1 (beginning-of-information).
- The file must be open when you issue the rewind request.

For Better Performance

Rewinding a file is more efficient than extensive backward skipping of records.

RMKDEF Call

Purpose Creates an alternate key in a keyed file.

NOTE

The NOS/VE RMKDEF call is provided for compatibility when migrating CYBER 170 programs that contain RMKDEF calls. When writing a new NOS/VE program, you should call the SCL utility CREATE_ALTERNATE_INDEXES using the SCLCMD call. (The CREATE_ALTERNATE_INDEXES utility is described in the SCL Advanced File Management Usage manual.)

Format **CALL RMKDEF (fit, akw, akp, akl, 0, akt, aks, akg, akc, anl, aie, ach, asp, avc)**

Parameters **(1) fit**

Name of the variable containing the file information table (FIT) pointer. This parameter is required.

(2) akw

Integer that, with the akp value, defines the key position. The akw value is multiplied by ten and added to the akp value to determine the byte position of the beginning of the key.

It is recommended that you specify zero as the akw parameter value so that the entire key position value is specified by the akp parameter.

(3) akp

Integer that, with the akw value, defines the key position. The akw integer is multiplied by ten and added to the akp value to determine the byte position of the beginning of the key. Bytes are numbered from the left, beginning with zero.

(4) akl

Key length in bytes (1 through 255).

A zero value indicates that this call and the following RMKDEF call define a key using sparse key control. (See the following Remarks.)

For a variable-length key, this parameter defines the maximum length of the key values.

(5) 0

Reserved position for unused parameter.

(6) akt

Key type.

'COLLATED' or 'C' or 'S'

Collated key (valid only for an indexed-sequential file with a collated primary key)

'INTEGER' or 'I'

Integer key (invalid for a variable-length key)

'UNCOLLATED' or 'UC' or 0

Uncollated key

NOTE

Unlike the SCL utility CREATE_ALTERNATE_INDEXES, RMKDEF cannot specify a separate collation table for the alternate key. A collated key created by RMKDEF uses the collation table for the primary key, which is available only if the file is an indexed-sequential file with a collated primary key.

---

(7) aks

Duplicate key control attribute.

'NOT_ALLOWED' or 'NA' or 'U' or 0

No duplicate key values allowed in the alternate index.

'ORDERED_BY_PRIMARY_KEY' or 'OBPK' or 'I'

Duplicate key values allowed; the key list for each value is kept in sorted ascending order.

'FIRST_IN_FIRST_OUT' or 'FIFO' or 'F'

Duplicate key values allowed; the key list for each value is not sorted so the primary-key values are in chronological order. (FIFO ordering is not allowed with repeating groups or variable-length keys.)

(8) akg

Optional repeating-group length. A zero value indicates that each record can contain only one value for the alternate key.

A nonzero value indicates that each record can contain more than one value for the alternate key.

For a fixed-length key (last parameter, avc, omitted), a nonzero akg value specifies the length, in bytes, of the repeating group of fields, that is, the distance from the beginning of an alternate-key value to the beginning of the next alternate-key value.

(9) akc

Indicates whether the search for alternate-key values continues to the end of the record.

0 (zero)

Search continues to the end of the record.

Nonzero positive integer

Search ends at the specified limit (valid only if akg is nonzero).

For a fixed-length key (avc omitted), a nonzero integer is the number of alternate-key values in each record. (Unless sparse-key control is used, the specified number of alternate-key values must fit in the minimum record length.)

For a variable-length key (avc specified), a nonzero integer is the length, in bytes, of the key field. The contents of the field is read as a sequence of key values, separated by delimiters. The end of the last key value is marked by a delimiter, the end of the field, or the end of the record.

(10) anl

Null-suppression attribute.

'FALSE' or 'F' or 0

Null suppression is not used.

'TRUE' or 'T' or 'N'

Null suppression is used.

(11) aie

Optional sparse-key control effect.

'INCLUDE_KEY_VALUE' or 'I' or 0

Include the alternate-key value in the alternate index if the sparse-key control character matches.

'EXCLUDE_KEY_VALUE' or 'E'

Exclude the alternate-key value from the alternate index if the sparse-key control character matches.

If a nonzero value is specified for aie, a nonzero value must be specified for ach.

(12) ach

Sparse-key control characters (character string from 1 through 256 characters long). This parameter must be nonzero if sparse-key control is used.

(13) asp

Sparse-key control position (integer). Bytes are numbered from the left, beginning with zero. See Remarks.

(14) avc

Character string (0 through 256 bytes) containing the delimiter characters for the key. Specify this parameter to define the key as a variable-length key. (Each delimiter character can occur only once in the string.)

Remarks

- The RMKDEF parameters are not FIT fields. Specifying a nonzero parameter value on an RMKDEF call does not implicitly store the value in the FIT. Specifying zero for a parameter does not cause a FIT field value to be used.

- The three values specifying the key position and length (akw, akp, and akl), taken together, must be unique among the file keys. RMKDEF cannot create an alternate key having the same position and length as the primary key. (The SCL utility CREATE_ALTERNATE_INDEXES can create an alternate key having the same position and length as the primary key.)

RMKDEF uses the three values (akw, akp, and akl) to name the key. For example, an alternate key specified using the values 567, 9, and 250 as its akw, akp, and akl values, respectively, is given the name ALTERNATE_567_9_250.

- An alternate key that is to use sparse-key control can be specified using one or two RMKDEF calls. (The two-call method is provided for CYBER 170 compatibility.)

  - Using the one-call method, the call specifies the byte position of the sparse-key control character as the thirteenth parameter value, asp.

  - Using the two-call method, the first call specifies only the first four parameters: the fit, the akw and akp parameters specifying the sparse-key control position, and a zero value for the akl parameter. (The thirteenth parameter, asp, is omitted.)

    The second call specifies the alternate-key field position and length as the akw, akp, and akl parameters. It also specifies any optional attributes desired for the key.

  See the sparse-key control examples under Examples.

- If the terminate_break_character (CCP default, control-t; CDCNET default, %2) is entered while RMKDEF is applying the alternate_key definition to the file, the terminal user receives a prompt, requesting confirmation of his or her intentions.

  The terminal user should enter a carriage return or any entry other than RUIN FILE (uppercase or lowercase) to continue the application of the alternate-key definition. No file operation can be performed on a ruined file; no data can be retrieved from the file. You should attempt to re-create the file using the SCL command COPY_KEYED_FILE.

  If the apply operation is allowed to complete, the CREATE_ALTERNATE_INDEXES utility can remove any unwanted alternate keys without harm to the file.

- Entry of a pause_break_character (CCP default, control-p; CDCNET default, %1) is ignored during application of alternate-key definitions.

- RMKDEF cannot specify a separate collation table for an alternate key. RMKDEF must use the collation table for the primary key.

  A primary-key collation table is available only if the file is an indexed-sequential file with a collated primary key. RMKDEF cannot create a collated key for a direct access file or a file with an integer or uncollated primary key.

  When RMKDEF cannot create the key, use the SCL utility CREATE_ALTERNATE_INDEXES described in the SCL Advanced File Management manual.

- RMKDEF cannot create a concatenated key. To create a concatenated key, execute the SCL utility CREATE_ALTERNATE_INDEXES described in the SCL Advanced File Management manual.

- An alternate-key definition cannot specify both first-in, first-out (FIFO) duplicate-key value control and repeating groups.

  It also cannot define a variable-length key with:

  - FIFO duplicate-key value control
  - Null suppression
  - Sparse-key control

Examples This call defines a 25-byte alternate key beginning at byte position 152. By default, the key type is uncollated and no duplicate key values are allowed.

```
CALL RMKDEF(fit,15,2,25)
```

This call defines a 2-byte alternate key beginning at byte position 0. The alternate key field repeats each 100 bytes. The key type is the default, uncollated; duplicate key values are allowed and are ordered by primary key.

```
CALL RMKDEF(fit,0,0,2,0,0,'OBPK',100,10)
```

This call defines the same alternate key as the first example, except that the alternate key now uses sparse-key control. The sparse-key control character is at byte position 5. The sparse-key characters are 1, 2, or 3. If the sparse key matches, the alternate-key value for the record is excluded from the alternate index.

```
CALL RMKDEF(fit,15,2,25,0,0,0,0,0,0,'E','123',5)
```

These calls define the same alternate key as the preceding example, except that the two-call method is used to specify sparse-key control as described under Remarks.

```
CALL RMKDEF(fit,0,5,0)
CALL RMKDEF(fit,15,2,25,0,0,0,0,0,0,'E','123')
```

This call defines the entire record as a single variable-length uncollated key. The key length is defined as the maximum record length for the file (80 bytes).

```
 CALL RMKDEF(fit, 0, 0, 80, 0, 'UNCOLLATED',
+            'ORDERED_BY_PRIMARY_KEY',0,0,'FALSE',0,0,0,'')
```

The following call is the same as the preceding one except it defines the key as a repeating group (rgl=1) and defines several delimiter characters. The call defines each string delimited by punctuation, digits, or spaces as a key value.

```
 CALL RMKDEF(fit, 0, 0, 80, 0, 'UNCOLLATED',
+            'ORDERED_BY_PRIMARY_KEY', 1,0,'FALSE',0,0,0,
+            '!@#$%^&*()_+-={}[]~`:";''\!<>?,./0123456789 ')
```

Notice that the apostrophe delimiter is specified using two apostrophe characters.

RSBUILD Call

Purpose Gets primary-key values from a keyed file and combines them with a result set.

Format **CALL RSBUILD (fit,source_result_set, target_result_set, low_key, major_low_key, low_key_relation, high_key, major_high_key, high_key_relation, logical_operation, new_result_placement, actual_result_set_placement, condition_code)**

Parameters **(1) fit**

Name of the variable containing the FIT pointer for the keyed file.

(2) source_result_set

Identifier of the result set to be combined (as returned by its RSOPEN call).

(3) target_result_set

Identifier of the target result set (as returned by its RSOPEN call).

(4) low_key

Key value at which the range begins. The value must be valid for the key type (integer for an integer key, characters for a collated or uncollated key).

(5) major_low_key

For a fixed-length key, a nonzero value indicates that the low end of the range is to be found by a major-key search. The specified value is the number of leftmost bytes of the low_key value to be used as the major key. A zero value indicates that the full low_key value is to be used.

For a variable-length alternate key, a nonzero value is required because it specifies the length of the low_key value.

(6) low_key_relation

Indicates where the range begins in relation to the lowest key value in the range.

'GREATER_KEY' or 'GK' or 'GT'

Exclude the lowest key value.

'EQUAL_KEY' or 'EK' or 'EQ' or 'GREATER_OR_EQUAL_KEY' or 'GOEK' or 'GE'

Include the lowest key value.

'LOWEST_KEY' or 'LK'

Start at the beginning of the index. (Ignore the low_key and major_low_key values.)

(7) high_key

Key value at which the range ends. The value must be valid for the key type (integer for an integer key, characters for a collated or uncollated key).

If the high_key value is less than the low_key value, RSBUILD returns the nonfatal $ERROR_STATUS value AA2750.

(8) major_high_key

For a fixed-length key, a nonzero value indicates that the high end of the range is to be found by a major-key search. The specified value is the number of leftmost bytes of the high_key value to be used as the major key. A zero value indicates that the full high_key value is to be used.

For a variable-length alternate key, a nonzero value is required because it specifies the length of the high_key value.

(9) high_key_relation

Indicates where the range begins in relation to the highest key value in the range.

'GREATER_KEY' or 'GK' or 'GT'

Include the highest key value.

'EQUAL_KEY' or 'EK' or 'EQ' or 'GREATER_OR_EQUAL_KEY' or 'GOEK' or 'GE'

Exclude the highest key value.

'HIGHEST_KEY' or 'HK'

Stop at the end of the index. (Ignore the high_key and major_high_key values.)

(10) logical_operation

Integer specifying the logical operation performed to combine the source result set with the new range of key values.

0 Logical AND. The combined result set is the intersection of the original result sets. It contains only those key values that belong to both of the original sets.

1 Logical OR. The combined result set is the union of the original result sets. It contains all key values from both original result sets.

2 Logical XOR. The combined result set is the union of the original result sets without the intersection of the original result sets. It contains all key values from each of the original result sets that do not belong also to the other result set.

(11) new_result_placement

Integer specifying the result set file to which the combined result set is written.

0 The combined result set overwrites the source result set. Use this value when the source result set is no longer needed.

1 The combined result set is written to the target result set. Use this value when the source_result_set is to be saved for later use. It is also used on the initial AMP$BUILD_RESULT_SET call for a new result set.

2 The placement of the combined result set is chosen to provide the fastest performance. The location chosen is returned in the variable specified by the actual_result_set_placement parameter on the call. Use this value when the source result set is no longer needed and the source and target result sets differ.

(12) actual_result_set_placement

Integer variable in which the call indicates the result set file to which the combined result set has been written.

0 The source result set has been overwritten.

1 The combined result set has been written to the target result set; the source result set has been preserved.

(13) condition_code

Integer variable in which the error status value is returned. A zero value indicates successful completion. For information on deciphering the condition_code, see the $ERROR_STATUS description later in this chapter.

$ERROR_STATUS	Nonfatal:	AA2000 -- key_not_found
		AA2750 -- high_end_not_above_low_end
	Fatal:	AA3365 -- keyed_file_expected
		AA3535 -- file_not_open
		AA3540 -- data_files_differ
		AA3545 -- nested_files_differ
		AA3550 -- target_not_given
		AA3555 -- wrong_data_file

Remarks

- RSBUILD adds a range of key values to a result set. It can be used to:
  - Add primary-key values to an empty result set.

    For this use, the call specifies the same result set as the source_result_set and as the target_result_set, but the new_result_placement value should be 1 (result_in_target). The logical_operation value should be 1 (logical OR).
  - Add primary-key values to an existing result set. The combined result set can overwrite the source result set or be written to the target result set.

    When the source result set is to be overwritten, the call specifies the same result set as the source_result_set and as the target_result_set. The new_result_placement value should be 0 (result_in_source).

    When the source result set is to be kept, the call specifies different result sets as the source result set and as the target result set and the new result placement value 1 (result in target).
  - When two result sets are specified, but it does not matter whether the source_result_set is overwritten, specify the new_result_placement value 2 (result_in_fastest_place).

- The specified data file, source result set, and target result set must be open. The data file is opened by an OPENM call; the result set is opened by an RSOPEN call.

  The data file and nested file identification in the result set files must match the data file cycle opened using the FIT and the nested file specified in the FIT. The file identification is stored in the result set when the result set is first opened.

  The currently selected nested file for the data file must be the nested file specified on the RSOPEN call. The nested file selected when the file is opened is the default nested file, $MAIN_FILE: to select another nested file, store the nested file name as the $NESTED_FILE_NAME value in the FIT.

- The currently selected key must be the key whose index is to be searched for the range specified on the call. The key selected when the file is opened is the primary key ($PRIMARY_KEY); to select another key, call STOREF to store the key name in the FIT.

  NOTE

  For a direct-access file, the selected key must be an alternate key. RSBUILD cannot use the primary key of a direct-access file.

- The search for the range specified on the call is the same as the range search performed by KLCOUNT. For more information, see the KLCOUNT call description.

- After finding the specified range in the index, the call gets the primary-key values from the index. If the index is for an alternate key which allows duplicate values, the call gets the list of primary-key values for each alternate-key value in the range.

RSCLEAR Call

Purpose Discards the existing result set in the result set file.

Format **Call RSCLEAR (result_set_id, condition_code)**

Parameters

(1) result_set_id

Identifier of the result set to be cleared (as returned by its RSOPEN call).

(2) condition_code

Integer variable in which the error status value is returned. A zero value indicates successful completion.

For information on decyphering the condition_code, see the $ERROR_STATUS description later in this chapter.

Fatal:	AA3535 -- file_not_open
	AA3580 -- invalid_result_set_id

Remarks The RSCLEAR call is used to erase the existing result set in a result set file after it has been opened by an RSOPEN call. After the file is cleared, it is equivalent to a new result set file.

RSCLOSE Call

Purpose Closes an open result set.

Format **Call RSCLOSE (result_set_id, condition_code)**

Parameters

(1) result_set_id

Result set identifier (as returned by its RSOPEN call).

(2) condition_code

Integer variable in which the error status value is returned. Return of a zero value indicates successful completion.

For information on decyphering the condition_code, see the $ERROR_STATUS description later in this chapter.

Fatal:	AA3535 -- file_not_open
	AA3580 -- invalid_result_set_id

Remarks

- Closing a result set prevents further operations using the result set until it is opened again.
- If an RSCLOSE call is not issued for an open result set, the result set is closed at task termination.
- A closed result set continues to exist until its file is deleted.

RSCOMB Call

Purpose Combines two result sets.

Format **CALL RSCOMB (first_result_set, second_result_set, target_result_set, logical_operation, new_result_placement, actual_result_set_placement, condition_code)**

Parameters

(1) first_result_set

Identifier of the first result set to be combined (as returned by its RSOPEN call).

(2) second_result_set

Identifier of the second result set be combined (as returned by its RSOPEN call).

If the new_result_placement parameter specifies 0 (result in source), the second source_result_set is overwritten.

(3) target_result_set

Identifier of the target result set (as returned by its RSOPEN).

(4) logical_operation

Integer specifying the logical operation performed to combine the two source result sets.

0 Logical AND. The combined result set is the intersection of the original result sets. It contains only those key values that belong to both of the original sets.

1 Logical OR. The combined result set is the union of the original result sets. It contains all key values from both original result sets.

2 Logical XOR. The combined result set is the union of the original result sets without the intersection of the original result sets. It contains all key values from each of the original result sets that are not in both of the original result sets.

(5) new_result_placement

Integer specifying the result set file to which the combined result set is written.

0 The combined result set overwrites the second source result set. Use this value only when the second source result set is no longer needed or the second source result set and the target result set are the same.

1 The combined result set is written to the target result set. Use this value when the second source result set is to be saved for later use.

2 The placement of the combined result set is chosen to provide the fastest performance. The location chosen is returned in the actual_result_set_placement variable. Use this value when the second source result set is no longer needed and the second source result set and target result set differ.

(6) actual_result_set_placement

Integer variable in which the call indicates the result set file to which the combined result set has been written.

0 Result in source. The second source result set has been overwritten.

1 Result in target. The combined result set has been written to the target result set file; the second source result set has been preserved.

(7) condition_code

Integer variable in which the error status value is returned. A zero value indicates successful completion.

For information on decyphering the condition_code, see the $ERROR_STATUS description later in this chapter.

Fatal:	AA3535 -- file_not_open
	AA3540 -- data_files_differ
	AA3545 -- nested_files_differ
	AA3550 -- target_not_given
	AA3580 -- invalid_result_set_id

Remarks

- The RSCOMB call performs the same combination operations that can be performed by an RSBUILD call. When possible, use RSBUILD to perform the combination at the same time the key values are taken from the data file.
- All result sets specified on the call must be open. However, the data file to which the result set applies need not be open.

  If the data file is open, its selected nested file need not be the nested file to which the result set applies. This is because the RSCOMB call does not require any information from the data file.
- All result sets specified on the call must apply to the same keyed file cycle and nested file in the keyed file. (The first RSOPEN call for a result set stores the identification of the data file cycle and nested file to which the result set file applies in the result set.)

RSDLTE Call

Purpose Deletes a primary-key value from a result set.

Format **CALL RSDLTE (target_result_set, key_location, condition_code)**

Parameters **(1) target_set_id**

Identifier of the result set from which the primary-key value is deleted (as returned by its RSOPEN call).

(2) key_location

Location of the primary-key value to be deleted from the result set.

If RSDLTE cannot find the specified key value in the result set, it returns the warning status AA1335.

(3) condition_code

Integer variable in which the error status value is returned. A zero value indicates successful completion.

For information on decyphering the condition_code, see the $ERROR_STATUS description later in this chapter.

Fatal:	AA3535 -- file_not_open
	AA3580 -- invalid_result_set_id

Remarks

- If the key value is not in the result set, the call does nothing and no message is issued.
- Use this call when only a few scattered primary-key values need to be deleted from the result set.

  When several primary-key values need to be deleted, it is more efficient to create a temporary result set containing those values and combine it with the original result set.

  For more information, see Adding and Deleting Key Values in the Result Sets description earlier in this section.
- This call can specify a primary-key value only. It cannot specify an alternate-key value.

  However, you can delete the primary-key values associated with an alternate-key value from the result set. To do so, perform the following steps:

  1. Select the alternate key.
  2. Call RSBUILD specifying the logical XOR (2) operation to remove the key values. It specifies the key values to be removed as a range containing only the one alternate-key value. (The low_key and high_key values of the range are the same.)
  3. If any of the primary-key values in the alternate key list might not be in the original result set, combine the target result set again with the original result set using a logical AND (0) operation.

RSGETN Call

Purpose Reads a record from a keyed file using a result set.

Format **CALL RSGETN (fit, source_result_set, result_set_not, wsa, ka, ex)**

Parameters

(1) fit

Name of the variable containing the FIT pointer for the keyed file.

(2) source_result_set

Identifier of the result set used to read the record (as returned by its RSOPEN call).

(3) result_set_not

Indicates whether the call reads the next record that is in the result set or the next record that is not in the result set.

'NO'

Reads the next record in the result set.

'YES'

Reads the next record NOT in the result set.

(4) wsa

Working storage area (location to which the record data is copied). (The default is the $WSA value in the FIT.)

(5) ka

Variable in which the primary-key value of the record is returned. (The default is the $KA value in the FIT.)

(6) ex

Error exit procedure name. (The default is the $EEPN FIT value.)

$ERROR_STATUS	Nonfatal:	AA2000 -- key_not_found
		AA2010 -- record_longer_than_wsa
		AA2075 -- key_found_lock_no_wait
		AA2620 -- wsa_not_given
		AA2635 -- cant_position_beyond_bound
		AA2977 -- illegal_key_or_nf_selected
	Fatal:	AA3535 -- file_not_open
		AA3580 -- invalid_result_set_id
		AA3590 -- is_file_expected
		AA3595 -- repeated_read_at_eoi

Remarks

- The RSGETN call is intended to be used to read a sequence of records. The sequence of records can be the records in the result set or all records in the data file that are not in the result set.

  To read the records in the result set, specify 'NO' for the result_set_not parameter on each RSGETN call. To read the records NOT in the result set, specify 'YES' for the result_set_not parameter on each RSGETN call.

NOTE

For a direct-access file, RSGETN can read the sequence of records in the result set, but it cannot read the sequence of records not in the result set. In other words, the NOT operator is invalid so each RSGETN call for a direct-access file must specify 'NO' for the result_set_not parameter.

- The data file must be open. The selected nested file must be the nested file specified when the result set was first opened. The selected key for the nested file must be its primary key.

- The RSOPEN call establishes the result set position at its beginning. (The result set is also positioned at its beginning by any of the calls that change the result set.)

  Each RSGETN call without the NOT operator repositions the result set forward one primary-key value. RSGETN calls with the NOT operator position the result set forward as needed.

  RSREWND, RSSTART, RSSKIP calls explicitly reposition the result set.

- Other calls can intervene between result set get calls. However, calls that reposition the data file must not intervene between result set get_not calls.

- An RSGETN call without the NOT operator issues a GET call using the primary-key value at the current result set position. It then advances the result-set position one primary-key value.

- RSGETN calls with the NOT operator get the records that are not in the result set. The first get_not call establishes the starting data file position.

  It does so by reading the primary-key value at the current result set position and then positioning the data file at that record. The first get_not call then reads a record, the same as subsequent get_not calls, as follows:

- Each get_not call performs the following steps:

  1. Calls GETN to read the record at the current data file position. (GETN reads a record and advances the data file position one record.)

  2. Compares the primary-key value returned by the GETN call and the primary-key value at the current position of the result set.

     a. If the values match, it:

        Discards the record read, advances the result set position forward one value, and continues at step 1.

     b. If the values do not match, it:

        Terminates, leaving the record read in the working storage area.

- Each call that returns a record, including the last record in the sequence, returns the $FILE_POSITION value 16 in the FIT. The next RSGETN call after the call that returns the last record returns the value 64, indicating that the end of the sequence has been reached. The call that returns 64 copies no data to the working storage area.

  A $FILE_POSITION value of 64 returned by a get call indicates that the result set is positioned at its end, and so all records in the result set have been read.

  A $FILE_POSITION value of 64 returned by a get_not call, indicates that the data file, as well as the result set, is positioned at its end, and so all records not in the result set have been read.

RSINFO Call

Purpose Returns current information about a result set.

Format **Call RSINFO (result_set_id, previous_key, next_key, key_count, keys_remaining, position, condition_code)**

Parameters

(1) result_set_id

Identifier of the result set for which information is returned (as returned by its RSOPEN call).

(2) previous_key

Variable in which the call returns the preceding primary-key value in the result set. Use an integer variable for an integer primary key; use a character variable for a collated or uncollated primary key. (A previous_key value is returned only when the position returned is 1 or 2.)

(3) next_key

Variable in which the call returns the next primary-key value in the result set. Use an integer variable for an integer primary key; use a character variable for a collated or uncollated primary key. (A next_key value is returned only when the position returned is 0 or 1.)

(4) key_count

Integer variable in which the call returns the number of primary-key values in the result set.

(5) keys_remaining

Integer variable in which the call returns the number of primary-key values from the current position to the end of the result set.

(6) position

Integer variable in which the call returns the relative position of the result set.

0 Positioned at its beginning (previous_key undefined).

1 Positioned somewhere between its beginning and end. (Both the previous_key and the next_key values are defined.)

2 Positioned at its end (next_key undefined).

3 The result set is empty. (The previous_key and next_key values are both undefined.)

(7) condition_code

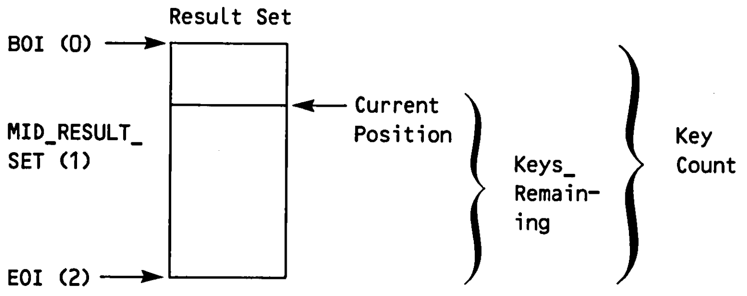
Integer variable in which the error status value is returned. A zero value indicates successful completion.

For information on decyphering the condition_code, see the $ERROR_STATUS description later in this chapter.

Fatal:	AA3535 -- file_not_open
	AA3580 -- invalid_result_set_id

Remarks

- The following figure illustrates the count and position values returned.



- Assuming the result set has not been repositioned since the call, the previous_key value is the primary_key value used by the last RSGETN call and the next_key value is the primary-key alue that will be used by the next RSGETN call.

- The key_count value returned is the number of primary-key values in the result set and therefore, the number of records that a series of get calls could fetch using the result set.

  However, to determine the number of records that series of get_not calls could fetch, your program must know the total number of records in the nested file and subtract the key_count value from that number. The record count for a nested file is not available from the keyed-file interface although you can display it using the DISPLAY_KEYED_FILE_PROPERTIES command.

- RSINFO calls do not change the result set position or the data file position and so do not disrupt a sequence of RSGETN calls.

RSOPEN Call

Purpose Opens a result set and positions it at its beginning.

Format **CALL RSOPEN (result_set_file, data_file, nested_file, result_set_id, condition_code)**

Parameters

(1) result_set_file

Name of the result set file to be opened (1 through 31-character string). If an existing file is to be used, it must be attached with at least read access. Otherwise, if the file does not exist, RSOPEN creates it.

(2) data_file

Name of the keyed file to which the result set applies (1 through 31-character string). The file must be attached with at least read access.

(3) nested_file

Name of a nested file in the data file (1 through 31-character string). (To specify the default nested file, specify either the name $MAIN_FILE or all blanks.)

(4) result_set_id

Integer variable in which the result set identifier is returned. It is used by later result set calls to identify the open result set.

(5) condition_code

Integer variable in which the error status value is returned. A zero value indicates successful completion.

For information on decyphering the condition_code, see the $ERROR_STATUS description later in this chapter.

Fatal:	AA3555 -- wrong_data_file
	AA3560 -- wrong_nested_file
	AA3565 -- file_must_exist
	AA3575 -- system_heap_full

Remarks

- A result set must be opened by an RSOPEN call before it can be used for any purpose. The result set remains open until it is closed by an RSCLOSE call or by the termination of the task.
- If the specified result set file does not exist or is not attached, RSOPEN creates a new temporary file and records in its file attributes that it is a result set. It also stores the identification of the specified data file and nested file in the result set.
- If the specified result set file is in the $LOCAL catalog, RSOPEN checks its attributes to ensure that it is a result set file. It also checks that the data file and nested-file identification in the result set file matches that of the data file and nested file specified on the call.
- The RSOPEN returns the identifier that all subsequent result set calls use to specify the result set.

NOTE

Do not change the contents of the result_set_id variable while the result set is open; any change would invalidate the identifier.

- The same result set identifier can be used with different instances of open of the data file. However, the result set may no longer be correct after the data file is updated. For more information, see Result Set Validity earlier in this section.
- A result set file can have only one instance of open at a time.

RSPUT Call

Purpose Adds a primary-key value to the result set.

Format **CALL RSPUT (target_result_set, key_location, condition_code)**

Parameters **(1) target_set_id**

Identifier of the result set to which the primary-key value is added (as returned by its RSOPEN call).

(2) key_location

Location of the primary-key value to be added to the result set.

(3) condition_code

Integer variable in which the error status value is returned. A zero value indicates successful completion.

For information on decyphering the condition_code, see the $ERROR_STATUS description later in this chapter.

Fatal:	AA3535 -- file_not_open
	AA3580 -- invalid_result_set_id

Remarks

- This call can be used to either:
  - Directly add a few scattered primary-key values to the result set.
  - Create a temporary result set of scattered primary-key values to be added or deleted from the result set.
- When several primary-key values need to be added, it is more efficient to create a temporary result set containing those values and combine it with the original result set. For more information, see Adding and Deleting Key Values in the Result Sets description earlier in this section.

  For Better Performance

  If possible, put primary-key values into a result set in ascending order.

- This call can specify a primary-key value only. It cannot specify an alternate-key value.

  However, you can add the primary-key values associated with an alternate-key value to the result set. To do so, perform the following steps:

  1. Select the alternate key.
  2. Call RSBUILD specifying the logical OR (1) operation to add the key values. The specified key-value range should contain only the one alternate-key value. (The low_key and high_key values of the range are the same value.)

RSREWND Call

Purpose	Repositions a result set at its beginning.
Format	**CALL RSREWND (source_result_set, condition_code)**

Parameters

(1) source_result_set

Identifier of the result set to be rewound (as returned by its RSOPEN call).

(2) condition_code

Integer variable in which the error status value is returned. A zero value indicates successful completion.

For information on deciphering the condition_code, see the $ERROR_STATUS description later in this chapter.

Fatal:	AA3535 -- file_not_open
	AA3580 -- invalid_result_set_id

Remarks

- The result set is also positioned at its beginning by an RSOPEN call and by any result set call that changes the result set.

RSSKIP Call

Purpose

Repositions a result set forward or backward.

Format

CALL RSSKIP (source_result_set, count, condition_code);

Parameters

(1) source_result_set

Identifier of the result set to be repositioned (as returned by its RSOPEN call).

(2) count

Number of primary-key values to be skipped. A positive integer causes a skip forward (toward the end of the result set); a negative integer causes a skip backward (toward the beginning of the result set).

(3) condition_code

Integer variable in which the error status value is returned. A zero value indicates successful completion.

For information on decyphering the condition_code, see the $ERROR_STATUS description later in this chapter.

Fatal:	AA3535 -- file_not_open
	AA3580 -- invalid_result_set_id

Remarks

- A skip forward that encounters the end of the result set does not return an error. The result set is left positioned at its end. The next RSGETN call returns no data and a $FILE_POSITION value of 64 in the FIT.

  Similarly, a skip backward that encounters the beginning of the result set does not return an error. The result set is left positioned at its beginning.

  If necessary, the program can call RSINFO to get the result set position after a skip.

RSSTART Call

Purpose Positions a result set using a primary-key value.

Format **CALL RSSTART (source_result_set, key_location, major_key_length, key_relation, condition_code)**

Parameters

(1) source_result_set

Identifier of the result set to be repositioned (as returned by its RSOPEN call).

(2) key_location

Location containing the primary-key value at which the result set is to be positioned.

(3) major_key_length

Indicates whether the primary-key value is to be located by major key. A zero value specifies that a major key is not used; a nonzero value specifies the number of bytes in the major key.

(4) key_relation

Indicates whether the primary-key value in the file must match the primary-key value specified on the call.

0	The primary-key values must match.
1	If a matching primary-key value is not found, the next greater primary-key value is used.
2	The first primary-key value found that is greater than the specified primary-key value is used.

(5) condition_code

Integer variable in which the error status value is returned. A zero value indicates successful completion.

For information on decyphering the condition_code, see the $ERROR_STATUS description later in this chapter.

Nonfatal:	AA2130 -- key_not_located
Fatal:	AA3535 -- file_not_open
	AA3580 -- invalid_result_set_id

Remarks

- The RSSTART call establishes the result set position at the primary-key value specified by the call. Subsequent get or get_not calls use only the result set values from the start position to the end of the result set.

SKIP Call

Purpose Repositions a keyed file either forward or backward the specified number of records.

Format **CALL SKIP (fit, count)**

Parameters **(1) fit**

Variable containing the FIT pointer returned by the FILEIS or FILEDA call that created the FIT.

(2) count

Number of records to be skipped. A positive integer causes a skip forward (toward the end-of-information); a negative integer causes a skip backward (toward the beginning-of-information).

If zero is specified for the count parameter, the $SKIP_COUNT value in the FIT is used. If it is also 0, no skipping is done.

$ERROR_STATUS	Nonfatal:	AM665 -- improper_skip_count
		AM670 -- skip_requires_read_perm
		AM1245 -- skip_encountered_boi
		AM1260 -- skip_encountered_eoi
		AA2825 -- no_skip_in_da

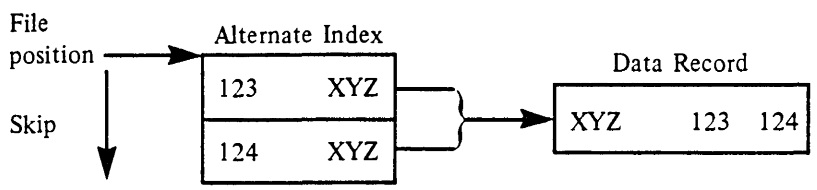
Remarks

- A SKIP call requires read access to the file.
- A SKIP call skips records in order by key value. This is because it actually skips key values in the index for the key.

  SKIP calls are valid only when an index exists for the selected key. Thus, SKIP calls are not valid while the primary key of a direct access file is selected. An attempt to do so returns the nonfatal error AA2825.

  If the currently selected key is an alternate key, it skips records in order by alternate-key value.

  The same record may be skipped more than once if it contains more than one alternate-key value. For example, suppose a record with primary-key value XYZ contains two integer alternate-key values, 123 and 124. Assume that the file is positioned to read the record with alternate-key value 123 as follows:



  A SKIP call to skip one record forward skips forward one alternate-key value in the alternate index. The file is then positioned to read the data record for alternate-key value 124, which is also the data record for alternate-key value 123.

For Better Performance

A skip call should be used for skipping a few records only, because each intervening record is read and counted, which increases execution time. A random read request takes less time than a lengthy skip request.

- The $FILE_POSITION value after a SKIP call is always 16, end-of-record (or 8, end-of-key-list, if an alternate key is selected) unless the SKIP reaches a file boundary (nonfatal error AA1005). The $FILE_POSITION value is then 1, beginning-of-information, or 64, end-of-information.

- A SKIP reaches a file boundary only when it cannot skip the requested number of records in the requested direction.

  For example, suppose the primary key is selected and the file is positioned to read the third record (the $FILE_POSITION is 16):

```
BOI..record1..record2..record3..EOI
                      ↑
```

  If a SKIP skips backward two records, the SKIP does not reach a file boundary and the $FILE_POSITION value is still 16:

```
BOI..record1..record2..record3..EOI
    ↑
```

  If the SKIP skips backward another record, it reaches the file boundary and the $FILE_POSITION value is 1. (The first record can be read from this position or the preceding position.)

```
BOI..record1..record2..record3..EOI
 ↑
```

  A SKIP now skips forward three records and the $FILE_POSITION value is 16.

```
BOI..record1..record2..record3..EOI
                              ↑
```

  A read at this position or one more skip forward reaches the file boundary, and the $FILE_POSITION value is 64.

```
BOI..record1..record2..record3..EOI
                                 ↑
```

  A skip backward one record positions the file to read the last record and the $FILE_POSITION value is 16.

```
BOI..record1..record2..record3..EOI
                      ↑
```

- When a skip encounters the end-of-information, it returns a nonfatal error ($ERROR_STATUS value AM1260). When a skip encounters the beginning-of-information, it also returns a nonfatal error ($ERROR_STATUS value AM1245).

  In either case, SKIP calls the DX procedure if one is specified in the FIT.

  If the program immediately calls SKIP again to skip in the same direction, SKIP calls the error-exit procedure (if one is specified in the FIT).

- If the skip reaches a file boundary and cannot be completed, the $SKIP_COUNT value in the FIT is the number of records that could not be skipped. The number of records actually skipped can be calculated by subtracting the residual skip count from the requested skip count.

STARTM Call

Purpose Positions a keyed file using the specified key value and key relation.

Format **CALL STARTM (fit, ka, kp, mkl, ex)**

Parameters

(1) fit

Variable containing the FIT pointer returned by the FILEIS or FILEDA call that created the FIT.

(2) ka

Key area (variable containing the key value used to position the file).

(3) kp

For CYBER 170 compatibility only. New programs should set this parameter to zero.

(4) mkl

For a fixed-length key, it is the length of the major key in bytes. A zero value indicates that the full key value is used.

For a variable-length key, a nonzero value is required because it specifies the length, in bytes, of the specified key value.

If the mkl value on the call is zero, the $MAJOR_KEY_LENGTH value in the FIT is used. However, the $MAJOR_KEY_LENGTH value is always reset to zero after any call with an mkl parameter.

(5) ex

Error-exit procedure name.

$ERROR_STATUS	Nonfatal:	AA2000 -- key_not_found
		AA2095 -- no_da_or_sk_start
		AA2615 -- non_embedded_key_not_given
		AA2640 -- major_key_too_long
		AA2650 -- not_enough_permission
	Fatal:	AA3250 -- file_is_ruined
		AA3430 -- file_at_file_limit

Remarks

- A STARTM call requires read access to the file.
- STARTM searches for the key value in the index of the selected key in the selected nested file only.

  A STARTM call is valid only when an index exists for the selected key. Thus, STARTM calls are invalid while the primary key of a direct access file is selected. An attempt returns the nonfatal error status AA2095.
- If an alternate key has been selected and the key is a concatenated key, the values for the pieces of the concatenated key are assembled in the key area. The pieces must be concatenated in the key area in the order defined for the alternate key.

  For example, if the key is the last three bytes of the record followed by the first three bytes of the record, the value in the key area must be the value of last three bytes followed by the value of the first three bytes.

- STARTM searches for the first key value that satisfies the relation specified by the $KEY_RELATION value in the FIT.
  - If the relation is EQUAL_KEY and an equal key value does not exist in the file, STARTM returns a nonfatal error ($ERROR_STATUS) value AA2000). The file is left positioned to read the next record (the record that would follow the specified record if it existed).
  - If the $KEY_RELATION value is GREATER_OR_EQUAL_KEY or GREATER_KEY and no key value in the file satisfies the relation, the data-exit (DX) procedure is called, if one is specified in the FIT. The file is left positioned at the end of information.
- STARTM cannot return a file position of 1 (beginning of information). When the key value to be found is less than any key value in the file, STARTM returns a file position value of 8 or 16 (end of key list or end of record).
- If the mkl value on the call or in the FIT is zero, the $MAJOR_KEY_LENGTH value in the FIT is used. The $MAJOR_KEY_LENGTH value in the FIT is cleared after any call having an mkl parameter. For more information, see the $MAJOR_KEY_LENGTH FIT value description.

  A nonzero mkl value is required while a variable-length alternate key is selected. Otherwise, a nonzero value is specified only when a major-key search is to be used.
- A STARTM call does not return a record to the working storage area.
- When an alternate key is selected and a primary-key area is specified in a FIT, a STARTM call returns the primary-key value of the record at which the file is positioned. The value is returned in the primary-key area.

STOREF Call

Purpose Stores a value in the FIT.

Format **CALL STOREF (fit, keyword, value)**

Parameters **(1) fit**

Variable containing the FIT pointer returned by the call that created the FIT.

(2) keyword

Character expression specifying a FIT keyword. The keyword can be specified using uppercase and/or lowercase letters.

(3) value

FIT value to be stored for the preceding keyword. The applicable values are listed in the individual keyword description. Character values can be specified using uppercase and/or lowercase letters.

Remarks

- You can call STOREF any time after the FILEIS or FILEDA call that returns FIT pointer.
- If the keyword specified is an SCL keyword, the value must be an SCL value. Similarly, if the keyword is a CYBER 170 keyword, the value must be a CYBER 170 value. For more information, see the discussion of FIT Keywords under FIT Values Introduction later in this section.
- To clear a FIT value, specify the keyword for the value and a 0 on a STOREF call.

  For example, suppose you previously specified a primary-key area, but now no longer want any primary-key values returned. To prevent this, you clear the $PRIMARY_KEY_ADDRESS value as follows:

```
CALL STOREF(fit, '$PRIMARY_KEY_ADDRESS', 0)
```

- Preserved file attribute values cannot be changed after the file has been first opened. These include:

$EMBEDDED_KEY	$MAXIMUM_BLOCK_LENGTH
$FILE_ORGANIZATION	$MAXIMUM_RECORD_LENGTH
$KEY_LENGTH	$MINIMUM_RECORD_LENGTH
$KEY_POSITION	$RECORD_TYPE
$KEY_TYPE	

  Specifying a value for $KEY_LENGTH or $KEY_POSITION after the file is first opened does not change the preserved attributes (the primary key length and position). Instead, the $KEY_LENGTH and $KEY_POSITION values can be used to select an alternate key or to specify the sparse-key control position for an RMKDEF call.

Examples

This call specifies that the key value is to be returned in the variable RETKEY. (RETKEY should be in a common block.)

```
CALL STOREF(fit, '$KEY_ADDRESS', retkey)
```

This call specifies the primary-key starting position as the beginning of the record.

```
CALL STOREF(fit, '$KEY_POSITION', 0)
```

This call clears the error-exit procedure specification.

```
CALL STOREF(fit, '$ERROR_EXIT_PROCEDURE', 0)
```

UNLOCKF Call

Purpose Clears a file lock for the currently selected nested file.

Format **UNLOCKF CALL (fit)**

Parameters **(1) fit**

Variable containing the FIT pointer returned by the FILEIS or FILEDA call. It specifies the instance of open whose file lock is to be cleared.

Remarks

- An UNLOCKF call clears only the file lock for the nested file specified by the $NESTED_FILE_NAME value in the FIT. It clears only the lock belonging to the instance of open.

  An UNLOCKF call clears only one nested file lock. It does not clear any other file locks or any key-value locks. To clear individual key-value locks or all locks, use UNLOCKK.

- If no lock exists for the specified instance of open, UNLOCKF returns the nonfatal $ERROR_STATUS value AA2090.

- When a lock expires, the task must clear the lock before it can perform any more operations on the instance of open. To clear all locks belonging to the instance of open (both file and key locks), call UNLOCKK with the 'ALL' parameter value specified.

  To read about lock expiration, see Lock Expiration and Clearing earlier in this section.

UNLOCKK Call

Purpose Clears either a single key-value lock or all locks for the currently selected nested file.

Format **CALL UNLOCKK (fit, ka, 'ALL')**

Parameters **(1) fit**

Variable containing the FIT pointer returned by the FILEIS or FILEDA call.

$ERROR_STATUS	Nonfatal:	AA2805 -- key_expired_lock_exists
		AA2850 -- key_last_exp_unlocked
		AA2855 -- key_non_last_exp_unlocked

(2) ka

Key area (location containing the primary-key value to be unlocked). Specify 0 for this parameter if 'ALL' is specified.

NOTE

The key area should be in a common block. If it is not, your program could execute incorrectly after being compiled with high optimization.

'ALL'

Requests clearing of all locks for this instance of open. If you specify 'ALL' as the third parmeter value, specify 0 for the second parameter (ka) value.

$ERROR_STATUS	Nonfatal:	AA2125 -- incompatible_unlocking
		AA2805 -- key_expired_lock_exists
		AA2850 -- key_last_exp_unlocked
		AA2855 -- key_non_last_exp_unlocked

Remarks

- An UNLOCKK call performs one of two operations depending on whether the third parameter value ('ALL') is specified:
  - If 'ALL' is specified, UNLOCKK clears all locks for the currently selected nested file (both the file lock, if any, and all key-value locks, if any).
  - If 'ALL' is omitted, UNLOCKK clears only the lock for the primary-key value at the specified key location (ka).
- A key value lock can be cleared without an UNLOCKK call:
  - It is cleared when the instance of open is closed.
  - If automatic unlock was requested for the lock, it is cleared when the task issues another call for the instance of open (other than an IFETCH or STOREF). (The lock is unlocked even if the request fails.)

NOTE

Do not call UNLOCKK to clear a key-value lock requested with automatic unlock. Such a call would first perform the automatic unlock and then the UNLOCKK operation. The second unlock operation would find no lock on the key value and issue the nonfatal error status AA2090.

- When 'ALL' is specified and no locks exist for the nested file, no error is returned. However, if 'ALL' is omitted and the instance of open does not own a lock on the key value, UNLOCKK returns the nonfatal $ERROR_STATUS value AA2090.
- When a lock expires, the task must clear the expired lock before it can perform any more operations on the instance of open.

  The task is notified that a lock has expired by the status returned by the next operation attempted. However, it is not notified as to which lock has expired.

  When notified that an expired lock exists, the task can either clear all locks or clear each lock individually. It can clear all locks by calling UNLOCKK with the 'ALL' option. An UNLOCKF call clears an individual file lock; UNLOCKK calls can clear individual key locks.

  While an expired lock exists, UNLOCKK calls that specify a key value return one of these nonfatal status values:

  AA2820

  The request fails because an expired lock exists. (You cannot unlock an unexpired lock while an expired lock exists.)

  AA2855

  This lock has expired and is cleared, but one or more additional expired locks exist.

  AA2850

  This lock has expired and is cleared, and no more expired locks exist.
- To read about lock expiration, see Lock Expiration and Clearing earlier in this section.

Examples

This call clears the lock on the key value in the variable specified by the $KEY_ADDRESS value in the FIT:

```
CALL UNLOCKK (fit)
```

This call clears the lock on the key value in variable KEY1 (and stores KEY1 as the $KEY_ADDRESS in the FIT):

```
CALL UNLOCKK (fit, key1)
```

This call clears all key-value and file locks for the currently selected nested file.

```
CALL UNLOCKK (fit, 0, 'ALL')
```

FIT Values

These are the keywords used to store and fetch FIT values. Most keywords have a $-prefixed SCL form, a $-prefixed SCL abbreviation form, and a CYBER 170 FORTRAN 5 form. (With a few exceptions, the FORTRAN 5 keywords are the same keywords used with AAM 2 in a FORTRAN 5 program.)

SCL Keyword	SCL Abbreviation	CYBER 170 Keyword
$ACCESS_MODE	$AM	PD
$AUTOMATIC_UNLOCK	$AU	AU
$AVERAGE_RECORD_LENGTH	$ARL	ARL
$COLLATE_TABLE	$CT	DCT
$COLLATE_TABLE_NAME	$CTN	CTN
$COMPRESSION_PROCEDURE_NAME	$CPN	CPN
$DATA_PADDING	$DP	DP
---	---	DX
$EMBEDDED_KEY	$EK	EMK
$ERROR_COUNT	$EC	ECT
$ERROR_EXIT_NAME	$EEN	EXN
$ERROR_EXIT_PROCEDURE	$EEP	EX
$ERROR_LIMIT	$EL	ERL
$ERROR_STATUS	$ES	ES
$ESTIMATED_RECORD_COUNT	$ERC	ERC
---	---	FNF
$FILE_IDENTIFIER	$FI	---
$FILE_ORGANIZATION	$FO	FO
$FILE_POSITION	$FP	FP
$FORCED_WRITE	$FW	FWI
$GET_AND_LOCK	$GAL	GAL
$HASHING_PROCEDURE_NAME	$HPN	HPN
$INDEX_LEVELS	$IL	NL
$INDEX_PADDING	$IP	IP
$INITIAL_HOME_BLOCK_COUNT	$IHBC	HMB
$KEY_ADDRESS	$KA	KA
$KEY_LENGTH	$KL	KL
$KEY_NAME	$KN	KN
$KEY_POSITION	$KP	RKP
$KEY_RELATION	$KR	REL
$KEY_TYPE	$KT	KT
$LAST_OPERATION	$LO	LOP
$LOCAL_FILE_NAME	$LFN	LFN
$LOCK_EXPIRATION_TIME	$LET	LET
$LOCK_INTENT	$LI	LI
$LOG_RESIDENCE	$LR	LR
$LOGGING_OPTIONS	---	---
$MAJOR_KEY_LENGTH	$MKL	MKL
$MAXIMUM_BLOCK_LENGTH	$MAXBL	MBL
$MAXIMUM_RECORD_LENGTH	$MAXRL	MRL
$MESSAGE_CONTROL	$MC	DFC

SCL Keyword	SCL Abbreviation	CYBER 170 Keyword
$MINIMUM_RECORD_LENGTH	$MINRL	MNR
$NESTED_FILE_NAME	$NFN	NFN
---	---	OC
---	---	ON
$OPEN_POSITION	$OP	OF
$PRIMARY_KEY_ADDRESS	$PKA	PKA
---	---	RL
$RECORD_LIMIT	$RL	FLM
$RECORD_TYPE	$RT	RT
$RECORDS_PER_BLOCK	$RPB	RPB
---	---	RKW
$SKIP_COUNT	$SC	SKP
WAIT_FOR_LOCK	$WFL	WFL
$WORKING_STORAGE_AREA	$WSA	WSA
$WORKING_STORAGE_LENGTH	$WSL	WSL

FIT Value Introduction

Some FIT values are file attributes; others are kept or used only by the FORTRAN keyed-file interface.

The FORTRAN interface values include the parameter values specified on keyed-file interface calls. If a value has not been stored for the parameter, the parameter value is initialized to 0 when the file is opened.

Unless indicated otherwise in the FIT value description, a parameter value specified on a call is stored in the FIT. The stored value becomes the value for that parameter until another value is specified for the parameter.

Other FIT values correspond to NOS/VE file attributes used outside the FORTRAN program. When the file is opened, these values are set as follows:

1. The attribute value specified on a SET_FILE_ATTRIBUTES command (if any).
2. For existing files, the attribute value stored with the file.
3. The attribute value specified by the FILEIS or FILEDA call or a STOREF call before the open.
4. The default value for the attribute.

NOS/VE file attributes fall into three categories: returned attributes, temporary attributes, and preserved attributes.

- Returned attributes are attributes whose values cannot be specified but can be fetched.
- Temporary attributes are attributes that are not stored with the file and may be changed each time the file is opened.

- Preserved attributes are attributes that are stored with the file when it is first opened and are preserved for the lifetime of the file.

In general, you cannot change a preserved file attribute after the file has been first opened. However, you can specify a preserved file attribute value in the FIT of an existing file to verify that the correct file is being used. For example, if you set the file_organization value to indexed-sequential in the FIT, the OPENM call checks that the preserved file_organization attribute is indexed-sequential. If it is not, OPENM returns an error.

FIT Keywords

To specify or fetch a FIT value, you specify the keyword for that value. Most of the FIT value keywords have three forms: an SCL name, an SCL abbreviation, and a CYBER 170 abbreviation.

In general, the SCL names and abbreviations are the same as those used by the SCL command SET_FILE_ATTRIBUTES, except that they have a dollar sign ($) prefix. The CYBER 170 abbreviations are identical to those used with CYBER 170 AAM 2 in FORTRAN 5 programs.

FIT keywords and character values can be specified using uppercase and/or lowercase letters.

FIT Value Forms

In some cases, two sets of values are defined for a keyword. The set of SCL values can be used only with the SCL keyword or abbreviation, and the set of CYBER 170 values can be used only with the CYBER 170 keyword.

For example, the following three STOREF calls are equivalent. The first call uses the SCL keyword, the second call uses the SCL abbreviation, and the third call uses the CYBER 170 keyword.

```
   CALL STOREF (AFIT, '$ACCESS_MODE', 'READ')
   CALL STOREF (AFIT, '$AM', 'R')
   CALL STOREF (AFIT, 'PD', 'INPUT')
```

In a value description, a Value Specified subsection describes the value you specify on a STOREF call. A Value Returned subsection describes the value returned by an IFETCH call.

$ACCESS_MODE ($AM or PD)

Purpose Set of access modes allowed for this instance of open (temporary attribute).

For an existing file, all modes in the set must be in the usage mode set specified when you attached the file.

Input SCL values (values specified with the $ACCESS_MODE or $AM keywords). (More than one mode may be specified.)

'READ'

Read access

'APPEND'

Append access

'SHORTEN'

Shorten access

'MODIFY'

Modify permission (file statistics are kept)

CYBER 170 values (values specified with the PD keyword)

'INPUT'

Read access (file statistics are not kept)

'OUTPUT'

Modify, shorten, and append access

'I-O' or 'IO'

Read, modify, shorten, and append access

'NEW'

Same as 'IO'. Specify 'NEW' only when creating a new file; it sets the old/new (ON) flag to 'NEW'

Output Integer as follows:

1	Read access only (file statistics are not kept)
2	Modify, shorten, and append access
3	Read, modify, shorten, and append access
4	Modify access only
5	Append access only
6	Shorten access only
7	Read and modify access
8	Read and append access
9	Read and shorten access
10	Modify and shorten access
11	Modify and append access
12	Shorten and append access
13	Read, modify, and shorten access
14	Read, modify, and append access
15	Read, shorten, and append access

Remarks

- **Default:** Read permission only ('INPUT').
- These are the access modes to the keyed file required for each call.

Call	Access Modes
CLOSEM	None
DLTE	Append, shorten, and modify
FILEDA	None
FILEIS	None
FLUSHM	Append, shorten, or modify
GET	Read[1]
GETN	Read[1]
IFETCH	None
KEYLIST	Read
KLCOUNT	Read
KLSPACE	Read
LOCKF	Any[1]
LOCKK	Any[1]
OPENM	Any
PUT	Append[2]
PUTREP	Append and shorten[2]
REPLC	Append and shorten[2]
REWND	Any
RMKDEF	Append, shorten, and modify
RSBUILD	Read
RSGETN	Read
SKIP	Any
STARTM	Read
STOREF	None
UNLOCKF	Any
UNLOCKK	Any

[1]If an Exclusive_Access lock is requested, shorten or append access is required.

[2]If one or more alternate keys are defined in the file, append, shorten, and modify access modes are required to update the alternate indexes.

- You can specify two or more values by enclosing the values in a single pair of apostrophes and separating the values with a comma.

NOTE

No spaces can separate the values, only a comma.

For example, the following STOREF call specifies read and modify access:

```
CALL STOREF (fit, '$AM', 'READ,MODIFY')
```

- Specifying a string of blanks requests no access modes. But, if you request no access modes, you cannot open the file.
- Although you can store another $ACCESS_MODE value while the file is open, the new value does not take effect until the next open.

$AUTOMATIC_UNLOCK ($AU or AU)

Purpose Indicates whether a lock should be cleared automatically (used when a lock is requested).

Input One of these strings:

'TRUE' or 'T' or 'YES' or 'Y' or 'ON'

The lock is cleared when the task issues a request for the instance of open (other than an IFETCH or STOREF call).

'FALSE' or 'F' or 'NO' or 'N' or 'OFF'

The lock is not cleared automatically. The lock is cleared by an UNLOCK call for the key value or when the instance of open closes.

Output One of these integers:

-1	Automatic unlock is requested (YES).
0	Automatic unlock is not requested (NO).

Remarks

- **Default:** YES (the lock is cleared automatically).
- This FIT value may be used by LOCKK, GET, and GETN calls as follows:
  - Used by LOCKK if the au parameter is omitted from the call.
  - Used when a GET or GETN call requests a lock, that is, when the FIT value $GET_AND_LOCK is YES (-1).

  NOTE

  Automatic unlock cannot be used with Preserve_Content lock intent.
- For an update request for the locked record, the automatic unlock does not occur until the operation completes. For all other requests, the automatic unlock occurs as soon as the request is issued.

$AVERAGE_RECORD_LENGTH ($ARL or ARL)

Purpose Estimated median record length, in bytes, of the data records to be stored in the file. (The length should not include the primary-key length if the primary key is nonembedded.)

NOS/VE uses this value to select the block size for a new file if the maximum block length for the file is not specified. This file attribute value is not preserved with the file because it is used only when the file is opened for the first time.

Input Integer from 1 through 65497.

Remarks

- **Default:** None. If the FIT value is zero when a new file is opened, NOS/VE uses the arithmetic mean of the minimum and maximum record lengths as the average record length when selecting the block size for the new file.

- When the file contains variable-length records, you should choose the average record length value as follows:

  - If almost all records in the file are nearly the same length, use that length.

  - If the record lengths are well-distributed, use the median record length.

$COLLATE_TABLE ($CT or DCT)

Purpose Variable defining the collation table for the primary key.

The value is used only when a new file is opened for the first time; it is not preserved with the file.

Input Name of a 256-character variable (CHARACTER*256). Each character in the variable is the collating weight for the corresponding ASCII character. (For example, the first character in the variable is the collating weight for the first ASCII character [code 00].)

Output A program should not fetch the $COLLATE_TABLE value from the FIT. It is stored as an address which the program cannot use.

Remarks

- **Default:** None. A collation table must be specified if the primary key type is collated. The collation table can be specified by the $COLLATE_TABLE or $COLLATE_TABLE_NAME value.
- OPENM copies the table from the variable to the internal entry point AAV$DCT. It then stores AAV$DCT as the collation table name and the collation table at AAV$DCT as the collation table for the new file.

$COLLATE_TABLE_NAME ($CTN or CTN)

Purpose Collation table for the primary key specified as the name of an entry point (preserved attribute). The value is used only when a new file is opened for the first time.

Input String of up to 31 characters specifying the entry point name.

Output The first 8 characters of the entry point name, left-justified, blank-filled (returned using uppercase letters even if the name was specified using lowercase letters).

Remarks

- **Default:** None. A collation table must be specified if the primary-key type is collated. The collation table can be specified by the $COLLATE_TABLE or $COLLATE_TABLE_NAME value.
- The collation table can be one of the NOS/VE predefined collation tables. The predefined collation tables are listed in appendix B.
- The COLSEQ routine can be used to create a table named FTV$USER_COLLATE_TABLE which can be specified as the $COLLATE_TABLE_NAME. For information on creating a collation table, see appendix H.
- The entry point can be in a module already loaded with the FORTRAN program or in a module in an object library in the program-library list. For a module to be loaded from an object library, it must be in the program-library list. For more information, see the SCL Object Code Management manual.

$COMPRESSION_PROCEDURE_NAME ($CPN or CPN)

Purpose Name of the data compression or encryption procedure (preserved attribute).

Input String of up to 31 characters specifying an entry point name in an object library in the current program library list.

The name must be enclosed in apostrophes ('name').

Output First 8 characters of the entry point name (letters are returned as uppercase letters even if specified as lowercase letters).

Remarks

- **Default:** None. Unless a procedure is specified when the file is created, no compression procedure is used.
- This FIT value can be specified only before the file is opened for the first time. The value is stored as a preserved attribute when the file is first opened.
- The compression procedure is not stored with the file. It must be loaded each time the file is opened. Therefore, the object library containing the compression procedure must be in the program library list.

  For example, this command adds a library to the library list:

  ```
  set_program_attributes, add_library=$user.my_obj_library
  ```

- A compression procedure name AMP$RECORD_COMPRESSION is provided with the system. It compresses strings of consecutive ASCII spaces, ASCII zeros, binary zeros, and nulls.

  When records are fetched from the file, AMP$RECORD_ COMPRESSION decompresses the record data to its original length and content.

  (The system-defined procedure is on a system library so you do not need to add it to your program library list.)

- Usually a compression procedure individually processes each byte of data for each record operation. This significantly increases the time required for each record operation. Therefore, you should specify a compression procedure only when the special processing it performs is worth the extra processing time.
- If you specify a compression procedure, you should consider its effect when specifying the file structure attributes. If you specify an $AVERAGE_RECORD_LENGTH value, it should be the average record length after data compression. Similarly, when creating a direct-access file, you should choose the INITIAL_HOME_BLOCK_COUNT value based on the size of the compressed file data.
- User-defined compression procedures can be written, but the procedures may be written in the CYBIL language only. For more information, see the CYBIL Keyed-File and Sort/Merge Interfaces manual.

- The NOS/VE compression procedure performs both compression and decompression, unlike CYBER 170 AAM 2 for which a compression routine (CPA) and a decompression routine (DCA) are required.

  Also, the primary-key field can be anywhere in the record. For compression on the CYBER 170, the primary key had to begin the record.

$DATA_PADDING ($DP or DP)

Purpose	Percentage of block space left empty as each data block is created during the first instance of open of an indexed-sequential file (preserved attribute).
Input	Integer from 0 through 99. The padding percentage must allow at least one maximum-length record to be written to each block.
Remarks	• **Default:** 0 (no data block padding).

Data Exit Procedure (DX)

Purpose End-of-data exit procedure.

Input Name of a subroutine that is declared as EXTERNAL.

Output A FORTRAN program should not fetch the DX value from the FIT. It is stored as an address which the program cannot use.

Remarks

- **Default:** None.
- The DX FIT value is provided for CYBER 170 FORTRAN compatibility. No SCL keyword is defined for the value.
- If a DX value has been stored in the FIT, a GETN or SKIP call calls the specified subroutine when the GETN or SKIP call encounters the beginning-of-information or end-of-information.
- The data-exit routine can determine whether the file is at its BOI or EOI by fetching the $FILE_POSITION value.

$EMBEDDED_KEY ($EK or EMK)

Purpose Indicates whether the primary key is embedded or nonembedded (preserved attribute).

Input SCL values (specified with $EMBEDDED_KEY or $EK)

'YES' or 'Y' or 'TRUE' or 'T' or 'ON'
Embedded key. (The key value is part of the record data.)

'NO' or 'N' or 'FALSE' or 'F' or 'OFF'
Nonembedded key. (The key value is separate from the record data.)

CYBER 170 values (specified with EMK)

'YES'
Embedded key. (The key value is part of the record data.)

'NO'
Nonembedded key. (The key value is separate from the record data.)

Output Integer as follows:

−1	Embedded key. (The key value is part of the record data.)
0	Nonembedded key. (The key value is separate from the record data.)

Remarks **Default:** Embedded key.

$ERROR_COUNT ($EC or ECT)

Purpose Number of trivial (nonfatal) errors that have been returned by keyed-file interface calls since the OPENM call.

Output Integer. The value is limited by a nonzero $ERROR_LIMIT value.

Remarks

- **Default:** Initialized to 0 when the file is opened.
- This attribute can be fetched only while the file is open.

$ERROR_EXIT_PROCEDURE_NAME or $ERROR_EXIT_NAME ($EEPN, $EEN or EXN)

Purpose Error-exit procedure specified as an entry point (temporary attribute).

Input String of up to 31 characters specifying the entry point name. The name must be enclosed in apostrophes ('name').

Output The first 8 characters of the entry point name, left-justified, blank-filled (returned using uppercase letters even if the name was specified using lowercase letters).

Remarks

- **Default:** None. If you do not specify a name before opening the file, the system does not load an error-exit procedure.

- The error-exit entry point may be an entry point already loaded with the program or an entry point in an object library. For a module to be loaded from an object library, it must be in the program library list. For more information on program libraries and the SET_PROGRAM_ATTRIBUTES command, see the SCL Object Code Management manual.

- You can clear the $ERROR_EXIT_PROCEDURE_NAME value by calling STOREF with a string of blanks. For example:

```
CALL STOREF (fit, '$ERROR_EXIT_PROCEDURE_NAME', ' ')
```

- The OPENM call gets the address of the $ERROR_EXIT_PROCEDURE_NAME procedure and stores it as the $ERROR_EXIT_PROCEDURE value in the FIT. Therefore, if you specify two error-exit procedures before opening the file: one using the $ERROR_EXIT_PROCEDURE_NAME keyword and the other, the $ERROR_EXIT_PROCEDURE keyword, the procedure specified using $ERROR_EXIT_PROCEDURE_NAME is used.

- Storing an $ERROR_EXIT_PROCEDURE_NAME value after the file is open has no effect; the value is used only if the file is re-opened using the same FIT.

  To change the error-exit procedure for the current open, specify an error-exit procedure parameter on a keyed-file interface call or specify the $ERROR_EXIT_PROCEDURE value on a STOREF call.

$ERROR_EXIT_PROCEDURE ($EEP or EX)

Purpose	Error-exit procedure specified as a subroutine name (parameter).
Input	Name of a subroutine that is declared as EXTERNAL in the calling program.
Output	A program should not fetch the $ERROR_EXIT_PROCEDURE value from the FIT. It is stored as an address which the program cannot use.

Remarks

- **Default:** None. The error-exit procedure specified by the ERROR_EXIT_PROCEDURE_NAME attribute before the file was opened (if any) is used.
- Specifying a value using the $ERROR_EXIT_PROCEDURE keyword changes the effective error-exit procedure immediately. (Specifying a value using the $ERROR_EXIT_PROCEDURE_NAME keyword changes the procedure only when the file is opened.)
- A nonzero value specified with the $ERROR_EXIT_PROCEDURE keyword is stored as the default error-exit procedure value in the FIT. It becomes the default eep parameter value until another eep value is specified on a call.
- To clear the $ERROR_EXIT_PROCEDURE value, call STOREF to store a value of zero as the $ERROR_EXIT_PROCEDURE value.

$ERROR_LIMIT ($EL or ERL)

Purpose	Nonfatal (trivial) error limit (temporary attribute). When the limit is reached, a fatal error is returned.
Input	Integer between 0 and 65535. 0 allows unlimited trivial errors.
Remarks	• **Default:** 0 (no limit). • ERROR_LIMIT is compared to ERROR_COUNT to determine when the error limit has been reached. For more information on error processing, see the earlier subsection, Keyed-File Interface Error Processing.

$ERROR_STATUS ($ES or ES)

Purpose Error status code returned by the previous keyed-file interface call.

Input Integer. (Specifying a value does not affect the value returned in the field.)

Output Integer status condition code. A zero value indicates the previous keyed-file interface call completed successfully, without error.

The condition code for an abnormal status consists of the two-byte product identifier (such as AA or AM) and a three-byte value specifying the particular error condition for that product.

To reference an AA condition code, you should define a statement function in your program such as the following:

```
INTEGER AA
AA(i) = 256**3 * (256 * ichar('A') + ichar('A')) + i
```

To reference a AM condition code, you should define a statement function in your program such as the following:

```
INTEGER AM
AM(i) = 256**3 * (256 * ichar('A') + ichar('M')) + i
```

To reference the error status values AA3230 and AM0100, your program would reference the AA and AM functions as follows:

```
   number = IFETCH(fit, 'ES')
   IF (number .EQ. AA(3230))
   THEN
C    process AA3230 error
   ELSEIF (number .EQ. AM(0100))
   THEN
C    process AM0100 error
     ENDIF
```

Remarks

- **Default:** Initialized to 0 before each keyed-file interface call.
- If an error-exit procedure has not been specified, the program should fetch the error status value after each keyed-file interface call. A nonzero value returned indicates that the call did not complete successfully.
- The Diagnostic Messages for NOS/VE manual lists the meaning of each status condition code.

ESTIMATED_RECORD_COUNT ($ERC or ERC)

Purpose Estimated number of data records to be stored in the file.

NOS/VE uses this value to select the block size for a new file if the maximum block length for the file is not specified. This file attribute value is not preserved with the file because it is used only when the file is opened for the first time.

Input Integer from 1 through 4398046511103 (2**42 - 1).

Remarks

- **Default:** The $RECORD_LIMIT value if specified. If no $RECORD_LIMIT value is specified, an estimate of 100,000 records is used.

Fatal/Nonfatal Flag (FNF)

Purpose	Indicates whether the severity of the last error for the file is fatal or nonfatal.
Input	Integer values as follows: 0 The error severity is nonfatal (trivial). −1 The error severity is fatal.
Remarks	This value is not defined outside the FORTRAN keyed-file interface. No SCL keyword or SCL abbreviation is defined for the FIT value.

$FILE_IDENTIFIER ($FI)

Purpose Returns the CYBIL file identifier for the current open of the file.

Output Integer.

Remarks

- An IFETCH call can fetch the file identifier only while the file is open. The file identifier cannot be fetched before the OPENM call or after the CLOSEM call.
- A FORTRAN program fetches the file identifier so that it can pass it to a CYBIL procedure. The CYBIL procedure requires the file identifier so that it can issue file interface calls for the open file.
- To receive the file identifier value as a parameter, a CYBIL procedure declaration specifies a VAR declaration of type AMT$FILE_IDENTIFIER. For example:

```
PROCEDURE cybil_proc (VAR fi: amt$file_identifier);
```

- The CYBIL procedure must not close a file opened in the FORTRAN program. A file opened by an OPENM call must be closed by a CLOSEM call (or by program termination). Otherwise, the results of the file operations are undefined.
- File interface calls made outside the FORTRAN program do not update the FIT. The CYBIL subprogram should not call AMP$STORE to change file attribute values because the changed values are not copied to the FIT. Subsequent calls to IFETCH would then return out-of-date information.

$FILE_ORGANIZATION ($FO or FO)

Purpose File organization (preserved attribute). (The file organization determines the method of storing and accessing file data.)

Input SCL value (specified with $FILE_ORGANIZATION or $FO)

'INDEXED_SEQUENTIAL' or 'IS'
Indexed-sequential file organization

'DIRECT_ACCESS' or 'DA'
Direct access file organization

CYBER 170 value (specified with FO)

'IS'
Indexed-sequential file organization

'DA'
Direct access file organization

Output Integer as follows:

3	Indexed-sequential file organization
5	Direct access file organization

Remarks

- **Default:** Set by the call that created the FIT. FILEIS sets the file organization to indexed-sequential; FILEDA sets the file organization to direct access.

$FILE_POSITION ($FP or FP)

Purpose Indicates the position of the file after the last keyed-file interface call (returned attribute).

Input One of the following integers:

1 File is positioned at the beginning-of-information (BOI).

8 File is positioned at the end of a key list (returned only if an alternate key is currently selected).

16 File is positioned at the end of a record (EOR), but not at the end of a key list.

64 File is positioned at the end-of-information (EOI).

Remarks When the file is opened, but before any records are processed, $FILE_POSITION has the same value as $OPEN_POSITION. The default $OPEN_POSITION value is $BOI.

$FORCED_WRITE ($FW or FWI)

Purpose Indicates when modified blocks of the file are to be written to mass storage (preserved attribute).

Input SCL values (specified with $FORCED_WRITE or $FW)

'TRUE' or 'T' or 'YES' or 'Y' or 'ON'

The system writes each modified block to mass storage immediately after the modification.

'FORCED_IF_STRUCTURE_CHANGE' or 'FISC'

The system writes modified blocks to mass storage immediately if the change affects more than one block.

'FALSE' or 'F' or 'NO' or 'N' or 'OFF'

The system determines when modified blocks are copied to mass storage. Modified blocks can remain in memory without a mass-storage backup copy.

CYBER 170 values (specified with FWI)

'YES'

The system writes each modified block to mass storage immediately after the modification.

'NO'

The system writes modified blocks to mass storage immediately if the change affects more than one block.

'UNFORCED'

The system determines when modified blocks are copied to mass storage. Modified blocks can remain in memory without a mass-storage backup copy.

Output Integer as follows:

−1	The system writes each modified block to mass storage immediately after the modification (TRUE).
0	The system writes modified blocks to mass storage immediately if the change affects more than one block (FORCED_IF_STRUCTURE_CHANGE).
+1	The system determines when modified blocks are copied to mass storage. Modified blocks can remain in memory without a mass-storage backup copy (FALSE).

Remarks

- **Default:** FALSE. (The system determines when modified blocks are copied to mass storage. Modified blocks can remain in memory without a mass-storage backup copy.)
- You can request that the entire file be copied to disk by calling FLUSHM. FLUSHM copies internal tables as well as data and index blocks. (A $FORCED_WRITE copy does not copy internal tables.)

- If the file could be shared and the $FORCED_WRITE value is either −1 or 0, the block size of the file should be a multiple of the system page size.

  This ensures that multiple opens are not updating blocks in the same page. Otherwise, a forced-write operation could write a page that contains partially-altered blocks. (A warning message is issued if this possibility exists.)

$GET_AND_LOCK ($GAL or GAL)

Purpose Indicates whether a GET or GETN call issues a lock request for the key value before reading the record.

Input One of these strings:

'YES' or 'Y' or 'TRUE' or 'T' or 'ON'
A GET or GETN call requests a lock.

'NO' or 'N' or 'FALSE' or 'F' or 'OFF'
A GET or GETN call does not request a lock.

Output One of these integers:

-1	A GET or GETN call requests a lock (YES).
0	A GET or GETN call does not request a lock (NO).

Remarks

- **Default:** NO (a GET or GETN call does not request a lock).
- These FIT values are used as parameter values for the lock if the $GET_AND_LOCK value is YES (-1):
  $AUTOMATIC_UNLOCK
  $LOCK_INTENT
  $WAIT_FOR_LOCK

$HASHING_PROCEDURE_NAME ($HPN or HPN)

Purpose Name of the hashing procedure used to hash primary-key values for the direct access file (preserved attribute).

Input String of up to 31 characters specifying an entry point name from an object library in the current program library list. The name must be enclosed in apostrophes ('name').

Output First 8 characters of the entry point name (letters are returned as uppercase letters even if specified as lowercase letters).

Remarks

- **Default:** AMP$SYSTEM_HASHING_PROCEDURE (the system default hashing procedure).
- This FIT value can be specified only before the file is opened for the first time. The value is stored as a preserved attribute when the file is first opened.
- A user-defined hashing procedure can be written in the CYBIL language only. For more information, see the CYBIL Keyed-File and Sort/Merge Interfaces manual.
- The hashing procedue is not stored with the file. It must be loaded each time the file is opened. Thus, the object library containing the hashing procedure must be in the program library list.
- Although any ring-attributes value is valid for the object library containing the hashing procedure, you should store the hashing procedure in a ring 4 object library.

  This improves performance because hashing procedures are executed as asynchronous tasks. (Usually, site personnel maintain the ring 4 object libraries.)
- A hashing procedure can be specified by name only; it cannot be specified by address. (The CYBER 170 FIT value HRL is not supported.)

$INDEX_LEVELS ($INDEX_LEVEL, $IL, or NL)

Purpose For a new indexed-sequential file, the target number of index levels or, for an existing indexed-sequential file, the current number of index levels.

Input Target number of index levels (integer from 0 through 15). The system uses this value as a guideline in its selection of the block size for a new file.

Output Current number of index levels (integer from 0 through 15). (An empty file has 0 index levels.)

Remarks

- **Default:** For a new file, 2 index levels.
- If specified before the file is created, NOS/VE uses the INDEX_ LEVELS value when selecting the block size for a new file if the maximum block length for the file is not specified. The specified value is not preserved with the file because it is used only when the file is opened for the first time.
- For an existing file, the value returned is the current number of levels of indexing in the indexed-sequential file.
- The current number of index levels can be fetched only while the file is open.

$INDEX_PADDING ($IP or IP)

Purpose	Percentage of block space left empty in each index block created during the first instance of open of the file (preserved attribute).
Input	Integer from 0 to 99. The padding percentage must allow at least three index records to be written to the block. (The index record length is the primary key length plus 4 bytes.)
Remarks	• **Default:** 0 (no index block padding).

$INITIAL_HOME_BLOCK_COUNT ($IHBC or HMB)

Purpose	Number of home blocks in the direct access file (preserved attribute).
Input	Integer from 1 through 4387945511193 (2**42 - 1).

Remarks

- **Default:** None. You must specify a value for this attribute when defining a new direct access file.
- This value specifies the number of blocks allocated for the new direct access file. The blocks should accommodate all records expected to be written to the file. The addition of more records would require allocation of overflow blocks, slowing access to the overflow records.
- The initial_home_block_count should allow for a loading factor of no more than 90%. In other words, allocate at least 10% extra space in the file because the hashing procedure may not uniformly distribute records among the home blocks.
- For best results, the initial_home_block_count should be a prime number.

$KEY_ADDRESS ($KA or KA)

Purpose Location of the key value, that is, the key area (parameter).

Input Variable name.

NOTE

The key area should be in a common block. If it is not, your program could execute incorrectly after being compiled with high optimization.

Output A FORTRAN program should not fetch the ka value from the FIT. It is stored as an address which the program cannot use.

Remarks

- A key address is required in these cases:
  - When a PUT call writes a record with a nonembedded key.
  - When a GET call reads a record by its primary-key value.
  - For any STARTM or LOCKK call.
- A key address is optional for a GET call when an alternate key is selected. GET reads the alternate-key value from the key address if a ka value is specified on the call or in the FIT.
- The ka value in the FIT is used when 0 is specified as the ka parameter on a call.
- If a keyed-file interface call specifies a ka value, the value is copied to the FIT. It becomes the default value for subsequent calls.

$KEY_LENGTH ($KL or KL)

Purpose	Key length (preserved attribute). It is the primary-key length for a new file. For an existing file, it is the key length when selecting a key by position and length.
Input	For an embedded key (primary or alternate), an integer from 1 through 255, but not greater than the minimum record length.
	For a nonembedded primary key, an integer from 1 through 255.
	For an integer key, an integer from 1 through 8.
Remarks	**Default:** None. You must specify the primary-key length before calling OPENM for a new file.

$KEY_NAME ($KN or KN)

Purpose Name of the selected key.

Input String of up to 31 characters specifying the key name. The name of the primary key is $PRIMARY_KEY.

Output The first 8 characters of the key name, left-justified, blank-filled (returned using uppercase letters even if the name was specified using lowercase letters).

Remarks

- **Default:** The primary key ($PRIMARY_KEY).
- A key name can be specified by the OPENM call or by a STOREF call while the file is open. It cannot be specified by the FILEIS or FILEDA call or by a STOREF call before the OPENM call or after the CLOSEM call.
- The name of an alternate key is defined when the key is defined. For more information, see the subsection Alternate-Key Creation earlier in this section.

$KEY_POSITION ($KP or RKP)

Purpose Byte position at which the key begins (preserved attribute).

It is the position of the primary key for a new file. For an existing file, it is the key position used when selecting a key by position and length. (See Selecting a Key earlier in this section.)

Input Integer from zero to the maximum record length for the file. However, the key position value added to the key length value must not exceed the minimum record length.

NOTE

The byte positions in a record are numbered from the left, beginning with zero.

Remarks **Default:** Zero. If the key is embedded, the key is assumed to begin at the leftmost byte of the record. If the key is nonembedded, the key position value is not used.

$KEY_RELATION ($KR or REL)

Purpose Relation between the key value in the record and the key value at the ka location.

Input SCL values (specified with $KEY_RELATION or $KR)

'EQUAL_KEY' or 'EK'

The record key value must be equal to the specified key value.

'GREATER_OR_EQUAL_KEY'or 'GOEK'

The record key value must be greater than or equal to the specified key value.

'GREATER_KEY' or 'GK'

The record key value must be greater than the specified key value.

CYBER 170 values (specified with KR)

'EQ'

The record key value must be equal to the specified key value.

'GE'

The record key value must be greater than or equal to the specified key value.

'GT'

The record key value must be greater than the specified key value.

Output Integer as follows:

1 The record key value must be equal to the specified key value.

3 The record key value must be greater than or equal to the specified key value.

6 The record key value must be greater than the specified key value.

Remarks

- **Default:** EQUAL_KEY. (The key value in the record must be equal to the specified key value.)
- The $KEY_RELATION value is used only by GET and STARTM calls. A GET call reads the first record that satisfies the relation. A STARTM call positions the file at the first record satisfying the relation.
- The $KEY_RELATION FIT value is not used by calls to a direct access file while its primary key is selected (because no index with ordered key values exists for the key).

$KEY_TYPE ($KT or KT)

Purpose Primary key type for a new indexed-sequential file (preserved attribute).

Input SCL values (specified with $KEY_TYPE or $KT).

'COLLATED' or 'C'

A key value is a string of characters; it is sorted byte-by-byte according to a user-specified collating sequence.

'INTEGER' or 'I'

A key value is a signed integer (8 bytes long); it is sorted in ascending numerical order.

'UMNGLATED' or 'U'

A key value is a string of characters; it is sorted byte-by-byte according to the default ASCII collating sequence.

CYBER 170 values (specified with KT)

'S'

A key value is a string of characters; it is sorted byte-by-byte according to a user-specified collating sequence.

'I'

A key value is a signed integer (8 bytes long); it is sorted in ascending numerical order.

'U'

A key value is a string of characters; it is sorted byte-by-byte according to the default ASCII collating sequence.

Output Integer as follows:

1 A key value is a string of characters; it is sorted byte-by-byte according to a user-specified collating sequence.

2 A key value is a signed integer (8 bytes long); it is sorted in ascending numerical order.

3 A key value is a string of characters; it is sorted byte-by-byte according to the default ASCII collating sequence.

Remarks

- **Default:** Uncollated keys ('U').
- The primary-key type for a direct access file is always uncollated, regardless of the specified value. (The primary-key values are not sorted so a sort-order specification is irrelevant.)

$LAST_OPERATION (LOP)

Purpose Last request for the file (returned attribute).

Input One of the following integers:

0	FILEIS (FIT created for an indexed-sequential file)
1	OPENM (open request)
2	CLOSEM (close request)
3	GET (random read request)
4	GETN (sequential read request)
5	PUT (write request)
8	DLTE (delete request)
9	REPLC (replace request)
10	REWND (rewind request)
11	PUTREP (put/replace request)
12	SKIP (skip forward request)
13	SKIP (skip backward request)
14	STARTM (start request)
19	RMKDEF (alternate-key definition request)
20	KLCOUNT (key-list count request)
21	KLSPACE (key-list block count request)
22	KEYLIST (key list request)
23	LOCKF (lock file request)
24	LOCKK (key value lock request)
25	UNLOCKF (clear file lock request)
26	UNLOCKK (clear key value lock request)
27	FILEDA (FIT created for a direct-access file)
28	FILESK (not implemented yet)
29	RSBUILD (result set build request)
30	RSGETN (result set get next request)

Remarks

- The following calls do not change the $LAST_OPERATION value in the FIT. After one of these calls, IFETCH returns the value of the preceding keyed-file interface call.
  - IFETCH, FLUSHM, and STOREF
  - The result set calls (other than RSBUILD and RSGETN)
- The keyword $LO is no longer supported for this FIT value.

$LOCAL_FILE_NAME ($LFN or LFN)

Purpose Name of the file in the $LOCAL catalog.

Input A valid SCL name. For a new file, the name cannot already exist in the $LOCAL catalog. For an existing file, the name must be the name of a file in the $LOCAL catalog. (It can be a temporary file or an attached permanent file.)

Output The first 8 characters of the file name (returned using uppercase letters even if the name was specified using lowercase letters).

Remarks

- **Default:** None. This is a required parameter; it must be specified by the FILEIS or FILEDA call that creates the FIT or a STOREF call before the file is opened.
- If the old/new flag (ON) is set to 'OLD', OPENM searches for a file with the specified name in the $LOCAL catalog. If the old/new flag (ON) is set to 'NEW', OPENM attempts to create a file with the specified name in the $LOCAL catalog.
- A FORTRAN program must set the $LOCAL_FILE_NAME (LFN) value in the FIT before calling OPENM. If the $LOCAL_FILE NAME value has not been specified, the OPENM call returns a fatal error.
- The LOCAL_FILE_NAME value cannot be changed while the file is open.

$LOCK_EXPIRATION_TIME ($LET or LET)

Purpose Number of milliseconds between the time a lock is granted and the time that it could expire (preserved attribute).

Input Integer from 0 through 604,800,000. (0 specifies an unlimited expiration time.)

Remarks

- **Default:** 60,000 milliseconds (60 seconds).
- An expired lock prevents further access to the file by the owner of the lock. To remove an expired lock, the owner must call UNLOCKK or close the instance of open.
- Although the lock expiration time is an attribute preserved with the file after its first open, the attribute value can be changed by the SCL command, CHANGE_FILE_ATTRIBUTE.
- To read about lock expiration, see Lock Expiration and Clearing earlier in this section.

$LOCK_INTENT ($LI or LI)

Purpose Purpose of the lock.

Input One of these strings (the string can be specified using uppercase and/or lowercase letters):

'Preserve_Content' or 'PC'
Preserve_Content for reading

'Preserve_Access_and_Content' or 'PAAC' or 'PAC'
Preserve_Access_and_Content for reading and possibly updating

'Exclusive_Access' or 'EA'
Exclusive_Access for updating (requires shorten or append access)

Output One of these integers:

0	Preserve_Content
1	Preserve_Access_and_Content
2	Exclusive_Access

Remarks

- **Default:** Preserve_Access_and_Content.
- This FIT value may be used when:
  - A GET or GETN call requests a lock, that is, when the FIT value $GET_AND_LOCK is YES (-1).
  - A LOCKF and LOCKK if the li parameter is omitted from the call.
- A Preserve_Content lock cannot be automatically unlocked. Also, a Preserve_Content lock must be cleared before the lock_intent for the lock can be changed to PAC or EA.
- An Exclusive_Access lock is allowed only if the instance of open has shorten and/or append access to the file.

$LOG_RESIDENCE ($LR or LR)

Purpose Catalog in which the update recovery log for the keyed file is written (preserved attribute).

Input Name of the character array containing the path to the log catalog.

Output First 8 characters of the log catalog path.

Remarks

- **Default:** None if the $LOGGING_OPTIONS value does not include M (enabling media recovery); otherwise, the default is $SYSTEM.AAM.SHARED_RECOVERY_LOG.

- The specified log must have been previously created using the Administer_Recovery_Log utility. (The default log is created during system installation.)

- Whenever you change the log residence attribute of an existing file to a log other than the default log, you should immediately backup the keyed file. Otherwise, if the file is damaged, the RECOVER_FILE_MEDIA option of the Recover_Keyed_File utility cannot execute successfully for the file.

- The Administer_Recovery_log utility and Recover_Keyed_File utility descriptions are in the SCL Advanced File Management Usage manual.

$LOGGING_OPTIONS

Purpose Options enabling use of keyed-file recovery options (preserved attribute).

Input String of characters, each specifying a logging option:

'P'

For future implementation.

'M'

Enable media recovery; the system maintains an update recovery log for the keyed file. (Update recovery logs are discussed in the SCL Advanced File Management manual.

'R'

Enable request recovery; when a task aborts, the automatic close removes any partially-completed update operation.

Output Logging option characters, left-justified and blank-filled in a word. The characters may not be returned in the same order used when the options were specified.

Remarks

- **Default:** No logging options selected.
- Multiple options can be specified in any order; for example, 'RMP' for all three logging options.

$MAJOR_KEY_LENGTH ($MKL or MKL)

Purpose Length of the key value to be used by the next STARTM or GET call. The location of the key value is given by the ka value.

For a fixed-length key, the value is the major-key length, the number of leftmost key-value bytes compared.

For a variable-length key, the value is the length of the key specified by the call.

Input Integer from 0 through the key length value.

Remarks

- **Default:** For a fixed-length key, 0 (the full key value is used).

  For a variable-length key, the key length is required so a nonzero value must be specified; otherwise, the call returns the nonfatal $ERROR_STATUS value AA2980.

- The $MAJOR_KEY_LENGTH FIT value is not used by calls to direct access file while the primary key is selected (because no index with ordered key values exists for the key).

- When using a major key for a fixed-length key, the call compares only the leftmost bytes of the key value.

- For a variable-length alternate key, the key value is compared with the full alternate-key value stored in the index, not just the leftmost bytes.

- The $MAJOR_KEY_LENGTH value is reset to zero after execution of the STARTM or GET call that uses the value.

- Major-key use with an integer key is not recommended. The leftmost bytes of an integer value are seldom meaningful beyond indicating the sign of the value.

$MAXIMUM_BLOCK_LENGTH ($MAXBL or MBL)

Purpose Block length, in bytes, for a new file (preserved attribute).

Input Integer from 1 through 65536. If the value is less than the maximum record length, it is increased to that value. Then, it is increased, if necessary, to the next power of 2 from 2048 through 65536.

If the specified value is less than the maximum record length, it is increased to that value. Then, if the value is not a power of 2 between 2048 and 65536, it is changed as follows:

- A value less than 2048 is increased to 2048 (the minimum allocation unit).
- A value between 2048 and 65536, but not a power of 2, is increased to the next power of 2 (4096, 8192, 16384, 32768, or 65536).
- A value greater than 65536 is decreased to 65536.

NOTE

If the file could be changed by more than one instance of open at the same time and forced-writing will be used (the $FORCED_WRITE attribute is -1 [TRUE] or 0 [FORCED_IF_STRUCTURE_CHANGE]), the block size should be a multiple of the system page size.

This ensures that more than one instance of open is not updating blocks in the same page; otherwise, a forced-write operation could write a page to mass storage that contains partially-altered blocks. (A warning message is issued if this situation exists.)

Remarks **Default:** The system selects the block length using the AVERAGE_RECORD_LENGTH, ESTIMATED_RECORD_COUNT, INDEX_LEVELS, and RECORDS_PER_BLOCK values, if specified. The minimum block length selected by the system is 1 page.

$MAXIMUM_RECORD_LENGTH ($MAXRL or MRL)

Purpose	Maximum record length, in bytes, for a new file (preserved attribute).
Input	Integer from 1 through 65497.
Remarks	**Default:** None. You must specify the maximum record length when creating a new keyed file.

$MESSAGE_CONTROL ($MC or DFC)

Purpose Indicates the additional information written to the $ERRORS file (temporary attribute).

Input SCL values (specified with $MESSAGE_CONTROL or $MC)

'MESSAGES' or 'M'
Informative messages

'STATISTICS' or 'S'
Statistic messages

'TRIVIAL_ERRORS' or 'T'
Trivial (nonfatal) error messages

' ' (one or more blanks)
No additional information (fatal and catastrophic messages only)

CYBER 170 values for MESSAGE_CONTROL are:

0 No additional messages (fatal and catastrophic messages only)

1 Nonfatal-error messages

2 Informative and statistic messages

3 All messages (catastrophic, fatal-error, nonfatal-error, informative, and statistic)

4 Informative messages

5 Statistic messages

6 Nonfatal-error and informative messages

7 Nonfatal-error and statistic messages

Output The integers listed above.

Remarks

- **Default:** 0 (no additional information). Only fatal and catastrophic error messages are written to the $ERRORS file.

  NOTE

  It is recommended that you request at least informative and trivial (nonfatal) error messages.

- To specify two or more values, enclose the values in a single pair of apostrophes; a comma is required between values. (Spaces are also allowed between values.)

$MINIMUM_RECORD_LENGTH ($MINRL or MNR)

Purpose Minimum record length, in bytes, for a new file (preserved attribute).

Input Integer from 0 through 65497 bytes. The value must be less than or equal to the maximum record length.

Remarks **Default:** For fixed-length records, the default value is 0; however, the length of each fixed-length record must be the $MAXIMUM_RECORD_LENGTH value.

For variable-length records with an embedded primary key, the default value is the sum of the key position and key length values. For variable-length records with a nonembedded primary key, the default value is 1 byte.

NOTE

For variable-length records, it is recommended that you explictly specify the minimum record length. The minimum record length must include the primary-key field and any alternate-key fields (or corresponding sparse-key control characters).

$NESTED_FILE_NAME ($NFN OR NFN)

Purpose Name of the selected nested file.

Input Name of an existing nested file in the file. (The FORTRAN keyed-file interface cannot create a new nested file.)

Output First 8 characters of the nested-file name (letters are returned as uppercase letters even if specified as lowercase letters).

Remarks

- **Default:** $MAIN_FILE (the default nested file).
- Storing a nested-file name in the FIT selects that nested file for use. All following calls operate on the selected nested file until another nested-file name is stored.
- A nested-file name can be specified only while the file is open. It cannot be specified by the FILEIS or FILEDA call or by a STOREF call before the OPENM call or after the CLOSEM call.
- When a nested file is selected for the first time during an instance-of-open, its open position is specified by the $OPEN_POSITION attribute of the file.

  Later re-selection of the nested file during the instance-of-open positions the file at its last position during its prior selection. Thus, a task can sequentially access records in one nested file, select another nested file, re-select the first nested file, and continue the sequential access.
- The first time a nested file is selected during an instance-of-open, the first key selected is the primary key.

  Later, when a nested file is re-selected during the instance-of-open, the selected key is set to the last key selected during the previous selection of the nested file. Thus, a task can select an alternate key, select another nested file, re-select the first nested file and continue use of the previously selected alternate key.
- Selection of another nested file does not release any locks.

  An expired lock status is not returned when locks expire for nested files other than the nested file currently selected. However, an expired lock status is returned if the task re-selects the nested file and attempts an operation on that nested file.
- The FORTRAN key-filed interface cannot create additional nested files in a keyed file. To do so, use the CREATE_KEYED_FILE utility, the SCL command COPY_KEYED_FILE or a CYBIL program.

Old/New Flag (ON)

Purpose Indicates whether the next OPENM call is to create a new file or open an existing file.

Input FORTRAN values (specified with ON)

'OLD'
The file exists.

'NEW'
The file is being created.

Output Integer as follows:

0	The file exists.
−1	The file is being created.

Remarks **Default:** Set to 'NEW' if the $ACCESS_MODES value is 'NEW'. Reset to 'OLD' by a CLOSEM call.

Open/Close Flag (OC)

Purpose	Indicates whether a file is open or closed.
Output	Integer as follows:

0	The file has never been opened.
1	The file is open.
2	The file is closed.

$OPEN_POSITION ($OP or OF)

Purpose Position at which the file is opened (temporary attribute).

Input SCL values (specified with $OPEN_POSITION or $OP)

'$BOI'
Open at beginning-of-information (BOI).

'$ASIS'
Open without changing the file position.

'$EOI'
Open at end-of-information (EOI).

CYBER 170 values (specified with OF)

'R'
Open at beginning-of-information (BOI).

'N'
Open without changing the file position.

'E'
Open at end-of-information (EOI).

Output Integer as follows:

1	Open at beginning-of-information (BOI).
3	Open at end-of-information (EOI).
4	Open without changing the file position.

Remarks

- **Default:** Open at beginning-of-information ('BOI').
- If an existing file is opened for append access only, the only valid open position is EOI.

$PRIMARY_KEY_ADDRESS ($PKA or PKA)

Purpose Location to which the primary-key value is returned.

Input Variable name.

NOTE

The primary-key area should be in a common block. If it is not, your program could execute incorrectly after being compiled with high optimization.

Output A FORTRAN program should not fetch the pka value from the FIT. It is stored as an address which the program cannot use.

Remarks

- **Default:** 0 (the primary-key value is not returned).
- If the pka value in the FIT is nonzero, get calls issued while an alternate key is selected return the primary-key value of the record read to the specified location.

Record Length (RL)

Purpose	Either the number of bytes written by a PUT call or the number of bytes read by the last GET or GETN call (parameter).
Input	Integer from 1 through the maximum record length.
Remarks	**Default:** When writing a variable-length (U or V) record, the record length must be specified. When writing a fixed-length (F) record, the maximum record length is used as the record length value.

$RECORD_LIMIT ($RL or FLM)

Purpose	Maximum number of records in the file (preserved attribute).
Input	Integer from 1 through 4398046511103 ([2**42] - 1).
Remarks	• **Default:** 4398046511103 ([2**42] - 1) • After the file is first opened, the RECORD_LIMIT attribute value is stored with the file. However, you can change the RECORD_LIMIT attribute value of an existing file with the SCL command CHANGE_FILE_ATTRIBUTES.

$RECORD_TYPE ($RT or RT)

Purpose Record type (preserved attribute).

SCL values (specified with $RECORD_TYPE or $RT)

'VARIABLE' or 'V'
CDC variable-length records.

'FIXED' or 'F'
ANSI fixed-length records.

'UNDEFINED' or 'U'
Undefined-length records.

CYBER 170 values (specified with RT)

'V'
CDC variable-length records.

'F'
ANSI fixed-length records.

'U', 'S', or 'W'
Undefined-length records.

Output Integer as follows:

0	CDC variable-length (V) records.
1	ANSI fixed-length (F) records.
7	Undefined-length (U, S, or W) records.

Remarks

- **Default:** Undefined-length (U) records.
- The keyed-file interface processes record types U, V, S, and W the same.
- The keyed-file interface does not support the trailing_character_ delimited record type.
- The S and W values are provided for CYBER 170 compatibility.

$RECORDS_PER_BLOCK ($RPB or RB)

Purpose	Estimated number of records to be stored in each data block of a new file. NOS/VE uses this value to select the block size for a new file if the maximum block length for the file is not specified. This file attribute value is not preserved with the file because it is used only when the file is opened for the first time.
Input	Integer from 1 through 65535.
Remarks	**Default:** Two records per block.

Relative Key Word (RKW)

Purpose	Value that, with the RKP ($KEY_POSITION) value, defines the key position.
Input	Integer. The RKW value is multiplied by 10 and added to the RKP value. The key position value cannot exceed the maximum record length.
Remarks	● **Default:** 0 (the key position is defined by the RKP value). ● The RKW value is provided for CYBER 170 FORTRAN compatibility. Do not use it when writing new programs. Specify the key position by the $KEY_POSITION value only.

$SKIP_COUNT ($SC or SKP)

Purpose Either the number of records to be skipped by the next SKIP call (parameter) or the residual skip count from the last SKIP call.

Input Integer. If the skip count is positive, a SKIP call skips forward the specified number of records. If the skip count is negative, a SKIP call skips backward the specified number of records.

Output A zero skip count indicates that the skip operation completed. A nonzero value indicates that the skip operation did not complete.

A nonzero value returned is the residual skip count. A residual skip count is the difference between the requested skip count and the actual number of records skipped.

Remarks

- **Default:** 0 (no file repositioning).
- The returned skip count is nonzero when the SKIP call encounters the BOI or EOI of the file before it completes the skip. To determine the file position, call IFETCH to return the $FILE_POSITION value.

$WAIT_FOR_LOCK ($WFL or WFL)

Purpose Indicates whether the lock request should wait until the lock is available or the time limit has been reached.

Input One of these strings:

'YES' or 'Y' or 'TRUE' or 'T' or 'ON'
The request waits for the lock.

'NO' or 'N' or 'FALSE' or 'F' or 'OFF'
The request does not wait for the lock.

Output One of these integers:

-1	The request waits for the lock (YES).
0	The request does not wait for the lock (NO).

Remarks

- **Default:** YES (the request waits for the lock).
- This FIT value may be used by GET, GETN, LOCKF, and LOCKK calls as follows:
  - Used by GET and GETN calls when the FIT value $GET_AND_LOCK is YES (-1).
  - Used by LOCKF and LOCKK if the wfl parameter is omitted from the call.
- When waiting is requested, the call checks for a possible deadlock. If a deadlock exists with another task, it immediately returns the nonfatal-error status AA2040.
- If the lock is owned by another instance-of-open of the same task, a self-deadlock exists and the call immediately returns the nonfatal error status AA2045.
- You can change the maximum waiting period for the lock (used if wfl is YES). (The default value is 60 seconds.) To change the waiting period, create an SCL integer variable named AAV$RESOLVE_TIME_LIMIT and initialize it to the new waiting period value in seconds (any integer greater than 1). For example, this call executes an SCL command that sets the waiting period at 45 seconds:

```
   CALL SCLCMD ('create_variable, name=AAV$RESOLVE_TIME_LIMIT,
  + kind=integer, value=45')
```

  Be aware of the scope of the AAV$RESOLVE_TIME_LIMIT variable. The default scope is LOCAL. If the time limit change should apply to all tasks, specify SCOPE=JOB on the CREATE_VARIABLE command.

$WORKING_STORAGE_ADDRESS ($WSA or WSA)

Purpose Location to which data is read and from which data is written (parameter).

Input Variable name.

NOTE

The working storage area should be in a common block. If it is not, your program could execute incorrectly after being compiled with high optimization.

Output A FORTRAN program should not fetch the wsa value from the FIT. It is stored as an address which the program cannot use.

Remarks

- You can specify the wsa location either on a STOREF call or on a get or put call. When you specify a wsa location on a call, the wsa location is stored in the FIT and used by all subsequent get or put calls until another wsa location is specified.
- The length of the working-storage area is stored in the FIT as the wsl value.

$WORKING_STORAGE_LENGTH ($WSL or WSL)

Purpose	Length, in bytes, of the working storage area (parameter).
Input	Integer greater than or equal to the maximum record length value.
Remarks	• **Default:** For read requests, the maximum record length value; for write requests, the record length value. • For fixed-length records, if the wsl value does not match the maximum record length, the wsl value is ignored and a warning message issued.

Sort/Merge 12

Sort/Merge 12

Sort/Merge is a set of powerful and efficient routines which operates under the NOS/VE operating system. Sort/Merge can be used with a single command, or with procedure calls from within a program written in COBOL, CYBIL, or FORTRAN.

This chapter introduces the functions and features of Sort/Merge using FORTRAN procedure calls.

What Sort/Merge Does

The purpose of sorting is to arrange items in order. The purpose of merging is to combine two or more sets of preordered items. Ordered information makes reports more meaningful and suggests critical relationships. Searches for information are faster with ordered lists.

The purpose of Sort/Merge is to arrange records in the sequence you specify. You describe the records you want to sort or merge and how Sort/Merge is to order them.

Sort/Merge can:

- Sort or merge records from as many as 100 files with one call to Sort/Merge
- Sort character and noncharacter key types
- Read input records with variable-length (V), fixed-length (F), or trailing-character-delimited (T) record type.
- Read input records from sequential, indexed-sequential, or direct-access files. It can write output records to sequential or indexed-sequential files.
- Read input records from and write output records to memory areas, mass storage files, and magnetic tape files.
- Sort according to twelve predefined collating sequences, thirteen numeric formats, and one or more user-defined collating sequences.
- Sum fields in records that have equivalent key values.
- Use owncode routines to insert, substitute, modify, or delete records during Sort/Merge processing
- Be called from any language that matches the calling sequence although some restrictions may apply (described later)

Merge capabilities are more restricted than sort capabilities. Merge input records cannot be supplied by owncode routines. Records to be merged must be presorted. Records to be merged and summed must be pre-sorted and pre-summed.

FORTRAN sorts are initiated with the SM5SORT procedure call and merges are initiated with the SM5MERG procedure call. You specify processing requirements for the sort or merge with various procedure calls.

Sort Keys

Sort or merge operations are based on the ordering of fields assigned to the data to be sorted or merged. These fields are called sort keys. This section discusses what sort keys are and how a key is defined.

A sort key is a field of data within each input record. Sort/Merge uses the contents of the sort key to determine the position of the record within the sorted sequence of records.

Data must be aligned correctly in a sort key field. Character data must be left-justified in the field, and numeric data must be right-justified in the field.

Multiple Keys

A file can be sorted on more than one sort key. The combined length of all key fields in a record cannot exceed 1023 bytes.

The first key you specify is the most important key and is called the major sort key. This key is sorted or merged first. The keys you specify after the first key are of lesser importance and are called minor sort keys. The minor keys are numbered in the order they are specified.

For example, if three sort keys are specified, the first key is the major sort key (key number 1), the next key listed is a minor key (key number 2), and the third key is another minor key (key number 3).

When two or more records have an equal major key, Sort/Merge determines the order by looking at the subsequent minor keys in the following order: key number 2, key number 3, and so on. Sort/Merge compares the minor keys until either an unequal key is found, or until there are no more keys.

For example, university student records could be sorted using multiple sort keys. Assume each record includes the last name and first and middle initials, the student number, the date of birth, the field of study, the grade point average, and a code representing class (freshman, sophpmore, junior, senior); all the fields are written with character data. The file could be maintained with the student number as the major key since records are normally retrieved by specifying the student number. The file can be sorted by the name in alphabetic order when a list of student names is needed.

When a university department needs to know which students are majoring in fields within the department, the file can be sorted on the field of study. The same sort can specify the name as a minor key so that records with the same field of study are also sorted in alphabetic order by the name. The file can be sorted by the class code as the major key and by the grade point average in descending numeric order as a minor key. This would produce a list of students sorted by class code with the students having the highest grade point average at the beginning of the list.

Defining Sort Keys

You must describe to Sort/Merge every field of data that you want used as a sort key. Sort key descriptions include the following information:

- Starting location of the key within the record
- Key length
- Type of data in the key field
- Sort order

You can define sort keys with SM5KEY procedure calls. The options and assumed values for describing sort keys are discussed in the following paragraphs.

Key Length and Position

You define key field length and position by specifying the first byte of the field and either the number of bytes in the field (length of the field) or the last byte of the field. The leftmost byte in a record is counted as number 1. For character data, each character is 8 bits and occupies 1 byte. For example, if you want to specify the name of the university student file as a sort key, and the name field is the leftmost field in the record, you specify the first byte as 1. If the name field is 20 characters long, you specify the length as 20.

Sort/Merge interprets the integers you specify for key length and position as bit numbers when the key type (discussed later in this chapter) specifies bits; otherwise, byte numbers are assumed. The first bit is numbered 1. Table 12-1 lists the maximum key field lengths for each key type. Sort/Merge allows key fields to overlap other key fields, except for the following:

- Key fields that are ordered by collating sequences defined with the alter option cannot overlap other key fields.
- Key fields cannot overlap sum fields.

Table 12-1. Maximum Key Field Sizes

Key Type	Maximum Size (in Bytes)	Key Type	Maximum Size (in Bytes)
Character	1023	BINARY	8
NUMERIC_FS	1023	BINARY_BITS	8184 (bits)
NUMERIC_LO	38	INTEGER	8
NUMERIC_LS	38	INTEGER_BITS	8184 (bits)
NUMERIC_NS	38	PACKED	19
NUMERIC_TO	38	PACKED_NS	19
NUMERIC_TS	38	REAL	8 or 16

Key Type

You specify the type of data in a key field with the name of a collating sequence or with the name of a numeric data format. The data in a key field can be character or noncharacter. Character data is represented in the computer as ASCII code values. To indicate the key type for character data, you specify the name of a collating sequence; for numeric character data, you specify the name of a numeric data format. Noncharacter data is represented in the computer as binary values, in packed decimal format, or in floating-point format.

The difference between the internal representation of character and noncharacter data is shown in figure 12-1.

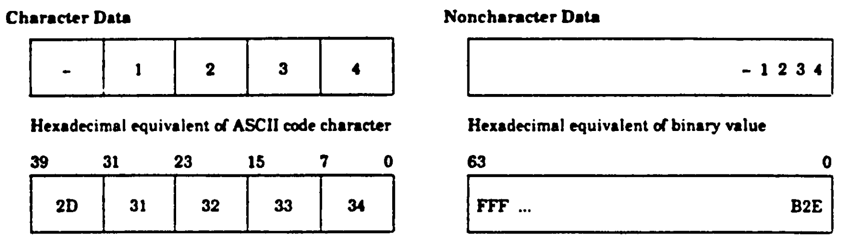


Figure 12-1. Internal Data Representation

If a sort key field contains any characters that are not meaningful for the key type you specify (an alphabetic character in a field defined as a numeric key, for example), the key field is considered to contain invalid data and so the record is invalid. The processing of invalid records is described later in this chapter.

The collating sequences and numeric data formats you can specify are discussed in the following paragraphs.

Collating Sequences

A collating sequence determines the precedence given to each character in relation to the other characters. You use a collating sequence for character data to determine the sort order. Character data must be in ASCII code characters.

Twelve predefined collating sequences are available to you as a Sort/Merge user. Six of the twelve predefined collating sequences are: ASCII, ASCII6, COBOL6, DISPLAY, EBCDIC, and EBCDIC6. If you do not specify a collating sequence, ASCII code is used. The predefined collating sequences are listed in appendix J.

For Better Performance

Sort/Merge sorts fastest when using the ASCII collating sequence.

Numeric Data Formats

Numeric data can appear in a key field in one of the formats listed in table 12-2.

For Better Performance

For numeric data, the most efficient numeric data formats are INTEGER, BINARY, and REAL.

Except for the BINARY_BITS and INTEGER_BITS formats, each field must start and stop on character (byte) boundaries.

Numeric data can be signed or unsigned. For character numeric data that is signed, the sign can be a floating sign, an overpunch representation over the leading (leftmost) digit, a leading separate character, an overpunch representation over the trailing (rightmost) digit, or a trailing separate character.

Table 12-2. Numeric Data Formats

Name	Data Type	Sign	Comments
BINARY	Binary integer	None	The field starts and ends on character boundaries. Data is ordered according to numeric value.
BINARY_ BITS	Binary integer	None	The field does not start or end on character boundaries. Data is ordered according to numeric value.
INTEGER	Two's complement binary integer	Positive if leftmost bit is 0; negative if leftmost bit is 1	The field starts and ends on character boundaries. Data is ordered according to numeric value.
INTEGER_ BITS	Two's complement binary integer	Positive if leftmost bit is 0; negative if leftmost bit is 1	The field does not start or end on character boundaries. Data is ordered according to numeric value.
NUMERIC_ FS	Leading blanks, numeric characters	– sign for negative values; a + character is not allowed	The field contains leading blanks (leading zeros must be converted to blanks before calling Sort/Merge); if the value is negative, the rightmost leading blank must be converted to a minus sign. If the field contains no leading blanks or does not begin with a negative sign, the value must be positive. This format is equivalent to the FORTRAN I format, or the COBOL picture clause for zero suppressed editing of numeric item. Data is ordered according to numeric value.

(Continued)

Table 12-2. Numeric Data Formats *(Continued)*

Name	Data Type	Sign	Comments
NUMERIC_ LO	Numeric characters	Leading overpunch	All characters are decimal digits except the leading character, which indicates a sign by an overpunch. Data is ordered according to numeric value with all forms of zero ordered equally.
NUMERIC_ LS	Numeric characters	Leading separate	All characters are decimal digits except the leading character, which is a negative or positive sign. Specifying a field that is not at least two characters in length causes a fatal error. Data is ordered according to numeric value with all forms of zero ordered equally.
NUMERIC_ NS	Numeric characters	None	All characters are decimal digits. Data is ordered according to numeric value.
NUMERIC_ TO	Numeric characters	Trailing overpunch	All characters are decimal digits except the trailing character, which indicates a sign by an overpunch. Data is ordered according to numeric value with all forms of zero ordered equally.
NUMERIC_ TS	Numeric characters	Trailing separate	All characters are decimal digits except the trailing character, which is a negative or positive sign. Specifying a field that is not at least two characters in length causes a fatal error. Data is ordered according to numeric value with all forms of zero ordered equally.

(Continued)

Table 12-2. **Numeric Data Formats** *(Continued)*

Name	Data Type	Sign	Comments
PACKED	Packed decimal	Signed	Data is ordered according to numeric value.
PACKED_NS	Unsigned packed decimal	Unsigned	Data is ordered according to numeric value. PACKED_NS is the same as COBOL COMPUTATIONAL-3 with no sign.
REAL	Normalized floating-point number, either single-precision (8 bytes) or double-precision (16 bytes)	Signed	All forms of zero are ordered equally. The order of indefinite values is undefined. The order of infinite values is ordered as if its value were infinity (can be signed infinity).

A floating sign is a negative sign embedded between leading blanks and the numeric characters. A floating sign can also be a negative sign followed by numeric characters. Leading zeros must be converted to blanks. Positive values in this format are not signed. The following examples are valid floating sign formats:

```
ΔΔ-1
ΔΔΔ1
ΔΔ-0
ΔΔΔ0
-123
1234
```

The following examples are invalid floating sign formats:

ΔΔ01	Leading zero not allowed
Δ-01	Leading zero not allowed
+123	Positive sign not allowed
ΔΔΔΔ	All-blank field not allowed

Diagnostic messages are issued for invalid floating sign formats or invalid overpunches.

A negative sign overpunch is equivalent to overstriking a digit with a – , which is a punch in row 11 of a punched card. A positive sign overpunch is equivalent to overstriking a digit with a + , which is a punch in row 12 of a punched card. When a signed overpunch digit is received as input, the digit is punched as indicated in the second column of table 12-3. When a signed overpunch digit is entered from a terminal or displayed as output, the digit appears as indicated in the third column of table 12-3. The hexadecimal value is in the fourth column.

Table 12-3. Sign Overpunch Representation

Sign and Digit	Input Punch	Input/Output Representation	Hexadecimal Value
+0	0	0	30
+1	1	1	31
+2	2	2	32
+3	3	3	33
+4	4	4	34
+5	5	5	35
+6	6	6	36
+7	7	7	37
+8	8	8	38
+9	9	9	39
+0	12–0	{	7B
+1	12–1	A	41
+2	12–2	B	42
+3	12–3	C	43
+4	12–4	D	44
+5	12–5	E	45
+6	12–6	F	46
+7	12–7	G	47
+8	12–8	H	48
+9	12–9	I	49
-0	11–0	}	7D
-1	11–1	J	4A
-2	11–2	K	4B
-3	11–3	L	4C
-4	11–4	M	4D
-5	11–5	N	4E
-6	11–6	O	4F
-7	11–7	P	50
-8	11–8	Q	51
-9	11–9	R	52
+0	12–8–4	<	3C
+0	12	&	26
-0	12–8–7	!	21
-0	11	-	2D

Sort Order

Sort/Merge can sort a key in ascending or descending order. If you do not specify a sort order, Sort/Merge sorts the key in ascending order.

When sorting a numeric key in ascending order, Sort/Merge sorts the key values in numeric order from least to greatest. When sorting a numeric key in descending order, Sort/Merge sorts the key values in numeric order from greatest to least.

A character key is sorted according to the collating sequence you specify for the key. When sorting a character key in descending order, Sort/Merge sorts the key values in reverse order of the collating sequence you specify.

Specifying the Record Length

Sort/Merge can sort records up to 65,535 bytes long. Sort/Merge determines the maximum and minimum record lengths for a file by its MAXIMUM_RECORD_LENGTH and MINIMUM_RECORD_LENGTH file attributes. The record length attributes are set when the file is created

The default sort key begins with the first byte in the record and extends to the smallest minimum record length value for all input files. If the minimum MINIMUM_RECORD_LENGTH attribute for all input files is 0, Sort/Merge uses 1 as the key length. If the minimum MINIMUM_RECORD_LENGTH attribute for all input files is greater than 1023, Sort/Merge uses 1023 as the key length.

When the Sort/Merge specification specifies an owncode 1 procedure and an owncode 3 procedure, but no input or output file, Sort/Merge expects all input records to be provided by the owncode 1 procedure and all output processing to be performed by the owncode 3 procedure. In this case you must specify the record length SM50FL or SM50MRL call.

Short Records

A short record is a record that does not contain all the key and sum fields defined for the sort or merge. Sort/Merge determines that a record is short when it reads the record from the input source.

NOTE

Records can become short when the system strips off all trailing blanks from variable-length (V) records. For example, when a variable-length (V) record containing all spaces is displayed by the SCL command DISPLAY_FILE, the spaces are stripped from the record, leaving a zero-length record.

When Sort/Merge attempts to use a field in a record and finds that the field is entirely beyond the end of the record, it uses a default value for the field. For character keys, the default value is all spaces. For numeric keys and sum fields, the default value is zero in the appropriate format.

Sort/Merge uses the default value only when using the key value or the sum field value. It does not pass the default value to an owncode procedure or store it in the output record.

Sort/Merge processing differs when the field it attempts to use is only partially beyond the end of the record. If the partial field is a character key field, Sort/Merge pads it with spaces, but if the partial field is a numeric key field or a sum field, Sort/Merge processes it as an exception.

Exception Processing for Partial Numeric Key Sum Fields

Exception processing for partial numeric key fields is as follows:

1. The record is written to the exception records file if one is specified for the sort or merge.
2. If an exception records file exists, the record is removed from the sort or merge; otherwise, its order is left undefined.
3. The count of partial numeric key fields or sum fields is incremented. A warning error message gives the count at the end of the sort or merge.

Exception Processing for Partial Sum Fields

Exception processing for partial sum fields differs if an exception records file is specified:

1. If an exception records file is specified:
   a. Sort/Merge writes the record with the partial sum field to the exception records file. It writes the record with its original data as it was read from the input source.
   b. It then removes the record from the sort or merge.
2. If an exception records file is not specified:
   a. Sort/Merge keeps the record with the partial sum field in the sort or merge.
   b. Later, if Sort/Merge finds other records whose key values are equivalent to the record, it sums the records as if the partial sum field contains a valid value; it does not process the partial sum field as invalid data. However, because the results of summing with a partial field are undefined, the resulting contents of the sum field are undefined.

If Sort/Merge reads any records with partial sum fields, it returns a summary diagnostic at the end of the sort or merge, giving the number of records with partial sum fields.

Zero-Length Records

A zero-length record is a record that contains no data and so its record length is 0. The processing of zero-length records read from input files depends on the SM5ZLR call in the Sort/Merge specification.

By default, if the SM5ZLR call is omitted, Sort/Merge deletes all zero-length records from the sort or merge. This is the DELETE option.

However, instead of a DELETE specification, SM5ZLR can specify one of these processing options for zero-length records:

PAD

Assign default values to key fields and sum fields in zero-length records (as it would short records) and keep the zero-length records in the sort or merge.

LAST

Write all zero-length records at the end of the output file or memory area.

Zero-length records are never written to the exception records file if the DELETE option is selected. Zero-length records are written to the exception records file if the PAD option is selected and either of the following situations exist:

- If merge order verification is requested and the input files contain zero-length records which are not pre-sorted on the merge keys.
- If AMP$PUT_NEXT detects an error while writing a zero-length record. (In general, attempts to write zero-length records to an indexed-sequential file cause errors.)

If duplicate records are to be omitted (as specified by an SM5OMIT call) and the PAD option is specified for zero-length records, only one zero-length record is included in the sort or merge.

Zero-length records are passed to owncode procedures only if the PAD option is selected. When passing a zero-length record to an owncode procedure, Sort/Merge passes an empty array of the maximum record length and the record length parameter set to zero.

The counts kept in the result array for the sort or merge may differ depending on the SM5ZLR specification:

Word 2, number of records read

Zero-length records are always included in the count.

Word 6, number of records sorted or merged

Zero-length records are included only if the PAD option is selected.

Words 13, 14, and 15; number of records written, the minimum record length, and the average record length

Zero-length records are included in the computation of these values only if the PAD or LAST option is selected.

Word 17, the number of zero-length records deleted from the sort or merge

This count is kept only if the DELETE option is selected.

Invalid Records

Sort/Merge checks that all key fields contain data that is valid for the key type. It determines whether a sum field contains valid data only when it attempts to use the data. It does not validate any fields other than key fields and sum fields.

A record can also be determined to be invalid when it is written. Sort/Merge writes records to the output file using the system procedure AMP$PUT_NEXT. A record is considered invalid if AMP$PUT_NEXT returns an error when it attempts to write the record. For example, when writing an indexed-sequential file, AMP$PUT_NEXT returns an error if the primary-key value for the record is already in the file.

Invalid records are processed as exceptions. The processing performed depends on whether the invalid data is in a key field or a sum field.

Exception Processing for Invalid Key Data

A warning error is issued if a key field contains invalid data. The warning error results in the following actions:

1. The record is written to the exception records file if an exception records file was specified.

2. The record is deleted from the sort or merge if an exception file was specified. If an exception records file was not specified, the record remains in the sort or merge, but its place in the sort order is undefined.

3. A diagnostic message is issued, as controlled by the list options specification.

4. The sort or merge continues normally.

Exception Processing for Summing Errors

Sort/Merge detects summing errors only when it attempts to sum fields. Only one error is detected per sum field. The summing error is processed as an exception. If the LIST_OPTIONS requests detailed error reporting (DE), Sort/Merge issues a diagnostic for each summing error.

The exception processing performed for summing errors depends on the error detected and on whether an exception records file is specified for the sort or merge.

If an exception records file is specified:

1. Sort/Merge restores all sum fields of both records so their contents is the same as it was before summing of the two records began.

2. If the error is due to invalid data or an indefinite real, Sort/Merge knows that at least one of the sum fields in the record is in error; it does not know if the same sum fields in the other record is also in error. Therefore, it writes the record it knows to be in error to the exception records file and removes it from the sort or merge, but it leaves the other record in the sort or merge.

3. If Sort/Merge detects an arithmetic overflow or underflow error or finds that each record has invalid data in different sum fields, it knows that both records are in error. Therefore, it writes both records to the exception records file and removes both from the sort or merge.

If an exception records file is not specified:

1. Sort/Merge deletes one of the records. If one record is longer than the other, the shorter is deleted. Otherwise, either record could be deleted.

2. The other record remains in the sort or merge with undefined data in the sum field for which the error was detected. Summing is completed for the other sum fields.

Performance Considerations

To improve the performance of Sort/Merge in your programs, consider the following:

- Do not use owncode procedures except when necessary. Allow Sort/Merge to read the input records from files and write the output records to a file.
- Ensure that all key fields and sum fields are within the minimum record length for all input records. Additional processing is required for short records.
- If possible, use a fixed record length instead of a variable record length.
- Sort/Merge sorts fastest when using the ASCII collating sequence. For numeric data, the most efficient numeric data formats are INTEGER, BINARY, and REAL.
- Sort/Merge can read and write files faster if the files use the following default attributes:
  - Sequential file organization
  - F or V record type
  - System-specified blocking
  - No error-exit procedure
  - No file access procedure (FAP)
  - The padding character is space
- Use the optimum page_aging interval for your sort, as described under Page_Aging_Interval in this chapter.

Sort/Merge also executes faster when your site uses a larger page size.

Limiting Memory Usage

By default, Sort/Merge limits the memory assigned to its sorting array to 262,144 (256K) bytes. However, you can change this limit by defining an SCL integer variable named SMV$MEMORY_USAGE_LIMIT. The integer you assign to the variable is used as the memory usage limit for subsequent sorts within the scope of the variable.

NOTE

The SMV$MEMORY_USAGE_LIMIT value is not used to limit memory usage for merges; it is used only for sorts (including the internal merge performed as part of a sort).

The integer assigned to the SMV$MEMORY_USAGE_LIMIT variable is the memory limit in 1024-byte (1K) units.

The minimum limit is 64. If you specify an integer less than 64, Sort/Merge uses the minimum limit of 64.

The maximum limit is 16,383. If you specify an integer greater than 16,383, Sort/Merge uses the maximum limit of 16,383.

A warning error is issued when you specify a value outside the range from 64 to 16,383.

As an example of creating the variable, the SCL command CREATE_VARIABLE is used to create the SMV$MEMORY_USAGE_LIMIT variable and assign it the value 64.

```
call sclc create_variable, SMV$MEMORY_USAGE_LIMIT,
  'CAT 'kind=integer, value=64, scope=local);
```

Page_Aging_Interval

Page_aging_interval is the job attribute that controls how quickly pages are aged from the working set of a task. If you increase the memory usage limit for your sorts using the SMV$MEMORY_USAGE_LIMIT variable, you should also increase your page_aging_interval value.

The optimum page_aging_interval depends on the CYBER 180 model you use. A smaller value is appropriate for the faster models. For example, when the default memory usage limit of 256 pages is used, the optimum page_aging_interval for a CYBER 180/830 is about 500,000 microseconds, while, for a CYBER 180/860, the optimum value is about 100,000 microseconds.

To see your current page_aging_interval attribute value, enter the following SCL command:

```
display_job_attribute, display_option=page_aging_interval
```

To change your page_aging_interval value, use the CHANGE_JOB_ATTRIBUTE command. For example, the following command changes the page_aging_interval to 500,000 microseconds:

```
change_job_attribute, page_aging_interval=500000
```

Sort/Merge Procedure Calls

FORTRAN Sort/Merge procedure calls must follow the same coding rules as other FORTRAN call statements. The Sort/Merge calls can be used by other languages that use the standard calling sequence.

NOTE

When a program written in a language other than FORTRAN or COBOL uses Sort/Merge calls described in this chapter, you must add the following object library to the program library list before executing the program:

$LOCAL.SMF$LIBRARY

If the Sort/Merge calls are compiled within the FORTRAN program unit, you do not need to add this library.

The system attaches the file when you login, but you must add it to your library list so that modules can be loaded from the file.

For example, the following SET_PROGRAM_ATTRIBUTE command adds the object library to the program library list:

```
set_program_attribute, add_library=$local.smf$library
```

To read more about the program library list, see the SCL Object Code Management manual.

The procedures can be called in any order with two exceptions: SM5SORT, SM5MERG, must be the first procedures called, and SM5END must be the last procedure called. Sort/Merge collects processing information until SM5END is called; the sort or merge is then performed.

Sort/Merge requires a value for the maximum record length for all procedure calls. You must specify a value for MAXRL on the SET_FILE_ATTRIBUTE command if the system default (MAXRL=256) is too small for your files.

Unless otherwise stated, the file characteristics are block type SYSTEM_SPECIFIED (BT=SS) and record type VARIABLE (RT=V). For files with other block types and record types, you must execute the SET_FILE_ATTRIBUTE command before the file is created and before the sort or merge.

Input files are named by the SM5FROM procedure; output files are named by the SM5TO procedure. You can enter the Sort/Merge parameter and user-defined values in uppercase, lowercase, or a combination, because Sort/Merge treats lowercase letters as being equal to uppercase letters. Owncode routine names must be specified in all uppercase letters.

If you specify an owncode procedure name in lowercase letters, Sort/Merge does not convert the name to uppercase letters, unless you specify the true option on the SM5CC procedure.

Unless stated otherwise, a procedure can be called only once during a sort or merge.

SM5CC

Purpose Specifies whether lowercase letters in owncode procedure names are to be converted to uppercase letters.

Format CALL SM5CC(opt)

Parameters opt

Required; string containing one of these values:

TRUE, T, YES, Y, ON, true, t, yes, y, on

Sort/Merge converts any lowercase letters in owncode procedure names to uppercase letters.

FALSE, F, NO, N, OFF, false, f, no, n, off

Sort/Merge does not convert lowercase letters in owncode procedure names.

If the SM5CC call is omitted, lowercase letters in owncode procedure names are not converted.

Remarks When Sort/Merge attempts to load an owncode procedure, it passes the procedure name as you have specified it on the SM5OWNn call. If you specify the name with lowercase letters, Sort/Merge passes the lowercase letters unless an SM5CC call requests conversion.

The system stores entry point names using uppercase letters only. Therefore, if the loader is given a procedure name containing lowercase letters, it cannot find that name in the program library list and so it cannot to load the requested procedure.

SM5DUCT

Purpose Specifies a user-defined collation table.

Format CALL SM5DUCT (ktype, collating_table_name)

Parameters ktype

Required; name you choose to call the collating sequence defined by the weight table. It is specified as the key type on the SM5KEY calls that use this collating sequence.

collating_table_name

Required; name of the 256-character string containing the collating weights.

Remarks

- Sort/Merge does not distinguish between lowercase and uppercase letters in the specified names.
- A Sort/Merge call sequence can include more than one SM5DUCT call.
- The total number of SM5SEQN, SM5LCT, and SM5DUCT calls in a Sort/Merge call sequence cannot exceed 100.
- The name SM5DUCT assigns to the collating sequence cannot be the name of a predefined collating sequence or another collating sequence already defined for the sort or merge.
- The weight table must already be loaded as part of the program. It must be a string declared by CHARACTER USER*256. Each character specifies the collating weight of the corresponding ASCII character.
- For more information, see the Collation Tables appendix in this manual.

SM5E

Purpose Specifies the file to which diagnostic messages for this sort or merge are written.

Format CALL SM5E (file)

Parameters file

Required; character expression specifying the name of the file to receive diagnostic messages.

If SM5E call is omitted, error messages are written to file $ERRORS.

Remarks

- Sort/Merge writes the error file only if it detects errors of at least the severity specified by the SM5EL call.
- Sort/Merge does not rewind the error file before or after it uses it.
- If you specify $NULL as the error file, diagnostic messages are not written.
- If you specify the same file as the listing file and as the error file (SM5E and SM5LIST), each diagnostic message is written only once to the file. (Otherwise, each message is written twice, once to the error file and once to the listing file.)
- The error level reported to the error file is specified by the SM5EL call.

SM5EL

Purpose Specifies the minimum severity level to be reported on the error file.

Format CALL SM5EL (lim)

Parameters lim

Required; character expression specifying the severity level of errors to be written to the error file:

I or i

All informational, warning, fatal, and catastrophic errors.

T or t

Same as informational; (obsolete value; its use is not recommended).

W or w

All warning, fatal, and catastrophic errors.

F or f

All fatal and catastrophic errors.

C or c

Only catastrophic errors.

NONE or none

No errors are written to the error file.

If the SM5EL call is omitted, all diagnostics are reported regardless of severity.

Remarks The error file is specified by the SM5E call.

SM5END

Purpose Terminates a sort or merge specification and initiates Sort/Merge processing.

Format CALL SM5END

Remarks The SM5END call is required. It must be the last in the sequence of Sort/Merge calls.

SM5ENR

Purpose Allows compatibility with NOS Sort/Merge 5. NOS/VE does not use the specified value.

Format CALL SM5ENR (value)

Parameters value

Required; an integer value indicating the estimated number of records to be sorted. The value can be from 1 through 16,777,215.

SM5ERF

Purpose Specifies the file to which invalid records are written.

Format CALL SM5ERF (file)

Parameters file

Required; character expression specifying the file name of the exception records file.

If the SM5ERF call is omitted, exception records are not removed from the sort or merge. The order of records with invalid keys is undefined. The contents of sum fields for which summing errors are detected is also undefined.

Remarks

- The exception records file cannot also be the output file or an input file. Its file organization must be sequential; it cannot be a keyed file.
- If you specify $NULL as the exception records file, each exception record is deleted as it is written to the file.
- All records written to the exception records file are deleted from the sort or merge.
- The records written to the exception records file include:
  - Records containing invalid key data.
  - Records containing invalid sum data if Sort/Merge attempts to sum the data.
  - Records that caused an arithmetic overflow or underflow when their sum fields were summed.
  - Short records in which Sort/Merge found a partial numeric key field or partial sum field.
  - Out-of-order merge input records if merge order checking was requested by an SM5VER call.
  - Records for which the system procedure AMP$PUT_NEXT returned an error when it attempted to write the record to the output file for the sort or merge.
- A summary of the records written to the exception records file is written to the errors file and to the list file.

SM5FMA

Purpose Specifies a memory area to be read as a source of input records.

Format CALL SM5FMA (variable, 'FIXED', max_record_length, number_of_records)

Parameters variable

Required; name of the memory location at which Sort/Merge begins reading input records.

'FIXED'

Required; string expression (FIXED or fixed) specifying that each input record read from the memory area is the fixed length specified by the third parameter on the call.

max_record_length

Required; integer giving the fixed record length in bytes. The maximum input record size is 65,536.

number_of_records

Required; integer giving the number of records Sort/Merge is to read from the memory area.

If the SM5FMA call is omitted, all input records are read from files or supplied by owncode procedures.

Remarks

- A Sort/Merge specification can specify up to 100 sources of input records. These sources can be files or memory area; the sources are read in the order you specify them. Files are specified by SM5FROM calls; memory areas are specified by SM5FMA calls.
- When a memory area is used as an input record source, a sort cannot use an owncode 1 or owncode 2 procedure.
- The record order is undefined when a memory area specified by an SM5FMA call overlaps the memory area specified by the SM5TMA call.

SM5FROM

Purpose Specifies one or more files from which input records are read.

Format CALL SM5FROM (file, ..., file)

Parameters file

Required; character expression specifying the name of an input file. The files are read in the order specified on the call.

If the SM5FMA call is omitted, input records are read from the specified memory area. Or, if SM5OWN1 is called, input records could be supplied by the owncode 1 procedure. Otherwise, Sort/Merge attempts to open and read file $LOCAL.OLD as the source of input records.

Remarks

- A Sort/Merge specification can specify up to 100 sources of input records. These sources can be files or memory areas; the sources are read in the order you specify them. Files are specified by SM5FROM calls; memory areas are specified by SM5FMA calls.
- All instances of open of the input files must be closed before the sort or merge begins. Sort/Merge opens each file before it reads it and closes it when it has finished reading it.
- Sort/Merge does not read past an end-of-partition delimiter embedded in an input file.
- The input files for a merge must be pre-sorted on the same keys used for the merge. For a merge with summing, the input files must also be pre-summed using the same sum fields specified for the merge.
- A Sort/Merge input file can reside on either mass storage or magnetic tape.
- The Sort/Merge output file can have sequential, direct-access, or indexed-sequential file organization and its record type can be variable (V), fixed-length (F), or trailing-character-delimited (T).

SM5KEY

Purpose Specifies a key field to be used by the sort or merge.

Format CALL SM5KEY (first, len, ktype, ad)

Parameters first

Required; integer expression specifying the first position of the key field. Bit positions are used for the BINARY_BITS and INTEGER_BITS key types, byte positions for all others. Positions are numbered from the left beginning with 1.

len

Required; integer expression specifying the number of positions in the key field. The number of bits are given for the BINARY_BITS and INTEGER_BITS key types, byte positions for all others.

To see the maximum key field sizes, see table 12-1.

ktype

Required; character expression specifying the numeric data format or, for character data, the collating sequence.

You can define a collating sequence name with an SM5DUCT, SM5LCT, or SM5SEQN call or use one of the following collating sequences without predefinition:

ASCII

ASCII collating sequence.

ASCII6

OSV$ASCII6_FOLDED collating sequence.

COBOL6

OSV$COBOL6_FOLDED collating sequence.

DISPLAY

OSV$DISPLAY64_FOLDED collating sequence.

EBCDIC

OSV$EBCDIC collating sequence.

EBCDIC6

OSV$EBCDIC6_FOLDED collating sequence.

Appendix J lists the predefined collating sequence.

The following are the available numeric data formats:

BINARY

Binary integer starting and ending on byte boundaries.

BINARY_BITS

Binary integer not required to start or end on byte boundaries.

INTEGER

Two's complement binary integer starting and ending on byte boundaries.

INTEGER_BITS

Two's complement binary integer not required to start or end on byte boundaries.

NUMERIC_FS

Numeric characters with floating sign (FORTRAN I format or COBOL zero-suppressed editing item).

NUMERIC_LO

Numeric characters with leading overpunch sign.

NUMERIC_LS

Numeric characters with leading separate sign.

NUMERIC_NS

Numeric characters with no sign.

NUMERIC_TO

Numeric characters with trailing overpunch sign.

NUMERIC_TS

Numeric characters with trailing separate sign.

PACKED

Signed packed decimal.

PACKED_NS

Unsigned packed decimal.

REAL

Normalized floating-point number, single-precision (8 bytes) or double-precision (16 bytes).

ad

Required; character expression specifying the order of the sort or merge operation:

A or a

Ascending order

D or d

Descending order

If the SM5KEY call is omitted, the only key field used begins at position 1 and extends through the smallest minimum record length of the input sources. However, the minimum key length used is 1 and the maximum key length used is 1023.

The key is sorted by the ASCII collating sequence in ascending order.

Remarks

- Sort/Merge treats lowercase letters in parameter values as being equal to uppercase letters.
- The combined length of all key fields defined for a sort or merge cannot exceed 1023 bytes.
- The total number of SM5KEY calls in a Sort/Merge call sequence cannot exceed 106.
- The significance of multiple keys corresponds to the order in which the keys are defined.
- Sort key fields can overlap other sort key fields with the following exceptions:
  - Key fields that are ordered by collating sequences defined with an SM5SEQA call cannot overlap other key fields.
  - Key fields cannot overlap sum fields.
- For more information, see the description of Short Records and Zero-Length Records earlier in this chapter.

SM5LCT

Purpose Loads a collation table, that is, a weight table that defines a collating sequence. The table may be a NOS/VE predefined collation table or a user-defined collation table in an object library.

Format CALL SM5LCT (ktype, collation_table_name)

Parameters ktype

Required; name you choose to call the collating sequence defined by the weight table. It is specified as the key type on the SM5KEY calls that use this collating sequence.

The name cannot be the name of a predefined collating sequence or the name of a collating sequence you have already defined.

collation_table_name

Required; name of a predefined weight table or an object library module defining a collating sequence.

Remarks

- Sort/Merge treats lowercase letters as being equal to uppercase letters.
- The total number of SM5DUCT, SM5LCT, and SM5SEQN calls in a Sort/Merge specification cannot exceed 100.
- The weight table must be loadable by PMP$LOAD and have 256 weight values.
- For more information, see the collation table appendix in this manual.

SM5LIST

Purpose Specifies the name of the list file.

Format CALL SM5LIST (file)

Parameters file

Required; character expression specifying the file name of the listing information file.

If the SM5LIST call is omitted, the default list file is $LIST.

Remarks

- Listing information includes the Sort/Merge version and level numbers, time and date, diagnostics, and statistics such as the number of records sorted or merged.
- If you specify the same file as the listing file and as the error file (SM5E and SM5LIST), each diagnostic message is written only once to the file. (Otherwise, each message is written twice, once to the error file and once to the listing file.)

SM5LO

Purpose Specifies the information written to the listing file.

Format CALL SM5LO (option)

Parameters option

Required; one of the following values:

OFF
All listing information is suppressed.

NONE
Same as OFF keyword.

DE
Detailed exception information (valid only if SM5ERF is called).

RS
Record statistics for those records sorted or merged.

MS
Merge statistics for the records merged.

S
Valid keyword, but meaningless on an SM5LO call.

Remarks

- The minimum information Sort/Merge writes to the listing file is the page heading, error messages, the exception file summary, and the number of records sorted or merged.
- You can specify only one option with each SM5LO call, but the Sort/Merge specification can include more than one SM5LO call.

SM5MERG

Purpose Signals the beginning of a sequence of Sort/Merge calls for a merge operation.

Format CALL SM5MERG (array)

Parameters array

Required; name of a one-dimensional array of 1 through 18 integers in which Sort/Merge returns statistics about the merge. Or, if you specify 0, Sort/Merge returns no statistics.

NOTE

The specified result array should be declared inside a common block. FORTRAN optimization requires that variables specified on a call, but modified after return from the call, occur only in common blocks.

Remarks

- SM5MERG must be the first routine called for a merge operation.
- In the first word of the array, you must specify the number of values (0 through 17) you want returned. Values are returned in words 2 through 18. The array must be long enough to contain the number of values you request in the first word.
- The result array format is listed in table 12-4.

Table 12-4. Result Array Format

Array Element	Contents
1	Number of elements of results you want returned in the array (0 through 17)
2	Number of records read from input files or memory areas
3	Number of records deleted by an owncode 1 procedure
4	Number of records inserted by an owncode 1 procedure
5	Number of records inserted by an owncode 2 procedure
6	Number of records sorted or merged. The count does not include records written to the exception records file or zero-length records (unless the SM5ZLR call selects the PAD option.)
7	Number of records deleted by an owncode 3 procedure
8	Number of records inserted by an owncode 3 procedure
9	Number of records inserted by an owncode 4 procedure
10	Number of records written to the exception file
11	Number of records deleted by an owncode 5 procedure
12	Number of records combined by summing
13	Number of records written to the output file or memory area
14	Actual minimum record length of all input records
15	Average record length (total record length divided by the total number of input records)
16	Actual maximum record length of all input records
17	Number of zero-length records removed from the sort or merge because the default SM5ZLR option (DELETE) is selected.
18	Number of records with equivalent key values (duplicates) removed from the sort or merge as requested by an SM5OMIT call.

SM5OFL

Purpose Specifies the length of each fixed-length record entering the sort or merge from an owncode procedure.

Format CALL SM5OFL (flen)

Parameters flen

Required; integer expression specifying the fixed length in bytes. Valid values are from 1 through 65,535.

If SM5OWN1 and SM5OWN3 are called, but SM5FROM and SM5TO are not, an SM5OFL or SM5OMRL call is required. Otherwise, if SM5OFL and SM5OMRL are omitted, the record length is the largest MAXIMUM_RECORD_LENGTH attribute for the input and output files used by the sort.

Remarks

- A fatal error occurs if a owncode procedure supplies a record of any other length.
- You cannot call both SM5OFL and SM5OMRL for the same sort operation.

SM5OMIT

Purpose Specifies whether Sort/Merge outputs only one record in each set of records with equivalent key values.

Format SM5OMIT (option)

Parameters option

Required; one of the following character expressions:

TRUE, T, YES, Y, ON, true, t, yes, y, or on

Duplicates are omitted.

FALSE, F, NO, N, OFF, false, f, no, n, or off

Duplicates are not omitted.

If the SM5OMIT call is omitted, duplicates are not omitted. The processing of records with equivalent key values depends on whether SM5OWN5, SM5RETA, or SM5SUM is called. If all of these calls are omitted, records with equivalent key values remain in the sort or merge, but their relative order is undefined.

Remarks

- Each sort or merge can specify only one method of processing records with equivalent key values. Therefore, the SM5OMIT, SM5OWN5, SM5RETA, and SM5SUM calls are mutually exclusive.
- When duplicates are omitted, Sort/Merge removes the shorter duplicate records from the sort or merge. When the duplicates have the same length, any of the duplicates could be the one that is kept.
- A count is kept in word 18 of the result array of the number of duplicate records deleted from the sort or merge due to an SM5OMIT call. (The result array is specified on the SM5MERG or SM5SORT call.)
- Duplicates omitted by an SM5OMIT call are not written to the exception records file.
- Zero-length records are processed as duplicates only if the SM5ZLR call specifies the PAD option.

SM5OMRL

Purpose Specifies the maximum length of any record entering the sort or merge from an owncode procedure.

Format CALL SM5OMRL (mlen)

Parameters mlen

Required; integer expression specifying the maximum length in bytes.

If SM5OWN1 and SM5OWN3 are called, but SM5FROM and SM5TO are not, an SM5OFL or SM5OMRL call is required. Otherwise, if SM5OFL and SM5OMRL are omitted, the record length is the largest MAXIMUM_RECORD_LENGTH attribute for the input and output files used by the sort.

Remarks

- 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 owncode procedures have the same length, SM5OFL should be called instead of SM5OMRL.

SM5OWNn

Purpose Specifies a user-written (owncode) procedure to be executed each time a certain event occurs during the sort or merge.

Format

CALL SM5OWN1(name)	Specifies the name of the owncode 1 procedure executed each time a sort reads an input record.
CALL SM5OWN2(name)	Specifies the name of the owncode 2 procedure executed each time a sort finishes reading an input file.
CALL SM5OWN3(name)	Specifies the name of the owncode 3 procedure executed each time a sort or merge is ready to write an output record.
CALL SM5OWN4(name)	Specifies the name of the owncode 4 procedure executed each time a sort or merge finishes writing its output records.
CALL SM5OWN5(name)	Specifies the name of the owncode 5 procedure executed each time a sort or merge finds two records with equivalent key values.

Parameters name

Required; character expression specifying the name of an owncode procedure.

The name must be specified using all uppercase letters unless the sort or merge calls SM5CC with the true option.

Owncode procedures are executed only if they are specified.

Remarks

- Merge specifications cannot call SM5OWN1 or SM5OWN2.
- Sort/Merge specifications that call SM5FMA cannot call SM5OWN1 or SM5OWN2. Sort/Merge specifications that call SM5TMA cannot call SM5OWN3 or SM5OWN4.
- Each sort or merge can specify only one method of processing records with equivalent key values. Therefore, the SM5OMIT, SM5OWN5, SM5RETA, and SM5SUM calls are mutually exclusive.
- For further information about owncode procedures, see the discussion later in this chapter.

SM5RETA

Purpose Specifies whether input records having equal keys are to be output in the same order they are input.

Format CALL SM5RETA (opt)

Parameters opt

Optional; character expression having one of the following values:

YES

Records with equal keys retain their original order.

NO

Records with equal keys may not retain their original order.

If this argument is omitted (no argument list specified), the default is YES.

Remarks

- Each sort or merge can specify only one method of processing records with equivalent key values. Therefore, the SM5OMIT, SM5OWN5, SM5RETA, and SM5SUM calls are mutually exclusive.
- If you select the 'YES' option and specify more than one input source, the order in which you specify the input sources is the order in which records with equal keys will be written.
- Maintaining the original order of records with equal key values increases the required processing time because Sort/Merge must keep track of the input order.

SM5SEQA

Purpose Used with the SM5SEQS call to specify whether characters are altered in the output. If characters are altered, all characters in the value step specified by the preceding SM5SEQS call are output as the first character in the value step.

Format CALL SM5SEQA (opt)

Parameters opt

Required; character expression having one of the following values:

TRUE, T, YES, Y, ON, true, t, yes, y, or on
Alters the equated characters.

FALSE, F, NO, N, OFF, false, f, no, n, or off
Does not alter the equated characters.

If the SM5SEQA call is omitted, characters are not altered.

Remarks SM5SEQA is used in a sequence of calls that define a user-defined collating sequence. The other calls are SM5SEQN, SM5SEQS, and SM5SEQR.

Examples The sequence of calls below converts all commas and semicolons to spaces:

```
CALL SM5SEQN ('ALTERSQ')
CALL SM5SEQS (' ', ',', ';')
CALL SM5SEQA ('YES')
```

SM5SEQN

Purpose

Specifies the name of the collating sequence specified by the following SM5SEQS, SM5SEQR, and SM5SEQA calls.

Format

CALL SM5SEQN (name)

Parameters

name

Required; character expression specifying the name of the user-defined collating sequence.

Remarks

- The end of the collating sequence definition is indicated by any statement other than an SM5SEQS, SM5SEQR, and SM5SEQA call.
- The specified name cannot be the same as that of any predefined collating sequence or user-defined collating sequence that you have already defined for the sort or merge.
- The specified name is used as the key type on SM5KEY calls defining key fields to be ordered by the user-defined collating sequence.

Examples

This statement names a user-defined collating sequence:

```
CALL SM5SEQN ('MYSEQ')
```

This statement defines a key field that uses the user-defined collating sequence:

```
CALL SM5KEY(1, 10, 'MYSEQ', 'A')
```

SM5SEQR

Purpose Defines the position of the remainder value step in the collating sequence being defined. The remainder value step consists of all characters that have not been included in value steps defined by SM5SEQS calls.

Format CALL SM5SEQR (opt)

Parameters opt

Required; character expression having one of the following values:

TRUE, T, YES, Y, ON, true, t, yes, y, or on

The remainder value step is defined at this position.

FALSE, F, NO, N, OFF, false, f, no, n, or off

The remainder value step is not defined.

If the SM5SEQR call is omitted, the last value step in the collating sequence is defined as the remainder value step.

Remarks SM5SEQR is used in a sequence of calls that define a user-defined collating sequence. The other calls are SM5SEQN, SM5SEQS, and SM5SEQA.

Examples The sequence below defines a collating sequence with two value steps: all nondigits followed by all digits.

```
CALL SM5SEQN ('DIGITS')
CALL SM5SEQR ('YES')
CALL SM5SEQS ('0','1','2','3','4','5','6','7','8','9')
```

SM5SEQS

Purpose

Specifies a value step in the collating sequence being defined.

A value step consists of one or more characters that are to have the same collating weight in the sequence.

The first CALL SM5SEQS statement specifies the first value step, the second SM5SEQS statement specifies the second value step, and so on until the collating sequence is completely defined.

Format

CALL SM5SEQS (char, ..., char)

Parameters

char

Required; character expression specifying a character in the value step.

Remarks

SM5SEQS is used in a sequence of calls that define a user-defined collating sequence. The other calls are SM5SEQN, SM5SEQR, and SM5SEQA.

Examples

This statement defines a value step consisting of one character:

```
CALL SM5SEQS ('A')
```

This statement defines a value step consisting of several characters:

```
CALL SM5SEQS ('1', '2', '3', '4')
```

SM5SORT

Purpose Signals the beginning of a sequence of Sort/Merge calls for a sort operation.

Format CALL SM5SORT (array)

Parameters array

Required; name of an integer array in which Sort/Merge returns statistics about the merge. Or, if you specify 0, Sort/Merge returns no statistics.

NOTE

The specified result array should be declared inside a common block. FORTRAN optimization requires that variables specified on a call, but modified after return from the call, occur only in common blocks.

Remarks

- SM5SORT must be the first routine called for a sort operation.
- In the first word of the array, you must specify the number of values (0 through 17) you want returned. Values are returned in words 2 through 18. The array must be long enough to contain the number of values you request in the first word.
- To see the result array format, see table 12-4.

SM5ST

Purpose Returns the severity level of the most severe error encountered during the sort or merge operation.

Format CALL SM5ST (lev)

Parameters lev

Required; 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	Informational errors
20	Warning errors
30	Fatal errors
40	Catastrophic errors

SM5SUM

Purpose Specifies that summing is to be performed on the specified fields.

Format CALL SM5SUM (first, len, type, rep)

Parameters

first

Required; integer expression specifying the first byte or bit of the sum field (numbered from the left starting with 1).

len

Required; integer expression specifying the number of bytes or bits in the sum field.

type

Required; character expression specifying the numeric data format. The numeric data formats are listed in table 12-2.

rep

Required; integer greater than zero specifying the number of times the field repeats in the record.

Remarks

- Each sort or merge can specify only one method of processing records with equivalent key values. Therefore, the SM5OMIT, SM5OWN5, SM5RETA, and SM5SUM calls are mutually exclusive.
- Sum fields cannot overlap one another. Sum fields cannot overlap key fields.
- SM5SUM can be called up to 100 times for each sort or merge.
- If SM5SUM is called, Sort/Merge processes records with equivalent values by combining the records into one output record. The sum fields contain the sums of the values in the corresponding sum fields in the input records. The rest of he record is taken from the longest of the original input records.
- To read about exception processing for partial sum fields, see the discussion under short records in this chapter.

SM5TMA

Purpose Specifies a memory area to used as the destination of output records.

Format CALL SM5TMA (variable, 'FIXED', max_record_length)

Parameters variable

Required; name of the memory location at which Sort/Merge begins writing output records.

'FIXED'

Required; string expression (FIXED or fixed) specifying that each input record written to the memory area is the fixed length specified by the third parameter on the call.

max_record_length

Required; integer giving the fixed record length in bytes. The maximum input record size is 65,536.

If the SM5TMA call is omitted, all output records are written to an output file or processed by an owncode 3 procedure.

Remarks

- A Sort/Merge specification can specify only one destination for output records. The destination can be a file or a memory area, but not both. A file is specified by an SM5TO call; a memory area is specified by an SM5TMA call.
- When a memory area is used as the destination for output records, the sort or merge cannot use owncode 3 or owncode 4 procedures.
- The record order is undefined when a memory area specified by an SM5FMA call overlaps the memory area specified by the SM5TMA call.
- A count of the records written to the memory area is kept in word 13 of the result array. (The result array is specified on the SM5SORT or SM5MERG call.)

SM5TO

Purpose	Specifies the file to receive the sorted or merged output records.
Format	CALL SM5TO (file)

Parameters

file

Required; character expression specifying the name of the file.

If the SM5TMA call is omitted, output records are written to the specified memory area. Or, if SM5OWN3 is called, output records are processed by an owncode 3 procedure. Otherwise, Sort/Merge writes the output records to file $LOCAL.NEW.

Remarks

- The output file cannot also be an input file or the exception records file or the error file or the list file.
- The file must be closed when the sort or merge begins. Sort/Merge closes the file when it completes the sort or merge.
- The Sort/Merge output file can reside on either mass storage or magnetic tape.
- The Sort/Merge output file can have either sequential or indexed-sequential file organization and its record type can be variable (V), fixed-length (F), or trailing-character-delimited (T).
- The Sort/Merge output file cannot use the direct-access file organization.
- If the output file is an indexed-sequential file with a nonembedded primary key, the primary-key value is removed from the beginning of the record when it is written to the output file.

  The removed primary-key value is stored in the primary index of the file. The record data stored is shortened by key_length characters.
- If the output file is an indexed-sequential file, the major sort key must be the primary key defined for the output file.

  The indexed-sequential file organization requires that each primary-key value be unique. Therefore, the value in the major sort key field must be unique for each output record. This can be ensured by specifying the OMIT_DUPLICATES=YES parameter or using an owncode 5 procedure.

- If the output (TO) file is an indexed-sequential file, Sort/Merge checks the KEY_POSITION, KEY_LENGTH, and KEY_TYPE attributes:
  - If the major sort key position does not match the KEY_POSITION attribute value, Sort/Merge issues a fatal error and terminates.
  - If the major sort key length does not match the KEY_LENGTH attribute value, Sort/Merge issues a warning error and changes the major sort key length to match the primary key length.
  - If the major sort key type does not match the KEY_TYPE attribute value, Sort/Merge issues a warning error and changes the major sort key type if the KEY_TYPE value is UNCOLLATED or INTEGER. (It does not issue a warning or change the key type if the KEY_TYPE value is COLLATED.)
    - If the KEY_TYPE is UNCOLLATED, the major sort key type is changed to ASCII.
    - If the KEY_TYPE is INTEGER, the major sort key type is changed to INTEGER.

SM5VER

Purpose Specifies whether Sort/Merge checks that the input records to a merge are in sorted order.

Format CALL SM5VER (opt)

Parameters opt

Required; character expression having one of the following values:

TRUE, T, YES, Y, ON, true, t, yes, y, or on
The order of merge input records is verified.

FALSE, F, NO, N, OFF, false, f, no, n, or off
The order of merge input records is not verified.

If the SM5VER call is omitted, the order of merge input records is not verified. Out-of-order input records remain in the merge. Their order in the output file is undefined.

Remarks

- If merge order verification is requested and Sort/Merge finds an input record out of order, it issues a warning message.

  If an exception records file has been specified (SM5ERF), any out-of-order input records are written to the exception records file and then deleted from the merge.

- If you include an SM5VER call is a sort specification, Sort/Merge issues a warning message, but otherwise ignores the call.

SM5ZLR

Purpose Specifies the disposition of zero-length records.

NOTE

The SM5ZLR option applies only to records read from input files; it does not apply to records read from memory areas or supplied by owncode procedures.

Format CALL SM5ZLR (keyword)

Parameters keyword

Required; character expression specifying one of the following keywords:

DELETE

Each zero-length record is deleted from the sort or merge. (The deleted records are not written to the exception records file.)

PAD

Each zero-length record is processed as a short record. Key fields are assigned default values (spaces for character keys; zero for numeric keys).

LAST

Each zero-length record is written at the end of the output.

If the SM5ZLR call is omitted, each zero-length record is deleted from the sort or merge.

Remarks For more information about zero-length records, see the discussion earlier in this chapter.

Owncode Routines

You can write subprograms to insert, substitute, modify, or delete input and output records during Sort/Merge processing. Such a subprogram, called an owncode routine, is executed each time the sort or merge reaches a certain point in Sort/Merge processing. Figure 12-2 illustrates the points at which Sort/Merge can call owncode routines.

Sort/Merge passes a record to the owncode routine, which processes the record. When the record is returned to Sort/Merge from the owncode routine, Sort/Merge processes the record according to a code passed by the owncode routine.

Owncode routines can also supply the records to be sorted. When Sort/Merge is ready for a record, it calls the owncode routine, which then passes a record to Sort/Merge.

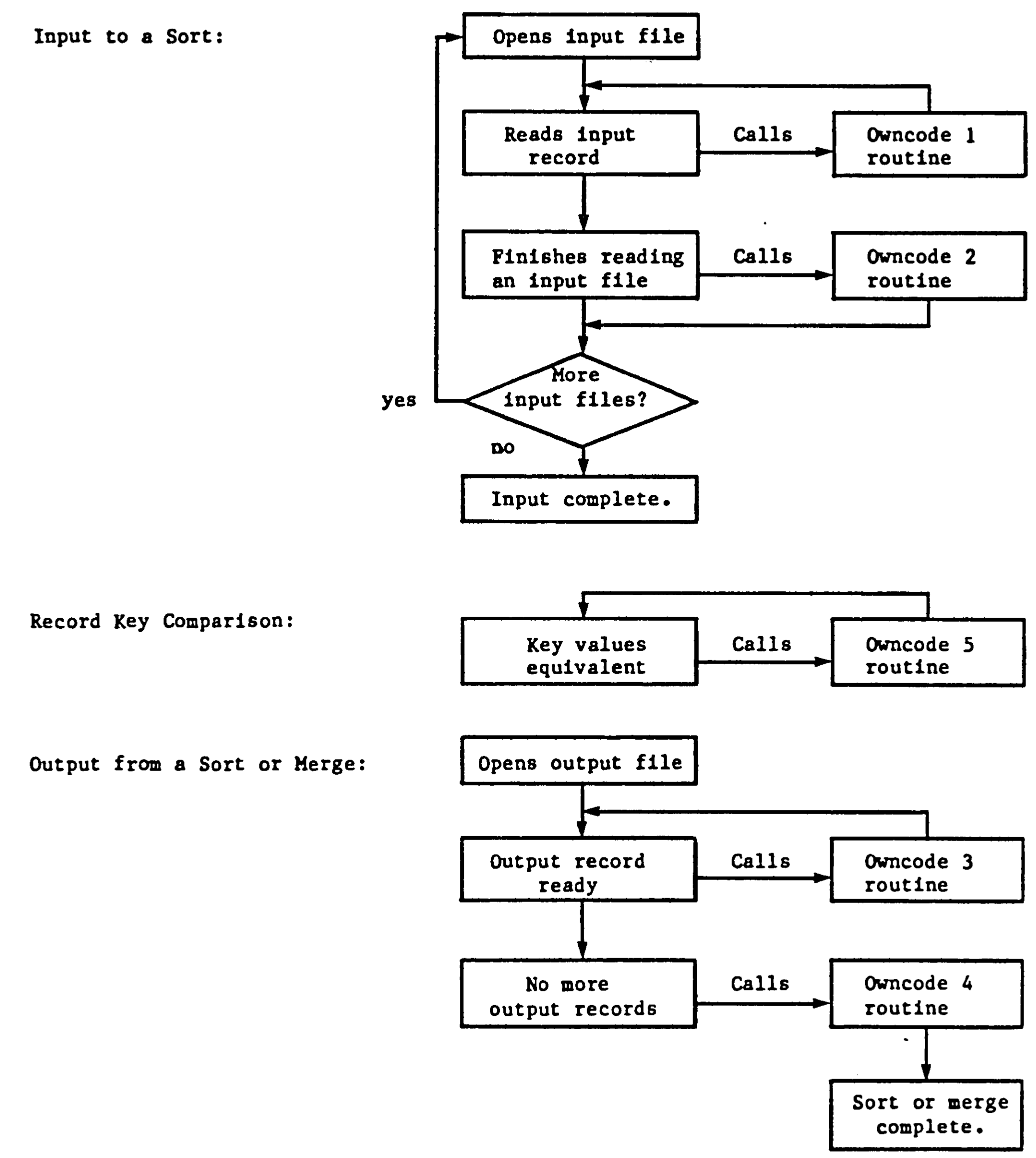


Figure 12-2. When Owncode Routines are Called

An SM5OWNn call specifies the name of an owncode routine Sort/Merge is to use; n is an integer from 1 through 5 that tells Sort/Merge at which point in processing the routine is executed. The SM5OWNn call is described earlier in this chapter.

Owncode routines 1 and 2 can be called for a sort only; owncode routines 3, 4, and 5 can be called for a sort or a merge.

SM5OWNn calls are optional. Each SM5OWNn call in the Sort/Merge sequence of calls must specify a different routine name.

NOTE

When Sort/Merge calls PMP$LOAD to load the owncode routine, it must pass it a name that uses only uppercase letters. Otherwise, PMP$LOAD cannot find the name in the program library list. Therefore, the user must either specify all owncode routine names using only uppercase letters or call SM5CC with the TRUE option to convert the names, if necessary.

You can write an owncode routine using any NOS/VE programming language, including FORTRAN (subroutine subprograms), COBOL (subprograms compiled with COBOL SP=TRUE option), or CYBIL. The owncode routine must be compiled and stored as a module in an object library.

Owncode routines must either be loaded with the main program or be loadable from the program library list. To load an owncode routine, Sort/Merge calls PMP$LOAD to load the routine. PMP$LOAD then searches for the specified owncode routine name in the directories of the object libraries in the program library list.

CYBIL owncode routines must be declared XDCL procedures.

For Sort/Merge to use an object library containing one or more owncode routines, the object library file must be in the program library list. To add a file to the program library list before executing the CYBIL program, execute a SET_PROGRAM_ATTRIBUTES command.

For detailed information on creating object libraries, see the SCL Object Code Management Usage manual. The example at the end of this chapter stores an owncode routine in an object library.

Owncode Procedure Parameters

Sort/Merge communicates with an owncode routine via the procedure parameter list. Sort/Merge passes record data to the procedure and the procedure returns record data and a code indicating how Sort/Merge is to process the record data.

The following lists the required CYBIL procedure parameter list for owncode 1, owncode 2, owncode 3, and owncode 4 procedures:

```
(VAR return_code: integer;
VAR reca: string(*);
VAR rla: integer);
```

The following lists the required CYBIL procedure parameter list for owncode 5 procedures:

```
(VAR return_code: integer;
VAR reca: string(*);
VAR rla: integer;
VAR recb: string(*);
VAR rlb: integer);
```

The return_code parameter passes an integer code back to Sort/Merge specifying how Sort/Merge is to process the returned records. Sort/Merge always initializes the return_code value to 0 when it calls an owncode routine. The owncode routine can leave the return_code value unchanged or change it to one of the valid values for the owncode routine. (The valid values are listed in the individual owncode routine description later in this chapter.) If an invalid return_code value is returned, Sort/Merge returns a fatal error.

The subsequent parameters are used to pass one or two records to the owncode routine. For an owncode 1 through owncode 4 procedure, Sort/Merge passes only one record, the current record being input or ourput. The record data is passed in the reca variable and the record length in bytes is passed in the rla variable.

When calling an owncode 5 procedure, Sort/Merge passes two records having equal keys. The record data is passed in the reca and recb variables and the corresponding record lengths in the rla and rlb variables.

An owncode routine can change the record data and record length values passed to it. The procedure must ensure that the record length value returned is correct for the record data returned. However, Sort/Merge does check that the record length returned does not exceed the maximum record length for the sort or merge.

Owncode Procedure Record Length

Sort/Merge checks the length of each record returned to it by an owncode routine. If a record is too long, Sort/Merge issues an error.

The Sort/Merge specification can explicitly specify the owncode record length. Otherwise, by default, the maximum record length is the largest MAXIMUM_RECORD_LENGTH file attribute value of the input files or output file specified for the sort or merge.

To explicitly specify the owncode record length, you must call SM5OFL or SM5OMRL. If the sort or merge specifies no input or output files, a call to specify the owncode record length is required.

If you call SM5OFL, the length of each record returned by an owncode routine must exactly match the specified record length value.

If you call SM5OMRL, the length of each record returned by an owncode routine cannot exceed the specified record length value.

Owncode 1: Processing Input Records

You specify an owncode 1 procedure to process or supply the input records for a sort. An owncode 1 procedure is used only with a sort request; specifying an owncode 1 procedure with a merge request returns a fatal error.

An owncode 1 procedure cannot be used when SMP$FROM_MEMORY is called.

Owncode 1 procedure processing differs depending on whether input files are specified for the sort.

One or More Input Files Specified

If you specify one or more input files for a sort (even if the input file is $NULL), Sort/Merge calls the owncode 1 procedure each time it reads an input record. Sort/Merge passes the input record to the procedure in the reca variable, the record length (in bytes) in the rla variable, and the return_code variable initialized to 0.

After owncode processing of the record, control returns to Sort/Merge, which processes the record passed back in reca according to the return_code value set by the owncode 1 procedure. The contents of the reca and rla variables can differ from those originally passed to the procedure.

The following are the valid return_code values and their meanings:

0	Sort/Merge sorts the record passed back in reca and reads the next input record.
1	Sort/Merge does not sort the record in reca and reads the next input record.
2	Sort/Merge sorts the record passed back in reca, but does not read the next input record. Instead, Sort/Merge calls the owncode 1 procedure again so additional records can be added to the sort. The owncode 1 procedure should continue to specify return_code 2 until all records to be inserted at this point have been passed; it should then set the return_code to 0.
3	Sort/Merge does not sort the record passed back in reca, closes the current input file, and calls the owncode 2 procedure if one has been specified. After owncode 2 processing has completed, Sort/Merge opens the next input file, if any, and reads the next input record.

For example, to insert one record after the current input record, the owncode 1 procedure performs the following steps:

1. Checks that the record passed in reca is the record after which the new record is to be inserted.

2. Sets the return_code value to 2 and returns control to Sort/Merge.

3. When called again, it stores the new record in reca, stores the length of the new record in rla, sets the return_code value to 0, and returns control to Sort/Merge.

Input Files Not Specified

If you do not specify any input files for the sort (SM5FROM is not called), Sort/Merge calls the owncode 1 procedure as the source of input records. Sort/Merge passes reca as an empty array of the maximum record length, rla set to 0, and the return_code variable initialized to 0.

The following are the valid return_code values and their meanings:

0 Sort/Merge sorts the record passed back in reca, clears the reca array, sets the rla and return_code variables to 0, and calls the owncode 1 procedure again.

2 Sort/Merge sorts the record passed back in reca, leaves the data in reca and the record length in rla, initializes the return_code to 0, and calls the owncode 1 procedure again.

3 Sort/Merge does not sort the record passed back in reca and calls the owncode 2 procedure if one has been specified; otherwise, terminates the input process.

Owncode 2: Processing Input Files

You specify an owncode 2 procedure to supply input records at the end of each input file. An owncode 2 procedure is used only with a sort request; specifying an owncode 2 procedure with a merge request returns a fatal error.

An owncode 2 procedure cannot be used when SM5FMA is used.

Owncode 2 procedure processing differs depending on whether input files are specified for the sort.

One or More Input Files Specified

If you specify one or more input files for the sort (even if the input file is $NULL), Sort/Merge calls the owncode 2 procedure when it terminates input from an input file. It terminates input when it reads an end-of-partition delimiter or the end-of-information or receives a return_code value of 3 from an owncode 1 procedure.

Sort/Merge passes reca as an empty array of the maximum record length, rla set to 0, and the return_code variable initialized to 0.

The following are the valid return_code values and their meanings:

0 Owncode 2 processing ends; Sort/Merge opens the next input file, if any, and reads the next input record.

1 Sort/Merge sorts the record passed back in reca, and calls the owncode 2 procedure again.

For example, to insert one record at the end of an input file, the owncode 2 procedure performs the following steps:

1. Stores the record in reca, stores the record length in rla, sets the return_code to 1, and returns control.

2. When called again, leaves the return_code value set to 0, and returns control to Sort/Merge.

Input Files Not Specified

If you do not specify any input files for the sort (SM5FROM is not called), Sort/Merge calls the owncode 2 procedure after the owncode 1 procedure returns a return_code value of 3.

Sort/Merge passes reca as an empty array of the maximum record length, rla set to 0, and the return_code variable initialized to 0.

The following are the valid return_code values and their meanings:

0 Owncode 2 processing ends, signaling the end of the input records for the sort.

1 Sort/Merge sorts the record passed back in reca, and calls the owncode 2 procedure again.

Owncode 3: Processing Output Records

You specify an owncode 3 procedure to process output records from a sort or merge.

An owncode 3 procedure cannot be used when SM5TMA is called.

Owncode 3 procedure processing differs depending on whether an output file is specified for the sort or merge.

Output File Specified

If you specify an output file for the sort or merge (even if it is $NULL), Sort/Merge calls the owncode 3 procedure each time an output record is ready to be written. Sort/Merge passes the output record to the procedure in the reca variable, the record length in bytes in the rla variable, and the return_code variable initialized to 0.

After owncode processing of the record, control returns to Sort/Merge, which processes the record passed back in reca according to the return_code value set by the owncode 3 procedure. The contents of the reca and rla variables can differ from those originally passed to the procedure.

The following are the valid return_code values and their meanings:

0 Sort/Merge writes the record passed back in reca to the output file. It then passes the next output record, if any, to the owncode 3 procedure.

1 Sort/Merge does not write the record passed back in reca to the output file. It then passes the next output record, if any, to the owncode 3 procedure.

2 Sort/Merge writes the record passed back in reca to the output file, leaves the data in reca and the record length in rla, initializes the return_code to 0, and calls the owncode 3 procedure again.

3 Sort/Merge does not write the record passed back in reca. It calls the owncode 4 procedure if one is specified; otherwise, it terminates the sort or merge.

For example, to insert one record after the current output record, the owncode 3 procedure performs the following steps:

1. Checks that the record passed in reca is the record after which the new record is to be inserted.

2. Sets the return_code value to 2 and returns control to Sort/Merge.
3. When called again, stores the new record in reca, stores the length of the new record in rla, sets the return_code value to 0, and returns control to Sort/Merge.

Output File Not Specified

If you do not specify an output file (you do not call SM5TO call for the sort or merge), the owncode 3 procedure performs all output processing. Sort/Merge passes each output record to the owncode 3 procedure, but it does not process any record returned by the procedure. It does not write any output records.

Sort/Merge passes the output record to the procedure in the reca variable, the record length in bytes in the rla variable, and the return_code variable initialized to 0.

The following are the valid return_code values and their meanings:

0	Sort/Merge calls the procedure again, passing the next output record.
1	Sort/Merge calls the procedure again, passing the next output record.
2	Sort/Merge calls the procedure again, passing the same output record.
3	Sort/Merge terminates the output process, even if it has additional output records. It then calls the owncode 4 procedure if one is specified; otherwise, the sort or merge is terminated.

Owncode 4: Processing the Output File

You specify an owncode 4 procedure to write additional output records to the end of the output file. An owncode 4 procedure can be used with a sort or merge.

An owncode 4 procedure cannot be used when SM5TMA is called.

Owncode 4 procedure processing differs depending on whether an output file is specified for the sort or merge.

Output File Specified

If you specify an output file for the sort or merge (even if it is $NULL), Sort/Merge calls the owncode 4 procedure after it has written its last output record to the output file.

Sort/Merge passes reca as an empty array of the maximum record length, rla set to 0, and the return_code variable initialized to 0.

The following are the valid return_code values and their meanings:

0	Sort/Merge terminates the sort or merge without writing the record passed back in reca.
1	Sort/Merge writes the record passed back in reca and calls the owncode 4 procedure again.

Output File Not Specified

An owncode 4 procedure cannot supply additional output records when no output file has been specified. Still, if you specify an owncode 4 procedure for a sort or merge without an output file, Sort/Merge calls the owncode 4 procedure after the owncode 3 procedure (if any) has terminated output.

Sort/Merge passes reca as an empty array of the maximum record length, rla set to 0, and the return_code variable initialized to 0.

The following are the valid return_code values and their meanings:

0 Sort/Merge terminates the sort or merge.

1 Sort/Merge terminates the sort or merge.

Owncode 5: Processing Records With Equal Keys

When an owncode 5 procedure is specified, Sort/Merge calls the owncode 5 procedure each time it compares the key values of two records and finds that the values are equivalent. It passes both records to the owncode 5 procedure for processing. An owncode 5 procedure is specified by an SM5OWN5 call.

NOTE

Sort/Merge can interpret character key values as equivalent that are not identical. When the collating sequence used for the key assigns the same collating weight to more than one character, those characters are equivalent key values.

An owncode 5 procedure cannot be used when the SM5SUM, SM5RETA, or SM5OMIT call is used. A sort or merge can use only one method of processing records with equivalent key values.

For a given number (n) of records with equivalent key values, each record is passed to the owncode 5 procedure log n times (assuming that duplicate records are not deleted). The order in which the records are passed is not defined.

NOTE

An owncode 5 procedure can change the record data passed to it, but it must not change the data in the key fields of the record. If it does so, the sort order of the modified key field is undefined.

The following are the valid return_code values for an owncode 5 procedure and the meaning of each:

0 Sort/Merge accepts the first rla bytes of reca as the first record and the first rlb bytes of recb as the second record.

1 Sort/Merge accepts the first rla bytes of reca as the first record and deletes recb from the sort or merge.

2 Sort/Merge accepts the first rlb bytes of recb as the first record and the first rla bytes of reca as the second record.

3 Sort/Merge accepts the first rlb bytes of recb as the first record and deletes reca from the sort or merge.

4 Sort/Merge deletes both records from the sort or merge.

5 Sort/Merge does not read the record data returned by the procedure; it processes the two records in their original order (reca before recb).

6 Sort/Merge does not read the record data returned by the procedure, but it deletes the second record (recb) from the sort or merge.

7 Sort/Merge does not read the record data returned by the procedure, but it reverses the order of the two records (recb before reca).

8 Sort/Merge does not read the record data returned by the procedure, but it deletes the first record (reca) from the sort or merge.

For Better Performance

When the owncode 5 procedure does not change the record data, it should use return_code values 5, 6, 7, or 8 instead of return_code values 0, 1, 2, or 3. Performance is improved because Sort/Merge does not read the returned record data.

Do not use return_code 0 to reverse the order of the two records by exchanging the contents of reca and recb. Performing an exchange sort is both incompatible with and much slower than the Sort/Merge sorting algorithm.

If the owncode 5 procedure sorts the two records using one or more keys in addition to those specified for the sort or merge, the procedure should use return_code values 5 and 7 only. (Return_code values 0 and 2 could also be used, but performance would be slower.)

Using FORTRAN Procedure Calls

A FORTRAN program DLIST containing the Sort/Merge procedure calls is shown below. File UNIVERSITY_STUDENTS is read, and student records with grade point average of 3.50 or better are written to an intermediate file (INT1). Sort/Merge is called to sort the file on grade point average in descending order (highest grade point average to lowest grade point average).

```
C
      PROGRAM DLIST
C
C     This program calls Sort/Merge using FORTRAN procedure
C     calls. The purpose of the program is to prepare a
C     list of students with grade point averages of 3.50
C     or better, sort the file on grade point averages in
C     descending order, replace the class code number with
C     the class level, and output the completed report to a
C     new file.
C
C
      INTEGER gpa
      CHARACTER sname*14, major*8, code*1, class*12
      DIMENSION iarray(16)
C
C
      OPEN (1,FILE='university_students')
      REWIND (UNIT=1)
      OPEN (2,FILE='completed_deans_list')
      OPEN (4,FILE='int1')
C
    1 READ (1,100,END=10) sname, major, gpa, code
      IF (gpa .GE. 350) WRITE (4,200) sname, major, gpa, code
      GO TO 1
C
   10 CONTINUE
      CLOSE (UNIT=4,STATUS='KEEP')
C
      IARRAY(1)=15
      CALL SM5SORT (iarray)
      CALL SM5LIST ('$OUTPUT')
      CALL SM5OMRL (80)
      CALL SM5FROM ('int1')
      CALL SM5KEY (33,3,'NUMERIC_NS','D')
      CALL SM5OWN1 ('CCODE')
      CALL SM5TO ('int2')
      CALL SM5END
C
      OPEN (3,FILE='int2')
      REWIND (3)
C
      WRITE (2,400)
   15 READ (3,300,END=20) sname, major, gpa, class
      WRITE (2,500) sname, major, class, gpa
      GO TO 15
```

```
C
  100 FORMAT (A14,12X,A8,I3,A1)
  200 FORMAT (A14,5X,A8,5X,I3,5X,A1,39X)
  300 FORMAT (A14,5X,A8,5X,I3,5X,A12,28X)
  400 FORMAT (36X,'DEANS LIST' // 15X, 'STUDENT',
     *        12X,'MAJOR',8X,'CLASS',12X,'GPA',65X /)
  500 FORMAT (15X,A14,5X,A8,5X,A12,5X,I3,59X)
C
   20 STOP
      END
```

The SM5OWN1 call specifies that an owncode 1 routine named CCODE is to be executed after Sort/Merge reads each record from INT1. Records are passed to the routine by Sort/Merge. The FORTRAN owncode routine is shown below.

```
C
C     This is the FORTRAN owncode routine that is executed
C     after Sort/Merge reads a record.  This routine
C     replaces the number class code with the class
C     level in words.
C     SUBROUTINE CCODE (retcode,rec,rl)
      INTEGER retcode, rl
      CHARACTER code*1, class*12, rec*(*)
C
      code = rec(41:41)
      IF (code .EQ. '1') THEN
         class = 'SENIOR'
      ELSE IF (code .EQ. '2') THEN
         class = 'JUNIOR'
      ELSE IF (code .EQ. '3') THEN
         class = 'SOPHOMORE'
      ELSE IF (code .EQ. '4') THEN
         class = 'FRESHMAN'
      ELSE IF (code .EQ. '5') THEN
         class = 'UNCLASSIFIED'
      ELSE PRINT *, code
C
      END IF
      rec(41:53) = class
C
C     Set the record length for extra length of class level.
      RL = 53
C
      RETURN
      END
```

The SUBROUTINE statement names the routine and the parameters passed by Sort/Merge. Parameter RETCODE is the return_code passed as 0, REC is an array containing the record, and RL is the record length in characters. The routine converts the class code in each record to the class name.

The records are returned to Sort/Merge in array REC. The return_code value is left as 0 because each record in this example is to be sorted. The record received by the owncode routine is lengthened (RL=53) because the class code is converted into a word and needs more space. Sort/Merge then sorts the record to file INT2. The sorted file is returned to the FORTRAN program to be written out in a formatted report. The content of the intermediate files, INT1 and INT2, is shown below. Figure 12-3 shows the output from the job, which is the completed dean's list report.

```
TERRELL   T H      ENG         386     1
SUGARMAN  B T      SOC         350     1
SMITH     C R      MATH        379     1
SHIELDS   L E      COMPSCI     390     1
DAVIS     D A      ENR         354     1
FRANKLIN  R H      PHIL        370     2
CLARK     D N      ECON        378     2
TIEMON    H R      LNGUIS      376     3
HANSEN    R P      BUS         358     3
SMITH     F R      PHIL        385     3
HORNE     D N      COMPSCI     389     4
```

```
SHIELDS   L E      COMPSCI     390     SENIOR
HORNE     D N      COMPSCI     389     FRESHMAN
TERRELL   T H      ENG         386     SENIOR
SMITH     F R      PHIL        385     SOPHOMORE
SMITH     C R      MATH        379     SENIOR
CLARK     D N      ECON        378     JUNIOR
TIEMON    H R      LNGUIS      376     SOPHOMORE
FRANKLIN  R H      PHIL        370     JUNIOR
HANSEN    R P      BUS         358     SOPHOMORE
DAVIS     D A      ENR         354     SENIOR
SUGARMAN  B T      SOC         350     SENIOR
```

```
                         DEANS LIST

        STUDENT                MAJOR         CLASS        GPA

        SHIELDS   L E          COMPSCI       SENIOR       390
        HORNE     D N          COMPSCI       FRESHMAN     389
        TERRELL   T H          ENG           SENIOR       386
        SMITH     C R          PHIL          SOPHOMORE    385
        SMITH     C R          MATH          SENIOR       379
        CLARK     D N          ECON          JUNIOR       378
        TIEMON    H R          LNGUIS        SOPHOMORE    376
        FRANKLIN  R H          PHIL          JUNIOR       370
        HANSEN    R P          BUS           SOPHOMORE    358
        DAVIS     D A          ENR           SENIOR       354
        SUGARMAN  B T          SOC           SENIOR       350
```

Figure 12-3. Output From the FORTRAN Program

Creating an Object Library

You must place an owncode routine into an object library when using command calls. A FORTRAN owncode 3 routine named OWNCODE is shown below. The routine OWNCODE will delete the first record in a file. The variable COUNT keeps track of the number of times the owncode routine is entered.

```
SUBROUTINE OWNCODE (retcode,reca,rla)
INTEGER retcode, rla, count
CHARACTER reca*38
DATA count /0/

count = count +1

IF (count.eq.1) THEN
   retcode = 1
ELSE
   retcode = 0
ENDIF

RETURN
END
```

For detailed information on placing a routine into a library, see the SCL Object Code Management Usage manual. The commands to place OWNCODE into a library named OWN_LIBRARY are shown below.

```
/ftn i=owncode
/create_object_library
COL/add_module library=$local.lgo
COL/generate_library library=$local.own_library
COL/quit
/display_object_library library=$local.own_library ..
../display_option=entry_point



OWNCODE                          - load module


entry points
~~~~~~~~~~~~


OWNCODE

/set_program_attribute add_library=$local.own_library
```

After executing these commands, the routine OWNCODE can be called from a FORTRAN program. A FORTRAN program calling OWNCODE is shown below.

```
PROGRAM OWN
    :
Call sm5sort(0)
Call sm5from('univer2')
Call sm5to('results')
Call sm5key(1,10,'ascii','a')
Call sm5own3('OWNCODE')
Call sm5end
   :
STOP
END
```

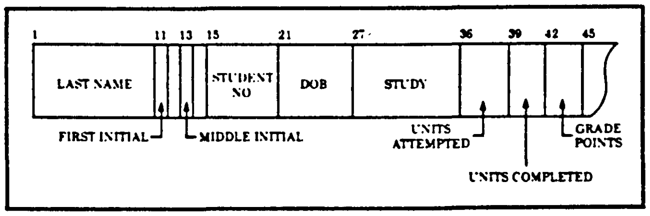
After the FORTRAN program is executed, the file UNIVERSITY_STUDENTS is sorted, with the first record deleted. The sorted records are written to the file RESULTS as shown below.

```
BILLINGS  C Y 101579111855MUS      2965
BRISCOE   J H 102343121157ENVIRO   2544
CARLSON   M K 102126022355ENGIR    3454
CHARLES   S H 101418032459ANTHRO   2453
CLARK     D N 101400102954ECON     3782
CLARK     D V 101023101956ENG      2083
COCHRAN   G L 100725111857BIO      3011
DAVIES    E D 100812080656JOURN    2031
DAVIS     D A 100972071650ENR      3541
  :
WALLIN    G E 101056041659POLISCI 3151
WARNES    D V 102116060861POLISCI 2814
WILSON    W L 101967010261MATH     3454
WONG      S T 101001012755PSYCH    2152
WOO       R M 101315100159BUS      3223
WOODSTOCK C T 101497030160CHEM     3483
YEH       F L 102005120645Art      2764
YOST      D L 100880111158ENG      2582
ZEITZ     F K 100963111858MATH     2612
ZIMMERS   C A 101075063059MATH     2992
```

Note that the owncode routine has deleted the first record in the file.

Summing Records

The record layout of a university student file named STUDENTS is shown below.



Each record contains three numeric fields. They are: number of units attempted, number of units completed, and grade points. The file STUDENTS is shown below with multiple records for each student.

```
GREENWOOD M R 102168101961EDU       002002000
IRVING    W R 101750111855ENG       004004016
GREENWOOD M R 102168101961EDU       003003009
IRVING    W R 101750111855ENG       098095375
QUINTERA  L S  90154101253BIO       003000000
ALLEN     M G 102056012561LNGUIS  005000000
ALLEN     M G 102056012561LNGUIS  025020077
ALLEN     M G 102056012561LNGUIS  004004012
```

Records are to be sorted according to the student number. Using the SM5SUM procedure, records with the same student number are combined into one record by adding the numeric fields together. The new record will give the total number of units attempted, total number of units completed, and the total number of grade points.

The procedure to sort and sum the file STUDENTS is as follows:

```
CALL SM5SORT (0)
CALL SM5FROM ('students')
CALL SM5TO ('summed_file')
CALL SM5KEY (15,6,'ascii','a')
CALL SM5SUM (36,3,'numeric_ns',3)
CALL SM5END
```

The input file STUDENTS is named, and the output file SUMMED_FILE will contain the results of the summing. The student number (positions 15 through 20) is specified as the sort key. The SUM procedure specifies that a three-position numeric field of type NUMERIC_NS begins in position 36 in each record. The repetition indicator specifies that three contiguous fields are to be summed. The output from the sort is shown below. Each record ends with nine digits: the first three digits are the total units attempted, the next three are the total units completed, and the final three are the total grade points.

```
QUINTERA  L S  90154101253BIO       003000000
IRVING    W R 101750111855ENG       102099391
ALLEN     M G 102056012561LNGUIS  034024089
GREENWOOD M R 102168101961EDU       005005009
```

The output file contains one record for each student. The numeric fields are the totals of the units attempted, units completed, and grade points.

Defining Your Own Collating Sequence

The file BIRTHDATES, ordered according to the student name, is shown below. The file contains the students' last names, students' first and middle initials, and the students' dates of birth.

```
ALLEN       M G 10-09-61
ANDERSEN    C R 05-01-60
EBERHARD    N I 06 05 58
GREENWOOD   M R 09-12-61
IRVING      W R 01/07/55
KING        M L 11 11 48
QUINTERA    L S 08/12/53
WALLACE     S T 12/09/55
```

You can standardize the separators in the students' birthdate by defining your own collating sequence.

The FORTRAN procedure to define your own collating sequence is as follows:

```
CALL SM5SORT (0)
CALL SM5FROM ('birthdates')
CALL SM5KEY (25,2,'mysequence','a')
CALL SM5KEY (19,3,'mysequence','a')
CALL SM5KEY (22,3,'mysequence','a')
CALL SM5SEQN ('mysequence')
CALL SM5SEQS ('0','1','2')
CALL SM5SEQS ('-',' ','/')
CALL SM5SEQA ('yes')
CALL SM5TO ('dates_sorted')
CALL SM5END
```

The procedure defines a collating sequence named MYSEQUENCE. The first SEQS procedure specifies the digits 0, 1, and 2, all of which will collate equally. The next SEQS parameter specifies one step consisting of hyphens, blanks, and slashes. This defines the hyphen, blank, and slashes as equal values. The SEQA procedure specifies that blanks and slashes are to be output as hyphens. The file is sorted according to the date of birth.

The file DATES_SORTED output from the sort is shown below.

```
KING        M L 11-11-48
QUINTERA    L S 08-12-53
IRVING      W R 01-07-55
WALLACE     S T 12-09-55
EBERHARD    N I 06-05-58
ANDERSEN    C R 05-01-60
GREENWOOD   M R 09-12-61
ALLEN       M G 10-09-61
```

The file BIRTHDATES has been sorted in numeric order according to dates of birth, and the separators in the dates have been changed to hyphens in all records.

Examples 13

This chapter presents some sample programs that illustrate some of the capabilities of FORTRAN.

Examples 13

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 13-1, 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 13-1. 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 13-2.

```
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 13-2. Program PASCAL Output

Program CORR

Program CORR, shown in figure 13-3, 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 13-3. 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 13-4.

```
/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 OF 10 NUMBERS
? 9999 0 0 0 0 0 0 0 0 0
```

Figure 13-4. Program CORR Output

Program COMPSAL

Program COMPSAL, shown in figure 13-5, 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 13-5. 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 13-6.

```
/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 13-6. Sample Terminal Dialog for Program COMPSAL

Subroutine COUNTC

Subroutine COUNTC, shown in figure 13-7, 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 ', I2, ' 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 13-7. 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 main program accepts 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 the entire string has been tested; control then returns to the calling program.

The main program in figure 13-7 reads a character string from the terminal, reads a single character, calls COUNTC, and prints the results. Figure 13-8 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 13-8. Sample Terminal Dialog for Subroutine COUNTC

Program SCLCALL

Program SCLCALL, shown in figure 13-9, 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 ', I2, /)
      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
```

Figure 13-9. Program SCLCALL

(Continued)

(Continued)

```
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 ', I2)

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 13-9. 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.

Figures 13-10 and 13-11 show 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.

```
/lgo
PARAMETER P1 NOT SPECIFIED.
PARAMETER P2 NOT SPECIFIED.
```

Figure 13-10. Sample Terminal Dialog for Program SCLCALL, Example 1

```
/creariable 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 13-11. Sample Terminal Dialog for Program SCLCALL, Example 2

Appendixes

Glossary A

This section presents a list of definitions of terms used in this manual. It does not include terms defined in the ANSI standard for FORTRAN, X3.9-1978. Terms are listed in alphabetical order.

A

Advanced Access Methods (AAM)

A file manager that processes keyed files.

Alternate Index

An index built in a keyed file for an alternate key. The index associates each alternate key value with a key list of one or more primary-key values.

Alternate Key

An optional key defined in addition to the primary key. An alternate key provides another method of directly accessing records in a keyed file. Unlike the primary key, an alternate key can be defined to allow duplicate values so that more than one record can have the same alternate-key value.

Alternate Key Definition

The set of attributes that specify alternate key characteristics. The alternate-key definition is used to build the alternate index for the key.

Ascending Sort Order

Used with the sort/merge interface, the order of sorting keys where the record having either a numeric or a non-numeric key, the highest value is written last on the output file. For non-numeric, the first item in the sequence has the lowest value. See Sort Order.

ASCII

American National Standard Code for Information Interchange. It is a 7-bit code representing a prescribed set of characters. NOS/VE stores each 7-bit ASCII code right-justified in an 8-bit byte.

B

Backup Copy

Copy kept for possible future recovery. Keyed-file backup copies should be written using the Backup_Permanent_File utility so they can be reloaded using the Recover_Keyed_File utility or the Restore_Permanent_File utility.

Basic Access Methods (BAM)

A file manager that processes sequential files.

Beginning-of-Information (BOI)

The point at which file data begins in a file. For a keyed file, the BOI file position means that the file is positioned to read the record with the lowest key value.

Bit

A binary digit. It has the value 0 or 1. See Byte.

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 records. In a keyed file, blocks are units of file space linked by pointers.

Buffer Statement

One of the input/output statements BUFFER IN or BUFFER OUT.

Byte

A contiguous group of bits. A NOS/VE word has 8 bytes having 8 bits each. NOS/VE stores each ASCII character code in the rightmost 7 bits of a byte.

Byte-Addressable File Organization

A file organization in which records are accessed by their byte address in the file.

C

Calling Sequence

A set of instructions used to transfer control to a subprogram.

Character

A letter, digit, or symbol represented by a code in a character set. Also, a unit of measure used to specify block length, record length, and so forth. Can be a nonprinting symbol or overpunch representation.

Close Operation

A set of terminating operations performed on a file when input and output operations are complete.

Close Request

A program request notifying the system that the program no longer intends to access file data through the specified instance of open. In response, the system flushes all modified data from memory to the file and ends the connection between the program and the file.

Collated Key

The key type that orders key values according to a user-specified collation table. Contrast with Uncollated Key.

Collating Sequence

A set of values defining the collation weights of the 256 ASCII characters. The collation weights determine the sequence in which characters are ordered and their relative values when compared.

Collation Table

A data structure defining a collating sequence.

Collation Weight

The value assigned to a character that determines the position of that character when ordered using the collating sequence.

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.

Concatenated Key

An alternate key that has two or more places. The pieces can be noncontiguous and can be concatenated in any order.

D

Data Block

A keyed-file block in which data records are stored. Contrast with Index Block.

Data Block Split

The process of creating two or three data blocks from an existing data block when a record to be written does not fit into the remaining space of the existing block.

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.

Default Value

The value used for the parameter value if no value is explicitly specified.

Descending Sort Order

Used with the sort/merge interface, the order of sorting keys where the record having either a numeric or a non-numeric key, the lowest value is written last on the output file. For non-numeric, the first item in the sequence has the highest value. See Sort Order.

Direct Access Input/Output

A method of input/output in which records can be read or written in any order.

Display Code

A 64-character subset of the ASCII code, which consists of alphabetic letters, symbols, and numerals.

Duplicate Key Value

The situation detected when a record to be written to the file has a key value that matches a key value already in the file (or another value for the alternate key in the same record). It can also be detected during application of a new alternate-key definition to a file.

Duplicate Key Value Control

The alternate-key attribute that indicates whether duplicate values are allowed for the key and, if so, how the duplicates are ordered.

E

EBCDIC

The abbreviation for extended binary-coded decimal interchange code, an 8-bit code representing a coded character set.

Embedded Key

Key that is part of the data in each record. (Alternate keys are always embedded.) Contrast with Nonembedded Key.

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. For a keyed file, the EOI file position means that the file is positioned after the record with the highest key value.

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.

Exception File

Used with the sort/merge interface, a file to which invalid records are written before the records are removed from the sort or merge.

Execution Time

The time at which a compiled program is executed. Also known as run time.

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

F Record Type

Fixed-length records, as defined by the ANSI standard.

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 define the file structure and processing limitations.

File Cycle

A version of a file. All cycles of a file share the same file entry in a catalog. The file cycle is specified in a file reference by its number or by a special indicator, such as $NEXT.

File Information Table

An internal table through which FORTRAN communicates with the NOS/VE keyed-file interface.

File Organization

The file attribute that determines the record access method for the file. See Sequential File Organization, Byte-Addressable File Organization, and Keyed File Organization.

File Position

The location in the file at which the next read or write operation will begin. The file position designators are:

$ASIS	Leave the file in its current position.
$BOI	Position the file at the beginning-of-information.
$EOI	Position the file at the end-of-information.

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; corresponds to FORTRAN types REAL and DOUBLE PRECISION.

Flush Request

A program request to write to the file device the parts of a file that have been modified in memory since the last time the file was written. For keyed files, the file device is always disk; for sequential files, the flush request can write to disk or to an interactive terminal.

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.

G

Generic Function Name

The name of an intrinsic function that accepts arguments of more than one data type. Except for data type conversion generic functions (and functions with boolean arguments), the type of the result is the same as the data type of the arguments.

Graphic Character

A character that can be printed or displayed.

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

An indexed-sequential file block in which index records are stored. Contrast with Data Block.

Index-Block Split

The process of creating two index blocks from an existing index block when a record to be written does not fit into the remaining space of the existing block.

Index Level

A rank in the index block hierarchy in an indexed-sequential file. To find the pointer to a data record, an index block must be searched at each index level.

Index Level Overflow

The condition when a record cannot be written to a file because writing the record would require addition of another index level and the file already has 15 index levels.

Index Record

A record in an index block that associates a key value with a pointer to either a data block or an index block in the next-lower level of the index hierarchy.

Indexed-Sequential File Organization

A keyed-file organization in which records can be read sequentially, ordered by key value, or read randomly by a key value.

Infinite Value

A value that results from a computation whose result exceeds the largest value that can be represented in the computer. The representation of an infinite value in a computer word does not correspond to the representation of a number.

Instance of Open

A particular opening of a file as distinguished from all other openings of the file. Closing the file ends the instance of open.

Integer Key

The key type that orders key values numerically. The key values can be positive or negative integers. Contrast with Collated Key and Uncollated Key.

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.

J

Job

A sequence of tasks executed for a user number.

Job Log

A chronological listing of all operations associated with a terminal from login to logout.

K

Key

A significant part of a data record.

For Sort/Merge, a key is a part of a record used to determine the position of the record within a sorted sequence of records.

In a keyed file, a key is a part of a record whose value is defined as a means of accessing records. See Primary Key and Alternate Key.

Key List

The sequence of primary-key values associated with an alternate key value in an alternate index. If the alternate key does not allow duplicate values, each key list contains only one value. Otherwise, each key list contains a primary key value for each record that contains the alternate-key value.

Key Type

The kind of data in a key.

For Sort/Merge, a key type is the name of a numeric data format or collating sequence.

For a keyed file, the possible key types are uncollated, collated, and integer.

Keyed File Organization

A file organization that provides for record access by a key value. Currently, the only keyed file organization is the indexed-sequential organization.

Keyword

A word within a format that must be entered exactly as shown.

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

A file that is accessed via the $LOCAL catalog. See also File, Path, and Local Path.

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.

Local Path

Identifies a local file as follows:

```
$LOCAL.file_name
```

Lock

A mechanism that makes a primary-key value (or, for a file lock, all primary-key values) inaccessible to other instances of the file.

Log

Entries recording a chronological series of events. The keyed-file interface uses update recovery logs. See also Update Recovery Log.

Login

The process used at a terminal to gain access to the system.

Logout

The process used to end a terminal session.

M

Major Key

The leftmost part of a key. The number of bytes to be used is specified as the major key length. A major key can be used to position or read a keyed file.

Major Sort Key

Used with the sort/merge interface, a sort key that is the most important and is specified first.

Mass Storage

Disk storage.

Mass Storage File

A particular kind of randomly accessible file, accessed 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.

Media

Storage device on which data is recorded. Currently, NOS/VE files can be recorded on mass storage or magnetic tape.

Merge

The process of combining two or more presorted files.

Minor Sort Key

Used with the sort/merge interface, a sort key that is specified after the major sort key on a SORT or MERGE command or in a procedure call. Minor keys are sorted after the major sort key.

Module

A unit of code. An object module is the unit of object code corresponding to a compilation unit. A load module is a unit of object code stored in an object library.

When using the Debug utility, module refers to a program unit.

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 part of the record data. Contrast with Embedded Key.

Normalized Floating Point Number

A floating point number with the most significant bit of the fractional portion being nonzero.

Null Suppression

Alternate-key attribute indicating that records with null alternate-key values are not included in the alternate index.

O

Object Code

Executable code produced by the compiler.

Object Library

A library of modules that the system can load and execute as needed.

Open

A set of preparatory operations performed on a file before input and output can take place.

Optimization

The manipulation of object code to reduce execution time. You can select the level of optimization performed by the compiler through the OPTIMIZATION_LEVEL parameter on the FORTRAN command.

Owncode

A user-written routine, executed by Sort/Merge, that inserts, substitutes, modifies, or deletes records.

P

Padding

Space deliberately left unused in each block created during the initial open of a keyed file.

Also used to refer to the non-data characters appended to a fixed-length (F) record if the data is shorter than the record length.

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.

Path

Identifies a file. A path may include the family name, user name, subcatalog name or names, and file name.

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.

Piece

One of the fields of a concatenated alternate key.

Primary Key

The required key in a keyed file. Primary-key value must be unique in the file. See also Alternate Key.

Procedure

A FORTRAN function subprogram, subroutine subprogram, or statement function.

Program-Library List

The list of object libraries searched for modules during program loading. A program-library list search is required to load a collation table module or a Sort/Merge owncode procedure module.

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. Also, a set of related data processed as a unit when reading or writing a file.

Record Length

The length of a record measured in words for unformatted input/output and in characters for formatted input/output.

Recovery

Actions taken after damage occurs to alleviate the effects of the damage. Keyed-file recovery actions include reloading a backup copy and restoring the copy with an update recovery log . See also Update Recovery Log.

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.

Relocatable

An object program that can reside in any part of memory. The actual starting address is established at load time.

Repeating Groups

An alternate-key attribute indicating that each data record can contain more than one value for the alternate key.

Rewind

To position a file at its beginning-of-information.

Run Time

The time at which a compiled program is executed; also known as 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 organization in which records can only be processed in physical order. Records are always read in the order that they were written to the file.

Sign

Indicates whether a number is positive or negative. A sign is one of the following characters:

+	Positive number
-	Negative number
space	Positive number

Sort

The process of arranging records in a specified order.

Sort Key

Used with the sort/merge interface, a field of information within each record in a sort or merge input file that is used to determine the order in which records are written to the output file.

Sort Order

Ordering of data according to key fields, either ascending or descending.

Source Code

Code written by the programmer in a language such as 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.

Sparse-Key Control

An alternate-key attribute that allows only certain records to be included in the alternate index. Inclusion or exclusion of a record is determined by the character at the sparse-key control position of the record.

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.

Statistics

Counts maintained for a keyed file. Each type of file access is counted as well as the number of records in the file.

Status Variable

The variable in which the completion status of the command or procedure is returned.

Sum Fields

Used with the sort/merge interface, a record field containing a numeric value from the corresponding field of another record when the records are summed. The sum of the two values is stored in the new record that is created by the summing.

Summing

Used with the sort/merge interface, the process of combining two records having identical key values. The result of the process is a new record containing the original values of the key fields, the summed values of the sum fields, and data from one of the original records in any other record fields.

System Command Language (SCL)

The 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

U Record Type

Records for which the record structure is undefined.

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 to 999, an L format unit file name or a segment access file. 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.

Update Recovery Log

Log on which each backup or update operation to a keyed file is recorded so that, if the file is damaged, a backup file copy can be reloaded and updated using the information on the log.

Utility

A NOS/VE processor consisting of routines that perform a specific operation.

V

V Record Type

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

Word

Eight bytes of information.

Working Storage Area

An area allocated by the task to hold data copied by get or put calls to a file.

Related Manuals B

Table B-1 lists all manuals that are referenced in this manual or that contain background information. A complete list of NOS/VE manuals is given in the SCL Language Definition manual. If your site has installed the online manuals, you can find an abstract for each NOS/VE manual in the online System Information manual. To access this manual, enter:

```
/explain
```

Ordering Printed Manuals

You can order Control Data manuals through Control Data sales offices or through:

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

Accessing Online Manuals

To access an online manual, log in to NOS/VE and specify the online manual title (listed in Table B-1) on the EXPLAIN command. For example, to read the FORTRAN online manual, enter:

```
/explain manual=fortran
```

Table B-1. Related Manuals

Manual Title	Publication Number	Online Title
FORTRAN Manuals:		
FORTRAN Version 1 for NOS/VE Quick Reference		FORTRAN
FORTRAN for NOS/VE Summary	60485919	
FORTRAN for NOS/VE Tutorial	60485912	FORTRAN_T
FORTRAN for NOS/VE Topics for FORTRAN Programmers Usage	60485916	
FORTRAN Version 2 for NOS/VE Language Definition Usage	60487113	
FORTRAN Version 2 for NOS/VE Quick Reference		VFORTRAN
SCL Manuals:		
SCL for NOS/VE Advanced File Management Usage	60486413	AFM
SCL for NOS/VE Language Definition Usage	60464013	
SCL for NOS/VE System Interface Usage	60464014	
SCL for NOS/VE Quick Reference	60464018	SCL
SCL for NOS/VE Source Code Management Usage	60464313	
SCL for NOS/VE Object Code Management Usage	60464413	
Additional References:		
Math Library Usage	60486513	
Debug for NOS/VE Usage	60488213	
Debug for NOS/VE Quick Reference		DEBUG
Diagnostic Messages for NOS/VE Usage	60464613	MESSAGES
Programming Environment for NOS/VE Usage		ENVIRONMENT
Professional Programming Environment Usage	60486613	
Professional Programming Environment Quick Reference		PPE

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.

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.

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.

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.

Division Operation

Dividing by zero in NOS/VE causes a divide fault, which terminates program executon with an immediate fatal runtime error. In NOS, such a division causes an invalid quotient which can generate inaccurate results when used as an operand.

CYBER Record Manager (AAM) Subprograms

The capabilities provided by CYBER Record Manager (CRM) are provided by the file interface routines under NOS/VE. As with CRM, all FORTRAN I/O is performed through the file interface, and a set of FORTRAN subprogram calls provides direct communication with the file interface.

Currently, NOS/VE supports only sequential, indexed-sequential, direct access, and byte-addressable file organizations. Only indexed-sequential and direct access files can be accessed by direct FORTRAN calls. Actual-key file organization is not supported. The Basic Access Methods word addressable organization has been replaced by the new byte-addressable organization.

You should check all uses of CRM Advanced Access Methods (AAM) subprogram calls in your FORTRAN programs. The NOS/VE keyed file interface calls offer only a subset of the features offered by the CRM AAM calls. The following paragraphs describe the feature differences.

File Organization

Currently, the only file organizations available via the keyed file interface calls are indexed-sequential and direct access.

Record Type

The record types available are fixed-length (F) and variable-length (U or V). NOS/VE does not support the AAM Version 2 record types D, R, S, T, and Z.

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 FILEDA, 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 returns an undefined value. If a value is stored in an unconventional manner, the value cannot be returned.

Keywords must be enclosed in apostrophes; for example, 'WSA'. The boolean form L"WSA", used in FORTRAN 5 programs, does not work.

The following CYBER Record Manager file information fields do not have equivalents in the file interface to FORTRAN:

BAL	BBH	BFF	BFS	BS	BT
BZF	B8F	CDT	CL	CM	CNF
CP	CPA	C1	DCA	DFLG	DKI
EFC	EO	EOFWA	EXD	FPB	FWB
HB	HL	HRL	IBL	IRS	KNE
KR	LA	LAC	LBL	LCR	LGX
LL	LNG	LOP5	LP	LT	LVL
LX	MFN	MNB	MUL	NDX	NOFCP
ORG	OVF	PC	PEF	PKA	PM
PNO	POS	PTL	RC	RDR	RMK
SB	SBF	SDS	SES	SOL	SPR
TL	TRC	ULP	VF	VNO	WA
WPN	XBS	XN			

Field FL, although not applicable to the 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. Although NOS/VE does not support data exit, the FORTRAN keyed file interface saves the subroutine address and calls the subroutine when the appropriate file position (BOI or EOI) 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.
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.
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 or FILEDA 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.
KP	Key position. This keyword, although it has no meaning in AAM NOS/VE, is accepted by the FORTRAN interface as a keyword in the CALL FILEIS or CALL FILEDA statement or as a parameter in the CALL STARTM, CALL STOREF, and CALL GET statements. KP is added to KA to determine the position of the key.
RKW	Relative key word. If RKW is specified in a CALL FILEIS or CALL FILEDA statement, the keyed-file interface multiplies the value by 10 and adds 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

Check whether your FORTRAN 5 program uses an INTEGER or REAL array for WSA. Because NOS and NOS/BE use a 10-byte word and NOS/VE uses an 8-byte word, the number of characters that can be stored in an INTEGER or REAL array differs.

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.

To write a FORTRAN program in which the same number of characters can be stored in the WSA when the program is executed by NOS, NOS/BE, or NOS/VE, declare the WSA using the CHARACTER data type.

Optimization

FORTRAN optimization (OL=HIGH) can cause unpredictable results when WSA, KA or PKA are not in common. If OL=HIGH is to be used, WSA, PKA 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 NOS/VE keyed file interface 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 SEEKF 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.

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 9. 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.)

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.

See also Other Collating Sequence Differences in this section.

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.

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.

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.

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.

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.

For example, in NOS, a number that becomes too small due to exponent underflow is rounded to zero, and processing continues. In NOS/VE, you can set an exponent underflow option with an SCL command. The default setting of the option is on, which means that a too small number results in processing being terminated with an immediate fatal runtime error. If you set the exponent underflow option off, NOS/VE treats exponent underflow the same way NOS does.

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.

Function Results

In NOS, function typing could in some cases be incorrect without causing an error. For example, if a double precision function is typed as real in the calling program, the correct data would be returned since only the most significant part of data is returned after a function reference. In NOS/VE, the least significant part is used in passing data, so an error occurs. As another example, NOS handles a character function typed as an integer function correctly up to a certain amount of characters. On NOS/VE FORTRAN, mistyping a character function almost always results in an error.

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.

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.

LOCF Function

The LOCF function is not supported under NOS/VE FORTRAN.

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.

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.

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.

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.

Permanent File Subroutines

The permanent file subroutines are not supported under NOS/VE FORTRAN. A similar capability is provided by the SCL interface subprograms.

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.

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.

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.

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.

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.

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.

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.

Sort/Merge

NOS/VE Sort/Merge is compatible only with Sort/Merge Version 5; it does not attempt compatibility with any other Sort/Merge version. NOS/VE Sort/Merge can only access NOS/VE disk files.

The File Management Utility (FMU) can convert NOS files into equivalent NOS/VE files. This utility converts the differences in byte size, collating sequence, record type, and block type. See the SCL Advanced File Management Usage manual for more details.

The following paragraphs list the major differences between NOS/VE Sort/Merge and NOS Sort/Merge Version 5.

Byte Size

Under NOS/VE Sort/Merge, the byte size is equal to 8-bits rather than 6-bits which is the case under NOS Sort/Merge 5.

Character Codes

Character data is internally represented in 8-bit ASCII character codes under NOS/VE Sort/Merge rather than 6-bit display codes which is the case under NOS Sort/Merge 5.

Character Sets

NOS/VE Sort/Merge supports only the 256-character ASCII character set. NOS/VE Sort/Merge does not support the 63- and 64-character sets.

Collating Sequences

There are now six predefined collating sequences under NOS/VE Sort/Merge: ASCII, ASCII6, COBOL6, DISPLAY, EBCDIC, and EBCDIC6. ASCII is assumed if a sequence is not specified.

Under NOS/VE a user-defined collating sequence has 256 positions. (NOS/VE can use the SEQR procedure to fill the rest).

Direct Processing

NOS/VE Sort/Merge does not support direct processing (all records are read and written through the access method). NOS Sort/Merge 5 reads and writes directly (instead of thorugh CYBER Record Manager) if so specified by the SM5FAST procedure.

Error File

The default error file is $ERRORS under NOS/VE Sort/Merge.

Error Messages

NOS/VE Sort/Merge error numbers and message text follow NOS/VE error message conventions.

The NOS/VE Sort/Merge error messages are listed in the NOS/VE Diagnostic Messages manual.

Estimated Number of Records

For NOS/VE Sort/Merge, the value can be specified on the SM5ENR procedure call.

Exception File Processing

NOS/VE Sort/Merge performs exception file processing if an exception file is specified for the sort or merge.

File Attributes

The NOS default file attributes are valid for a sort or merge.

The NOS/VE default value for the minimum record length attribute could cause a fatal error if no key field was specified for the sort or merge.

File Manipulation

Files are not rewound by NOS/VE Sort/Merge. The open position of a NOS/VE file is determined by the value of its open_position attribute.

Interactive Prompting

Interactive prompting is not currently implemented on NOS/VE Sort/Merge.

Listing File

NOS/VE Sort/Merge provides the SM5LIST procedure to specify the listing file. The default listing file is file $LIST.

Messages

For NOS/VE Sort/Merge, messages are written to the list and error files.

Messages are written to the dayfile for NOS Sort/Merge 5.

Owncode Procedures

For NOS/VE Sort/Merge, any owncode procedures specified for a sort or merge must be accessible from an object library in the current object library list. If you enter an owncode procedure name in lowercase letters, Sort/Merge does not convert the name to uppercase letters. Uppercase letters must be used when naming an owncode procedure.

Procedures for NOS/VE Only

New procedures for NOS/VE Sort/Merge include: SM5DUCT, SM5LCT, and SM5LO procedure calls.

Signed Overpunches

34 overpunches are defined for NOS/VE Sort/Merge; 20 overpunches are defined for NOS Sort/Merge 5.

SM5EL Procedure

The maximum error level can only be specified as a letter for NOS/VE Sort/Merge.

SM5OWNn Procedures

For NOS/VE Sort/Merge, an owncode procedure is specified by the entry point name. If you enter the owncode routine name in lowercase letters, NOS/VE Sort/Merge will not convert the name to uppercase letters. Uppercase letters must be used to name an owncode procedure.

SM5ST Procedure

The NOS/VE SM5ST procedure specifies a status variable in which the completion status of the command or procedure is returned.

Zero Comparison

Positive and negative zero are ordered equally for NOS/VE Sort/Merge.

Negative zero is ordered before positive zero for NOS Sort/Merge 5.

8 Bit Subroutines

The 8 bit subroutines are not supported under NOS/VE FORTRAN.

SYSTEMC or SYSTEM Calls

FORTRAN 5 error numbers are automatically mapped into the corresponding NOS/VE FORTRAN error number for use with the SYSTEM or SYSTEMC calls.

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
- Specification of extensible common or segment access files, called loader control
- Definition of external procedures to be used within your FORTRAN program, called external control
- Definition of SCL parameters to be passed through the execution command (C$ PARAM directive, described in chapter 9)

A C$ directive line is identified by the letter C in position 1 together with the character $ in position 2. Such a line will be interpreted as a comment if the COMPILATION_DIRECTIVES parameter is not 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 warning 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:

1 Enable the specified option.

0 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 1, the first loop is compiled and the second loop is ignored. If M is not 1, the first loop is ignored and the second loop is compiled. Note, however, that the PARAMETER statement sets M = 1.

```
      PROGRAM B
      PARAMETER (M=1)
      DIMENSION A(10)
      DATA A/10*0.0/
          :
C$    IF(M .EQ. 1)
      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
```

Loader Control

The loader control directives are used to allow a named common block to be extensible (using C$ EXTEND) or to associate a named common block with a segment access file (using C$ SEGFILE).

Extensible Common

The C$ EXTEND directive causes a named common block to be extensible. A common block that is extensible has no upper bound other than the size of the memory segment it is contained in. This directive has the form:

C$ EXTEND (bname/, . . . ,*/bname)*

bname

A common block name. The name must be defined by a named COMMON statement in the same program unit.

You must declare an extensible common block to be extensible in all program units that define the named common block. For more information about named common, see the COMMON statement description in chapter 3. Blank common blocks are always extensible.

If the C$ EXTEND directive will be used in program units where over-indexing of arrays may occur, then subscript bounds checking should be deactivated with the RUNTIME_CHECKS = NONE option on the FORTRAN command.

The following example shows the use of the C$ EXTEND directive to make the named common blocks (a, b, and c) extensible.

```
      PROGRAM E
      COMMON /a/ a(1), /b/ b(1), /c/ c(1)
C$    EXTEND (/a/,/b/,/c/)
        ⋮
      a(2)=7
      b(2)=8
      END
```

If the program is compiled with the C$ directive ignored (CD=OFF on the FORTRAN command), the assignments 'a(2)=7' and 'b(2)=8' overwrite the original values in the common block for b(1) and c(1).

Compiling the program with the C$ EXTEND (CD=ON on the FORTRAN command), however, will protect any over-indexed array references within the bounds of the memory segment.

In the following example, note that only the last dimension of the last array in a common block with more than one entity can be over indexed:

```
      PROGRAM X
      COMMON /A/ a(2), b(3), c(3,4)
C$    EXTEND A
        ⋮
```

The array element c(3,5) can be referenced while the array element a(3) can not be because a(3) would reference the memory location as b(1).

Segment Access File Common Blocks

The C$ SEGFILE loader control directive associates a named common block to a segment access file. After a named common block is associated with a segment access file, you can access the file directly through the common block's variables and arrays. This directive has the form:

C$ SEGFILE (bname/, ..., */bname)*

bname

A common block name. The name must be defined by a named COMMON statement in the same program unit.

You must associate, or map, a common block to a segment access file with the OPEN statement. The UNIT specifier on the OPEN statement will specify /bname/.

You must not reference a segment access file until after the OPEN statement has opened and associated the file with a common block. Therefore, you should initialize values in segment access files after the OPEN statement rather than in a DATA statement since the OPEN statement will change the values when it is executed.

The following example shows the use of the C$ SEGFILE directive to map the common CB1 to the segment access file SFILE:

```
      COMMON /CB1/A(1000)
C$    SEGFILE (/CB1/)
      OPEN (UNIT=/CB1/, FILE='SFILE')
```

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.

External Control

The external control directive allows a FORTRAN program to recognize and call a routine written in a language with different naming conventions and calling sequences than FORTRAN. The directive has the form:

C$ EXTERNAL (ALIAS='exname', LANG=lspec), name

exname

Name of the external routine; can be up to 31 characters.

lspec

Selects the programming language in which the external procedure is written. Options are:

C

Selects the C programming language

CYBIL

Selects the CYBIL programming language

FTN

Selects the FORTRAN Version 2 programming language

name

Name of the routine as it will be known in your FORTRAN program. Must be a valid FORTRAN program unit name.

For more information on calling a routine from a FORTRAN program, see Calling Other Language Subprograms in chapter 7.

The following example shows a FORTRAN program that calls a C routine named c_program:

```
      PROGRAM M
      INTEGER JCOUNT
C$    EXTERNAL (ALIAS='c_program', LANG=C) CPROG
      JCOUNT=3
      CALL CPROG(JCOUNT)
      END
```

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 the internal NOS/VE file interface routines.

Indexed sequential files, which can be processed directly through a set of FORTRAN-callable subprograms, are described in chapter 11, Keyed-File Interface.

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 NOS/VE 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.

FORTRAN Fast I/O

FORTRAN fast I/O improves internal processing of buffered, direct-access, and sequential input/output for certain files. The files are those that can be opened with access modes of READ and WRITE, share modes of NONE, and meet the following criteria:

- the file is not a system standard file
- the file is either fixed or variable record type
- the file is a system-specified block type
- the file has no associated file-access procedure
- the file has a blank padding character.

Fast I/O is not used for terminal or tape files. When a file is opened for fast I/O, the file cannot be opened again concurrently. It is necessary to close the file first before it can be opened again. This can cause differences in some input/output situations. The following paragraphs describe the types of differences that occur with fast I/O and ways to deactivate fast I/O.

Open Sharing

Some programs that want to share opens may behave differently due to fast I/O. The message, open share mode NONE, means that a file cannot be opened again by NOS/VE or another language or utility (such as COBOL or SORT), until it is first closed. In some cases, a FORTRAN OPEN statement within the same task can be performed on a file that FORTRAN has already opened. For example, a FORTRAN OPEN satement can be used to change the BLANK= specifier that applies from a previous OPEN statement. This is not really opening the file again so it can be done without the file being sharable. If a file must be shared, and its attributes do not preclude fast I/O, the user must prohibit fast I/O on the file by turning off fast I/O or by using one of the methods described in the following paragraphs to disable fast I/O for individual files.

Files Shared with Another Task or Another Language Subprogram

If another task, or another language subprogram (such as COBOL), tries to do input/output operations on a file that is opened for fast I/O, the routine will fail because other languages try to open the file again. The solution is to close the FORTRAN file before calling another language subprogram.

Connected Files

Programs that write to a file in more than one instance of open may behave differently using fast I/O. For example, assume the following file connection exists:

```
CREATE_FILE_CONNECTION  $ERRORS  TAPE6
```

If a FORTRAN program also opens TAPE6 for fast I/O, then a later attempt to write to $ERRORS causes incorrect information to be written to $ERRORS. This error occurs because TAPE6 cannot be opened once it is already opened for fast I/O. This problem can be avoided by using the commands:

```
CREATE_FILE_CONNECTION  $ERRORS  TAPE6X
CREATE_FILE_CONNECTION  TAPE6  TAPE6X
```

(Neither TAPE6 or $ERRORS is a disk file.)

CALL SCLCMD

If a file is open for fast I/O, a CALL SCLCMD statement to write onto the same file aborts because the SCL command attempts to open the file. For example, if TAPE6 is being used by a FORTRAN program and has been opened for fast I/O, then

```
CALL SCLCMD ('COPY_FILE F TAPE6.$EOI' )
```

will cause the program to abort.

This can be avoided by using the statement

```
CLOSE (6, STATUS = 'KEEP')
```

before the call to SCLCMD.

Fast I/O for Individual Files

Fast I/O is automatically used on files residing in the $local catalog. To prevent this, make the files permanent with the appropriate access and share modes. For example,

```
CREATE_FILE  $USER.TAPE6
DETACH_FILE  $USER.TAPE6
ATTACH_FILE  $USER.TAPE6  SM=NONE
```

Fast I/O will not be used on permanent file $user.tape6 because the file has the default ATTACH_FILE attributes of READ and EXECUTE, but not of WRITE which is required for fast I/O. Fast I/O is also not used when individual files are attached with share modes other than none.

Disabling Fast I/O

To turn off fast I/O, you can do one of the following:

- Create an SCL variable accessible to the FORTRAN program. The variable must be named FLV$IO_OPT_HIGH; it is of type boolean and the initial value is NO (or FALSE or OFF):

  ```
  CREATE_VARIABLE  FLV$IO_OPT_HIGH  BOOLEAN  VALUE=NO  SCOPE=JOB
  ```

  When the value of this variable is NO (at the beginning of a FORTRAN program) no file is opened for fast I/O. Fast I/O can be activated again by the SCL command:

  ```
  FLV$IO_OPT_HIGH = on
  ```

  This variable has no effect at compile time.

- Set the pad character of a file to any character other than blank.

- Use a block type other than system-specified or a record type other than fixed or variable. This method may increase execution time.

- Connect a file to another file or associate a file access procedure to a file. This method may increase execution time.

File and Record Definitions

A file is a collection of records. It is the largest collection of information that can be referenced by a name. A file begins at its beginning-of-information and ends at its end-of-information. A record is a contiguous group of bytes within a file; it 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 a dagger; those attributes which can be overridden prior to any open of the file are indicated by two daggers. 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

File Attribute	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[1]	n/a	n/a	RECL= in OPEN statement
OPEN_POSITION	$BOI[2]	$BOI[2]	$BOI[2]	n/a	n/a
ACCESS_MODE	R/W/A/M[2]	R/W/A/M[2]	R/W/A/M[2]	R/W/A/M[2]	R/W/A/M[2]
FILE_ ORGANIZATION	SQ	SQ	SQ	BA	BA
RECORD_TYPE	V[1]	V[1]	V[1]	U	F
PADDING_ CHARACTER	n/a	n/a	n/a	n/a	blank[1]
PAGE_WIDTH	132 characters (nonconnected file)[1] 72 characters (connected file)[1]	n/a	n/a	n/a	n/a

[1] Can be overridden by SETFA command prior to file creation

[2] Can be overridden by SETFA command prior to any open

n/a = Not applicable to this mode of input/output

$BOI = Beginning of information

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 keywords*
- *FILE_CONTENTS=keyword*
- *FILE_ORGANIZATION=keyword*
- *FILE_STRUCTURE=keyword*
- *MAXIMUM_RECORD_LENGTH=integer*
- *OPEN_POSITION=keyword*
- *PADDING_CHARACTER=character*
- *PAGE_WIDTH=integer*
- *RECORD_TYPE=keyword*

This format shows only those parameters which are applicable to the FORTRAN files described in this chapter. Refer to the SCL System Interface manual for a complete description of the SET_FILE_ATTRIBUTE command for sequential files.

The FILE_CONTENTS attribute must be LIST (to match the OUTPUT attribute) to honor carriage control.

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

File Attribute	**Source Input File**	**Compiler Output Listing File**	**Error File**	**Binary Object File**
FILE_ORGANIZATION	SQ	SQ	SQ	SQ
FILE_STRUCTURE	DATA[1]	DATA[1]	DATA[1]	DATA
FILE_CONTENTS	LEGIBLE[1]	LIST[1]	LIST[1]	OBJECT
RECORD_TYPE	V[1]	V[1]	V[1]	V

[1] 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 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, or the name of a common block enclosed in slashes
fs	format specification
iolist	input/output list
ios	input/output status indicator
recn	record number

Other symbols are defined individually in the statement descriptions. Boldface type indicates required parameters or arguments; italicized type indicates optional parameters or arguments.

Assignment

v = arithmetic expression

boolean v = boolean expression

character v = character expression

logical v = logical or relational expression

Multiple Assignment (CDC Extension)

v=...*v*= **expression**

Type Declaration

INTEGER*length* **v***length,...,v*length*

REAL*length* **v***length,...,v*length*

COMPLEX*length* **v***length,...,v*length*

DOUBLE PRECISION v,...,*v*

BOOLEAN v,...,*v (CDC Extension)*

LOGICAL v,...,*v*

CHARACTER **length*,**v***length,...,v*length*

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

where ac is 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*length,...,name*length*

Storage Allocation

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

where type is INTEGER*length, CHARACTER, BOOLEAN, REAL*length, COMPLEX*length, DOUBLE PRECISION, or LOGICAL.

where d is one through seven array bound expressions separated by commas, as described in chapter 2.

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

where nlist is a list of variables or arrays, separated by commas, to be included in the common block.

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

where 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

where clist is a list of constants or symbolic constants specifying the initial values. Forms for list items are described in chapter 2.

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

where 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***)**

where 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

where 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*

where n is a string of 1 through 5 digits, or a character constant of 1 through 70 characters

END

Main Program

PROGRAM name (upar,...,upar)

where upar is 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*)

where 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

where name is a namelist group name.

Buffer Input/Output (CDC Extension)

BUFFER IN (u,p) (a,b)

BUFFER OUT (u,p) (a,b)

where p is disregarded under NOS/VE.

where a is the first word of the data block to be transferred.

where b is 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

where v is the starting location of the record to be transferred.

where c specifies the number of characters to be transferred to or from each record.

Format Specification

sl FORMAT (flist)

where flist is a list of items, separated by commas, having the following forms:

red
ned
r(flist)

where ed is a repeatable edit descriptor.

where ned is a nonrepeatable edit descriptor.

where r is a nonzero unsigned integer constant repeat specification.

Table F-1. Edit Descriptors

Format	Description
*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.
*r***Lw**	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)
kP	Changes the position of a decimal point of an input or output real number.
BN	Blanks ignored on numeric input.

(Continued)

Table F-1. Edit Descriptors *(Continued)*

Format	**Description**
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 nth 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.

s optional 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.

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, SIZE=n*)

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.
  - Two- and four-byte integer constants have been added.
  - Sixteen-byte real constants have been added.
- 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.

  - Integer expressions can contain two- and four-byte integer values.
  - Real expressions can contain 16-byte real values.

- 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 IMPLICIT NONE statement has been added.
  - 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.
  - The INTEGER*n (where n=2, 4, or 8) statement has been added.
  - The REAL*n (where n=8 or 16) statement has been added.

- 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.
  - Segment 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, COTAN)
    - miscellaneous (SHIFT, MASK, RANF, EXTB, INSB, SUM1S)
  - 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:
    - Keyed-File Interface 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 Keyed Files H

One of the key types supported by the keyed-file interface (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 $COLLATE TABLE NAME (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 J.

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 J. 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')
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.
```

Figure H-1. Creation Program

(Continued)

(Continued)

```
   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

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
```

Figure H-1. Creation Program

(Continued)

(Continued)

```
C  ***************************************************************************
C  * This subroutine sets up the collation table.  The user MUST specify *
C  * a collation table if the key type is collated.                       *
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
      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
```

Figure H-1. Creation Program

(Continued)

(Continued)

```
          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
          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
```

Figure H-1. Creation Program

(Continued)

(Continued)

```
        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

(Continued)

(Continued)

```
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
```

Figure H-1. Creation Program

```
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
```

Figure H-2. Creation Program Output

(Continued)

(Continued)

```
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. Creation Program Output

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								
	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL
0	0	1	2	3	4	5	6	7
	BS	HT	LF	VT	FF	CR	SO	SI
1	8	9	A	B	C	D	E	F
	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB
2	10	11	12	13	14	15	16	17
	CAN	EM	SUB	ESC	FS	GS	RS	US
3	18	19	1A	1B	1C	1D	1E	1F
	SP	!	"	#	$	%	&	'
4	20	21	22	23	24	25	26	27
	(	)	*	+	,	-	.	/
5	28	29	2A	2B	2C	2D	2E	2F
	0	1	2	3	4	5	6	7
6	30	31	32	33	34	35	36	37
	8	9	:	;	<	=	>	?
7	38	39	3A	3B	3C	3D	3E	3F
	@	A	B	C	D	E	F	G
8	40	41	42	43	44	45	46	47
	H	I	J	K	L	M	N	O
9	48	49	4A	4B	4C	4D	4E	4F
	P	Q	R	S	T	U	V	W
10	50	51	52	53	54	55	56	57
	X	Y	Z	[	\	]	^	_
11	58	59	5A	5B	5C	5D	5E	5F
	`	a	b	c	d	e	f	g
12	60	61	62	63	64	65	66	67
	h	i	j	k	l	m	n	o
13	68	69	6A	6B	6C	6D	6E	6F
	p	q	r	s	t	u	v	w
14	70	71	72	73	74	75	76	77
	x	y	z	{	|	}	~	DEL
15	78	79	7A	7B	7C	7D	7E	7F

Figure H-3. Collation Weight Table

(Continued)

(Continued)

Word								
16	80	81	82	83	84	85	86	87
17	88	89	8A	8B	8C	8D	8E	8F
18	90	91	92	93	94	95	96	97
19	98	99	9A	9B	9C	9D	9E	9F
20	A0	A1	A2	A3	A4	A5	A6	A7
21	A8	A9	AA	AB	AC	AD	AE	AF
22	B0	B1	B2	B3	B4	B5	B6	B7
23	B8	B9	BA	BB	BC	BD	BE	BF
24	C0	C1	C2	C3	C4	C5	C6	C7
25	C8	C9	CA	CB	CC	CD	CE	CF
26	D0	D1	D2	D3	D4	D5	D6	D7
27	D8	D9	DA	DB	DC	DD	DE	DF
28	E0	E1	E2	E3	E4	E5	E6	E7
29	E8	E9	EA	EB	EC	ED	EE	EF
30	F0	F1	F2	F3	F4	F5	F6	F7
31	F8	F9	FA	FB	FC	FD	FE	FF

Figure H-3. Collation Weight Table

The Programming Environment and the Professional Programming Environment I

The Programming and Professional Programming Environment both offer a full screen access to NOS/VE products.

Programming Environment

The Programming Environment is a full screen utility that provides functions to facilitate programming in FORTRAN on NOS/VE. The Programming Environment for NOS/VE Summary/Tutorial (publication number 60486819) provides more details about the Programming Environment.

Entering the Environment

The Programming Environment can be entered using the command

ENTER_PROGRAMMING_ENVIRONMENT or
ENTPE
DEFAULT_PROCESSOR = keyword,
ENVIRONMENT_CATALOG = catalog path

ENVIRONMENT_CATALOG must be the catalog path that the Environment is to use to store its files. This parameter defaults to $USER.PROGRAMMING_ENVIRONMENT.

DEFAULT_PROCESSOR may be FORTRAN, COBOL, PASCAL VECTOR_FORTRAN or C. VECTOR_FORTRAN selects the FORTRAN Version 2 programming language. This value is displayed in the programming language field of the Environment screens, and is used to specify the processor for each program created. FORTRAN is the default value.

Do not modify or delete any files in the Programming Environment catalog. If the Environment files are altered, Control Data cannot be responsible for the proper functioning of the Environment.

Providing HELP Information

Depending on where you are in the Environment, a HELP request can:

- Generate a short message
- Take you to a menu of HELP options
- Provide explanations of the current screen
- Take you to the programming language usage manual
- Provide an explanation of the functions currently available

Creating a Program

You can use the Create function to:

- Give the Environment the name of a new program; then code that program while in the Environment
- Specify an existing file, whose contents are to become a program, to be used in the Environment

Modifying a Program

You can use the Modify function to:

- Enter the Full Screen Editor of NOS/VE and edit a program
- Call the Usage Manual of the current programming language
- Format the current program according to the formatting convention of the current programming language

Running a Program

You can use the Run function to:

- Compile a program
- Compile a program that has changed or has compilation parameters that have changed
- Run a previously compiled program
- Fix the detected compilation errors
- Get a message about the status of the run
- Alter run time parameters

Debugging a Program

You can use the Debugging function to:

- Call up the Full Screen Interactive Debugger
- Set and delete breaks for debugging
- Run the program until the next interrupt
- Display a program value
- Change a program value
- Execute the program one line at a time
- Execute the program n lines at a time
- Terminate program execution

Printing Components

You can use the Printing function to print the following components:

- Source programs
- Compilation listings
- Terminal output
- Loadmap

Viewing Components

You can use the View function to view the following:

- Source programs
- Compilation listings
- Terminal output
- Performance graph
- Loadmap

Delete

You can use the Delete function to:

- Remove a component of a program to conserve file space
- Delete a program

Restore

You can use the Restore function to:

- Restore a deleted component within a Programming Environment session
- Restore a deleted program within a Programming Environment session

Exporting a Component

You can copy a component from the environment into a user specified file.

Generating a Program Performance Graph

You can use this function to generate a graph of program performance showing:

- User written routines
- System routines
- Number of calls
- Amount of time spent in each procedure

Tailoring the Environment

You can use this function to change the FORTRAN, COBOL, Pascal, VECTOR_FORTRAN (for FORTRAN Version 2), or C program templates.

Changing Runtime Parameters

You can use this function to:

- Display the current program parameters
- Change the value of a program parameter
- Return a parameter to its compiler default setting
- View parameter lists stored in the parameter list library
- Save a parameter list in the library of named parameter lists

Import a Program

You can bring an existing source program into the Environment with this function.

Professional Programming Environment

The Professional Programming Environment (PPE) is primarily for users working as part of a multi-person programming project that is developing a product for use under the NOS/VE operating system.

Entering PPE

After preparing your terminal for full-screen use (see the Professional Programming Environment usage manual for information on this), you can start PPE by using the SCL command:

ENTER_PPE or
ENTP
  ENVIRONMENT_CATALOG=catalog_path
  STATUS=status variable

ENVIRONMENT_CATALOG or *EC*

Path to the subcatalog for which PPE is executed. It is the lowest level of the PPE hierarchy presented in the session.

PPE creates the subcatalog if it does not exist. If the subcatalog belongs to another user, the owner must grant you the following catalog permit:

Access_Modes=(all, cycle, control)

Application_Information='I1'

If you omit the ENVIRONMENT_CATALOG parameter, the subcatalog used is $USER.PROFESSIONAL_ENVIRONMENT.

STATUS

Optional status variable in which the command returns its completion status.

PPE Capabilities

As a programming environment, PPE integrates the programming tasks, including:

- Editing source text.
- Compiling source text.
- Debugging source text.
- Executing object code.

In addition, PPE can coordinate the activities of a multi-person programming project providing these capabilities:

- Full-screen interface to SCU deck and modification creation.
- Extraction and transmittal of SCU decks and modifications within a source library hierarchy. It enforces interlocks to ensure that only one copy of a deck can be changed.
- Expansion and compilation of the product source, including copying decks from higher levels of the hierarchy when a deck is not present at the lower level.
- Tracing of compilation errors to the decks containing the source.
- Maintenance of an object library at each level of the hierarchy. Each object library contains the compiled code for the source decks at that level.
- Partial builds of the product, expanding and compiling only those source decks that have changed.
- Execution of the product version at the current level of the hierarchy, using object modules at higher levels as needed.

PPE Limitations

- All code for the product being developed must be written in one language.
- The only programming languages supported are NOS/VE FORTRAN Version 1 and NOS/VE COBOL.
- PPE does not support SCU features or groups, nor does it provide a method of changing other modification or deck header information. To assign features and decks to groups (or change the header information), you must use SCU directly. For more information, see the SCL Source Code Management manual.
- PPE does not provide a method of using SCU selection criteria files.

ASCII Character Set and Collating Weight Tables J

Tables B-1 through B-12 give the ASCII character set, the hexadecimal 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.

The following table shows the COLSEQ call parameter values and their corresponding weight table selection:

CLOSEQ Call Parameter Value	Selected Collating Weight Table
ASCII	Standard ASCII
ASCII6	OSV$ASCII6_FOLDED
ASCII6S	OSV$ASCII6_STRICT
COBOL6	OSV$COBOL6_FOLDED
COBOL6S	OSV$COBOL6_STRICT
DISPLAY	OSV$DISPLAY64_FOLDED
DISPLAYS	OSV$DISPLAY64_STRICT
DISPLAY63	OSV$DISPLAY63_FOLDED
DISPLAY63S	OSV$DISPLAY63_STRICT
EBCDIC	OSV$EBCDIC
EBCDIC6	OSV$EBCDIC6_FOLDED
EBCDIC6S	OSV$EBCDIC6_STRICT
INSTALL	OSV$COBOL6_FOLDED

Table J-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

(Continued)

Table J-1. ASCII Character Set and Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
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
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

(Continued)

Table J-1. ASCII Character Set and Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
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
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

(Continued)

Table J-1. ASCII Character Set and Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
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

ASCII codes 80 through FF hexadecimal (not listed in this table) are ordered as equal to the space (ASCII code 20 hexadecimal).

Table J-2. OSV$ASCII6_FOLDED Collating Sequence

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

(Continued)

Table J-2. **OSV$ASCII6_FOLDED Collating Sequence** *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
40	48,68	H,h	Uppercase H, lowercase h
41	49,69	I,i	Uppercase I, lowercase i
42	4A,6Ai	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
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

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).

Table J-3. OSV$ASCII6_STRICT Collating Sequence

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

(Continued)

Table J-3. OSV$ASCII6_STRICT Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
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
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

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).

Table J-4. OSV$COBOL6_FOLDED Collating Sequence

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

(Continued)

Table J-4. OSV$COBOL6_FOLDED Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
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
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

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).

Table J-5. OSV$COBOL6_STRICT Collating Sequence

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	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

(Continued)

Table J-5. OSV$COBOL6_STRICT Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
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
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

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).

Table J-6. OSV$DISPLAY63_FOLDED Collating Sequence

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

(Continued)

Table J-6. OSV$DISPLAY63_FOLDED Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
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
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

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).

Table J-7. OSV$DISPLAY63_STRICT Collating Sequence

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

(Continued)

Table J-7. OSV$DISPLAY63_STRICT Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
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
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

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).

Table J-8. OSV$DISPLAY64_FOLDED Collating Sequence

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

(Continued)

Table J-8. OSV$DISPLAY64_FOLDED Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
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
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

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).

Table J-9. OSV$DISPLAY64_STRICT Collating Sequence

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

(Continued)

Table J-9. OSV$DISPLAY64_STRICT Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
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
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

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).

Table J-10. OSV$EBCDIC Collating Sequence

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

(Continued)

Table J-10. OSV$EBCDIC Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
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
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

(Continued)

Table J-10. OSV$EBCDIC Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
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
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

(Continued)

Table J-10. OSV$EBCDIC Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
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
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

(Continued)

Table J-10. OSV$EBCDIC Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
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

(Continued)

Table J-10. OSV$EBCDIC Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
200	48	H	Uppercase H
201	49	I	Uppercase I
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

(Continued)

Table J-10. OSV$EBCDIC Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
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 J-11. OSV$EBCDIC6_FOLDED Collating Sequence

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

(Continued)

Table J-11. OSV$EBCDIC6_FOLDED Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
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
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

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).

Table J-12. OSV$EBCDIC6_STRICT Collating Sequence

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

(Continued)

Table J-12. OSV$EBCDIC6_STRICT Collating Sequence *(Continued)*

Collating Sequence Position	ASCII Code (Hexadecimal)	Graphic or Mnemonic	Name or Meaning
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
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

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).

Introduction to Debug K

Debug is an SCL command utility that lets you debug a program during execution. Using Debug, you can stop execution at selected points, display the values of selected variables, and resume execution.

Debug is easy to use. It requires no modification of your source code and no knowledge of assembly language. You can reference variables by their symbolic names rather than their addresses in memory. Furthermore, you don't need to interpret memory dumps, insert PRINT statements into your program, or use a load map.

Debug can be used in line mode or screen mode. Also, you can use Debug to perform machine-level debugging as well as symbolic debugging. This discussion focuses on using screen mode Debug for symbolic debugging. For information about line mode Debug, machine-level debugging, and other Debug features, see the Debug Usage manual.

Screen mode Debug gives you all of the Debug features with the ease of use of a full screen interface. You can execute Debug commands by pressing function keys rather than typing commands. Online HELP enables you to learn screen mode Debug as you use it.

Using screen mode Debug, you can:

- View your source code as it executes (an arrow points to the next line to be executed).
- Change the values of program variables while execution is suspended.
- Change the location where execution of your program resumes.
- View the program units of your program.

Getting Started

Using Debug in screen mode requires that your terminal support full screen operation. If your terminal is not set up for full screen operation, see the SCL System Interface manual for terminal definitions that support the full screen interface.

To execute your FORTRAN program with Debug and use the symbolic debugging capability, you must compile the program with the OPTIMIZATION_LEVEL (OL) and DEBUG_AIDS (DA) parameters specified. Furthermore, to enter Debug in screen mode, you must enter the command:

```
CHANGE_INTERACTION_STYLE STYLE=SCREEN
```

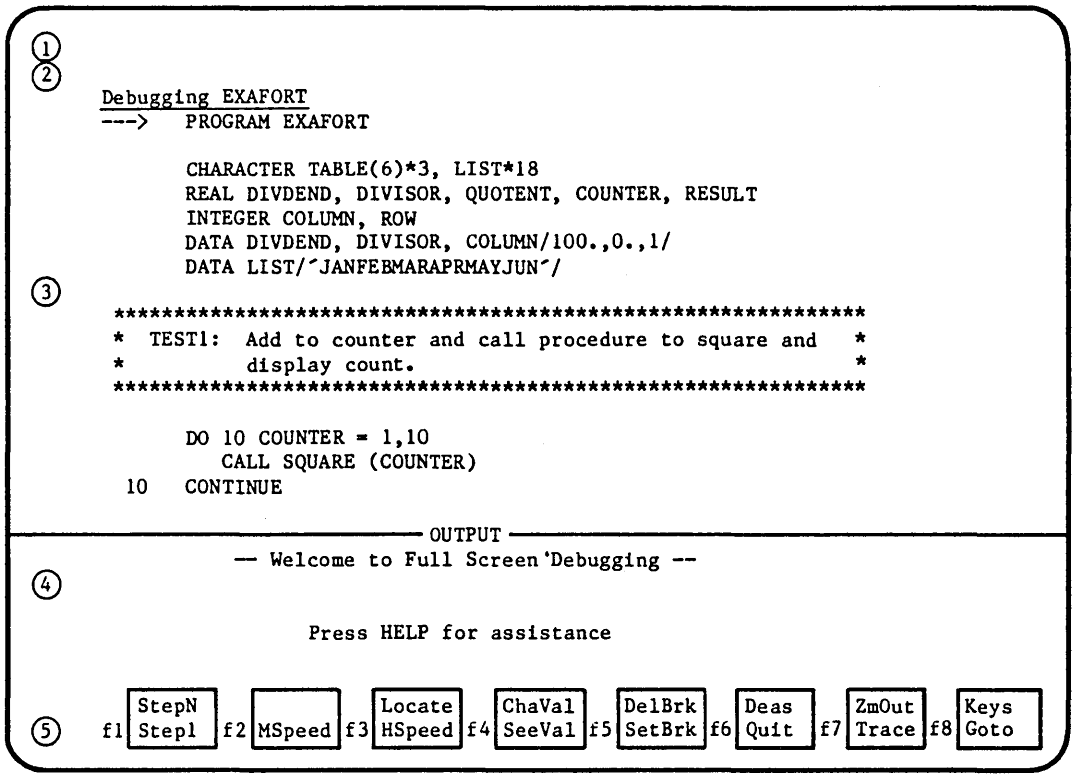
For example, to prepare the source program EXAFORT contained in permanent file $USER.EXAMPLE_FORT for use with Debug, enter the following commands:

```
/change_interaction_style style=screen
/fortran input=$user.example_fort binary_object=lgo optimization_level=debug ..
../debug_aids=all
```

To execute EXAFORT with screen mode Debug, enter the following command:

```
/execute_task file=lgo debug_mode=on
```

The source listing of EXAFORT is displayed as follows on a Viking 721 terminal (on other terminals, the screen format may vary slightly).



```
Debugging EXAFORT
--->   PROGRAM EXAFORT

       CHARACTER TABLE(6)*3, LIST*18
       REAL DIVDEND, DIVISOR, QUOTENT, COUNTER, RESULT
       INTEGER COLUMN, ROW
       DATA DIVDEND, DIVISOR, COLUMN/100.,0.,1/
       DATA LIST/'JANFEBMARAPRMAYJUN'/

*************************************************************
*  TEST1:  Add to counter and call procedure to square and  *
*          display count.                                    *
*************************************************************

       DO 10 COUNTER = 1,10
          CALL SQUARE (COUNTER)
   10  CONTINUE
─────────────────────────── OUTPUT ───────────────────────────
             -- Welcome to Full Screen Debugging --


                   Press HELP for assistance

   StepN       Locate   ChaVal   DelBrk   Deas     ZmOut    Keys
f1 Step1 f2 MSpeed f3 HSpeed f4 SeeVal f5 SetBrk f6 Quit f7 Trace f8 Goto
```

①	Home line	The line on which you enter Debug commands and SCL commands.
②	Response line	The line on which short responses and advisory messages from Debug are displayed.
③	Source window	The area in which the program you are debugging is displayed.
④	Output window	The area in which the output generated by your program (or output delivered by Debug) is displayed.
⑤	Row of function	The Debug functions assigned to function keys. Also, you can enter key assignments Debug commands on the home line.

How to Get Help

There are two ways to get help information while using screen mode Debug:

1. The HELP key.

   Pressing the HELP key displays the Help window. The Help window overlays a portion of your screen and prompts you to enter the function for which you need help. If you press a function key, a short description of the function you select is displayed in the Help window. To exit HELP, press RETURN. Upon exiting HELP, your screen is restored to its original contents.

2. The EXPLAIN command.

   you can request help by entering the EXPLAIN command on the HOME line. This command is used to read an online manual while you are debugging your program. To leave the online manual, press QUIT. When you leave the online manual, the screen is restored to its contents before you entered EXPLAIN. For example, if you need information about FORTRAN constants, press the HOME key and type the following EXPLAIN command on the HOME line:

   ```
         explain s='constants' m=fortran
   ```

   This command takes you to the VFORTRAN online manual for an explanation of FORTRAN constants. To return to screen mode Debug, press QUIT. See the SCL System Interface Manual for more information about EXPLAIN.

Example

This example demonstrates some commonly used Debug functions. It is represented as a series of steps. To get the most benefit from this example, you should create the sample program, EXAFORT, illustrated in figure K-1 then perform each step.

EXAFORT is divided into the following test cases:

TEST1

A loop that increments a counter and then calls a subprogram to square and display the count. TEST1 demonstrates the use of the CHAVAL, GOTO, HSPEED, SEEVAL, STEP1, and STEPN functions.

TEST2

A loop that builds a 6-row table of 3-character strings. Input to the table is an 18-character list for the months JAN through JUN. TEST2 moves three characters at a time from the character list to the table and displays each entry. TEST2 shows how to step through loops, use line mode Debug commands in screen mode Debug, and how to scroll through Debug and program output data.

TEST3

A division test that results in a divide fault. TEST3 demonstrates how Debug handles execution errors.

In each test case, the application of some Debug functions is demonstrated. After you work this example, you can begin to debug your FORTRAN programs using screen mode Debug.

```
      PROGRAM EXAFORT

      CHARACTER TABLE(6)*3, LIST*18
      REAL DIVDEND, DIVISOR, QUOTENT, COUNTER, RESULT
      INTEGER COLUMN, ROW
      DATA DIVDEND, DIVISOR, COLUMN/100.,0.,1/
      DATA LIST/'JANFEBMARAPRMAYJUN'/

****************************************************************
*  TEST1:  Add to counter and call procedure to square and     *
*          display count.                                      *
****************************************************************

      DO 10 COUNTER = 1,10
        CALL SQUARE (COUNTER)
   10 CONTINUE

****************************************************************
*  TEST2:  Create single column table for each month.          *
****************************************************************

      DO 20 ROW = 1,6
        TABLE(ROW) = LIST(COLUMN : COLUMN + 2)
        PRINT*, 'THE MONTH IS:  ', TABLE(ROW)
        COLUMN = COLUMN + 3
   20 CONTINUE

****************************************************************
*  TEST3:  Create divide fault.                                *
****************************************************************

      QUOTENT = DIVDEND / DIVISOR
      PRINT*, 'ANSWER IS:  ', QUOTENT

      END

****************************************************************
*  Subroutine SQUARE                                           *
****************************************************************

      SUBROUTINE SQUARE (COUNTER)
        RESULT = 0.
        RESULT = COUNTER * COUNTER
        PRINT*, COUNTER, ' TIMES ', COUNTER, ' = ', RESULT
      END
```

Figure K-1. Example of EXAFORT Source Listing

Preparing to Debug

After you create EXAFORT, you must prepare it for use with screen mode Debug. This requires preparing the screen mode environment and compiling EXAFORT for use with Debug. You can then execute it under screen mode Debug control.

1. Prepare and compile EXAFORT contained in permanent file $USER.EXAMPLE_FORT specifying the OPTIMIZATION_LEVEL=DEBUG and DEBUG_AIDS=ALL parameters by entering the following commands:

```
/change_interaction_style style=screen
/fortran input=$user.example_fort binary_object=lgo ..
../optimization_level=debug debug_aids=all
```

2. Execute EXAFORT under control of Debug by entering the following command:

```
/execute_task file=lgo debug_mode=on
```

The source listing of EXAFORT is displayed in the source window. The Debug functions are displayed at the bottom of the screen.

Display Screen Mode Commands

The function below is used to display helpful information about the Debugging enviornment:

HELP

Displays the Help window. Press a function key and a short explanation of the function's use appears in the Help window.

Now perform the following steps to become familiar with the Debug functions:

1. Press the HELP key. The Help window is displayed.
2. Press each function key corresponding to a function displayed at the bottom of the screen. As you press each function key, a short explanation of the purpose of each function is displayed in the Help window.
3. Press RETURN. Exit HELP.

Setting Breaks

It is often helpful to suspend program execution when debugging a program. The device for suspending execution of a program is called a break. In this sample session, the following functions are used to illustrate setting breaks:

BKW

Scrolls backward to the previous screen of text.

FIRST

Displays the first screen of the source listing. Because FIRST is a lower priority function, it may not be assigned to a function key on terminals with only 16 function keys. Instead, FIRST is entered on the HOME line.

FWD

Scrolls forward to the next screen of text.

LOCATE

Prompts you to type in text, then searches the source listing for matching text. If a match is found, the cursor is moved to the line containing the matching text.

SETBRK

Sets an execution break on the line containing the cursor. The line is highlighted to show that it contains a break. Execution is suspended before the line containing the break is executed. Execution resumes with the statement on the line containing the break.

Perform the following steps to place three execution breaks in EXAFORT:

1. Press the LOCATE function key. At the top right hand corner of the screen, you are prompted for the text to be located.
2. Enter the following text exactly as it appears in EXAFORT:

   ```
   DO 20
   ```

   The cursor is moved to the line:

   ```
   DO 20 ROW = 1,6
   ```

3. Press the SETBRK function key. A break is set and the line containing the cursor is highlighted to show that it contains an execution break.
4. Use the down-arrow key to move the cursor to the line containing:

   ```
   COLUMN = COLUMN + 3
   ```

   If you do not see this line on your screen, press the FWD key. The next screen of the EXAFORT source listing is displayed. Use the down-arrow key to position the cursor on the correct line.
5. Press the SETBRK function key. The line is highlighted to show that it contains an execution break.
6. Use the down-arrow key to move the cursor to the line:

   ```
   QUOTENT = DIVDEND / DIVISOR
   ```

   If you do not see this line on your screen, press the FWD key. The next screen of the EXAFORT source listing is displayed. Use the down-arrow key to position the cursor on the correct line.
7. Press the SETBRK function key. The line is highlighted to show that it contains an execution break.
8. Press the FIRST function key. The first screen of the EXAFORT source listing is displayed in the source window.

   If FIRST is not assigned to a function key, FIRST must be entered on the HOME line. To do this, press the HOME key. This moves the cursor to the HOME line. Enter the following on the HOME line:

   ```
   first
   ```

   The first screen of the EXAFORT source listing is displayed in the source window.

Debugging TEST1

Using Debug, you can execute a program one line or several lines at a time. Also, you can examine a variable's contents, change its contents, and execute code containing the variable several times. These capabilities are demonstrated in this sample session using the following functions:

CHAVAL

Prompts you to enter a variable name and the value you want it to contain, then changes the variable's contents to the new value.

GOTO

Moves the execution pointer to the line that contains the cursor. Execution resumes with the statement on this line.

HSPEED

Executes a program until a break is encountered or the program ends.

SEEVAL

Prompts you to enter a variable name, then displays the value of the variable in the output window.

STEP1

Executes a program one line at a time.

STEPN

Executes N lines of a program, where N is an integer.

Perform the following steps to demonstrate the use of the CHAVAL, GOTO, HSPEED, SEEVAL, STEP1, STEPN funtions:

1. Press the STEP1 function key. The first statement of EXAFORT is executed, moving the execution arrow to the statement:

   ```
   DO 10 COUNTER = 1,10
   ```

2. Press the STEP1 function key again. The DO statement is executed; the execution arrow points to the statement:

   ```
   CALL SQUARE (COUNTER)
   ```

3. Press the STEP1 function key six times. An iteration of TEST1 is executed one line at a time. The output from the iteration is displayed in the output window.

4. Press the SEEVAL function key. A prompt to enter a variable name is printed in the upper right hand corner of the screen. Enter the name:

   ```
   counter
   ```

   The value of COUNTER is displayed in the output window:

   ```
   counter = 2.
   ```

   Thus, you can use SEEVAL to observe the contents of a variable.

5. Press the CHAVAL function key. A prompt for a variable name and its new value is displayed in the upper right hand corner of the screen; enter:

   ```
   counter=8
   ```

   The value of COUNTER is changed to 8.

6. Press the SEEVAL function key. When you are prompted for a variable name, enter:

   ```
   counter
   ```

   The following message is displayed in the output window:

   ```
   counter = 8.
   ```

   Thus, the change of COUNTER's value is verified.

7. Press the STEPN function key. In the upper right hand corner of the screen, you are prompted for the number of lines to execute; enter:

   ```
   6
   ```

   STEPN executes 6 lines of TEST1. The output from this loop iteration is displayed in the output window.

8. Press the SEEVAL function key. When you are prompted for a variable name, enter:

   ```
   counter
   ```

   The value of COUNTER is displayed in the output window:

   ```
   counter = 9.
   ```

   Therefore, the value given to COUNTER in step 5 is used by the DO statement.

9. Use the up-arrow key to move the cursor to the line:

   ```
   DO 10 COUNTER = 1,10
   ```

10. Press the GOTO function key. The execution arrow moves to the line containing the cursor; execution resumes with this statement.

11. Press the HSPEED function key. Execution resumes from the DO statement; COUNTER is initialized to 1. Execution of EXAFORT continues until an execution break is encountered.

Debugging TEST2

After program execution is resumed in step 11 of TEST1, it stops at the break set on the DO statement in TEST2. The following functions are used in TEST2 to illustrate more Debug capabilities:

BKW

Scrolls backward to the previous screen of text.

DELBRK

Deletes execution breaks.

HSPEED

Executes a program until a break is encountered or the program ends.

This section also uses the following items:

HOME

Press the HOME key to move the cursor to the HOME line. Line mode Debug commands can be entered on the HOME line for execution in screen mode Debug.

DISPLAY_PROGRAM_VALUE

A line mode Debug command that displays the values of program variables.

Perform the following steps to learn how to execute loops one iteration at a time, execute line mode Debug commands, and scroll output data when using Debug:

1. Press the HSPEED function key. Execution stops at the break set on the last line of the DO loop in TEST2; output from the loop is displayed in the output window.

2. Press the HSPEED function key again. One iteration of the DO loop is executed; execution stops at the break set at the statement, COLUMN = COLUMN + 3. Each time HSPEED is used, an iteration of the loop is performed. By using strategically placed execution breaks, as in this example, a loop can be executed one iteration at a time.

3. Press the HSPEED function key. One more loop iteration is performed.

4. Press the HOME key. The cursor moves to the HOME line.

5. Enter the line mode Debug command:

   ```
   display_program_value name=$all
   ```

   The values of all variables in EXAFORT are displayed in the output window. Thus, line mode Debug commands can be used in screen mode Debug by entering them on the HOME line. For more information about using line mode Debug commands see the Debug Usage Manual.

6. Press the DELBRK key. The execution break is deleted.

7. Press the down-arrow key until the cursor is inside of the output window.

8. Press the BKW key. The data in the output window scrolls backward. When the cursor is contained within the output window, you can use the BKW and FWD keys to scroll backward and forward through the data in the window.

9. Press the HSPEED function key. The execution of EXAFORT resumes, stopping when the line containing the third break is reached. The execution arrow points to the first statement of TEST3.

Debugging TEST3

After resuming execution of EXAFORT in step 9 of section TEST2, execution stops at the begining of TEST3. In TEST3, Debug is presented with an execution error. The following functions are used in this sample session to demonstrate how Debug can be used when an execution error is encountered:

CHAVAL

Prompts you to enter a variable name and the value you want it to contain, then changes the variable's contents to the new value.

GOTO

Moves the execution pointer to the line that contains the cursor. Execution resumes with the statement on this line.

SEEVAL

Prompts you to enter a variable name, then displays the value of the variable in the output window.

STEP1

Executes a program one line at a time.

QUIT

Used to exit Debug.

Perform the following steps to finish the example:

1. Press the STEP1 function key again. The DIVISION statement is executed, execution of EXAFORT halts, and the following message flashes in the top right hand corner of the screen:

   ```
   divide_fault
   ```

2. Press the SEEVAL function key. When you are prompted for a variable name, enter:

   ```
   divisor
   ```

   The following message is displayed in the output window:

   ```
   divisor = 0.
   ```

   A division by zero caused the execution error.

3. Press the CHAVAL function key. When you are prompted, enter:

   ```
   divisor=1
   ```

   The value of DIVISOR is changed to 1.

4. Press the SEEVAL function key. When you are prompted, enter:

   ```
   divisor
   ```

   The following text is displayed in the output window:

   ```
   divisor = 1.
   ```

   The change to DIVISOR is verified.

5. Press the GOTO function key. The execution arrow points at the DIVISION statement and program execution resumes with this statement.

6. Press the STEP1 function key. The DIVISION statement is executed. Therefore, the GOTO and CHAVAL functions can be used in concert to recover from execution errors. However, to correct execution errors permanently, you must exit Debug, edit the program, and recompile it.

7. Press the STEP1 function key again. The result of the DIVISION statement is displayed in the output window.

8. Press the STEP1 function key. EXAFORT ends and the following message is displayed in the output window:

   ```
   DEBUG:  The status at termination was: NORMAL.
   ```

9. Press the QUIT function key. Exit Debug.

Now that you have concluded this example, you should be able to begin using screen mode Debug to debug your FORTRAN programs. For more information about screen mode Debug and line mode Debug commands, see the Debug Usage manual.

Index

Index

FORTRAN Version 1 Language Definition Usage 60485913 H

We value 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?	**How do you use this manual?**
☐ Manager	☐ As an overview
☐ Systems analyst or programmer	☐ To learn the product or system
☐ Applications programmer	☐ For comprehensive reference
☐ Operator	☐ For quick look-up
☐ Other ______________________	

What programming languages do you use? ______________________________

How do you like this manual? Check those questions that apply.

Yes	Somewhat	No	
☐	☐	☐	Is the manual easy to read (print size, page layout, and so on)?
☐	☐	☐	Is it easy to understand?
☐	☐	☐	Does it tell you what you need to know about the topic?
☐	☐	☐	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?

Comments? If applicable, note page and paragraph. Use other side if needed.

Would you like a reply? ☐ Yes ☐ No

From:

Name ______________________	Company ______________________
Address ______________________	Date ______________________
______________________	Phone ______________________
______________________	

Please send program listing and output if applicable to your comment.

Comments (continued from other side)

Please fold on dotted line;
seal edges with tape only.

FOLD

NO POSTAC
NECESSAR
IF MAILEI
IN THE
UNITED STA'

BUSINESS REPLY MAIL
First-Class Mail Permit No. 8241 Minneapolis, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CONTROL DATA
Technology & Publications Division
SVL104
P.O. Box 3492
Sunnyvale, CA 94088-3492

CONTROL DATA