IBM RT PC

VS FORTRAN Reference Manual

Programming Family

IBM
Personal
Computer
Software

SH23-0130

VS FORTRAN Reference Manual

Programming Family

Personal
Computer
Software

First Edition (March 1987)

The information in this manual applies to Version 1 of IBM RT PC VS FORTRAN for use with Release 2.1 of the AIX Operating System; and it applies to all subsequent releases and modifications until otherwise indicated in new editions or Technical Newsletters.

Changes are made periodically to the information herein; these changes will be incorporated in new editions of this publication.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM program product in this publication is not intended to state or imply that only IBM's program product may be used. Any functionally equivalent program may be used instead.

International Business Machines Corporation provides this manual "as is" without warranty of any kind, either express or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this manual at any time.

Requests for copies of this product and for technical information about the system should be made to your authorized IBM RT PC dealer.

A reader's comment form is provided at the back of this publication. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Preface

This reference manual contains a formal description of the FORTRAN 77 programming language as implemented in IBM RT PC VS FORTRAN. Enhancements to FORTRAN 77 are highlighted in blue. IBM RT PC VS FORTRAN is for use on an IBM RT Personal Computer (RT PC[1]) operating on the AIX[2] Operating System.

The procedures for compiling and running FORTRAN programs using IBM RT PC VS FORTRAN are described in the *RT PC VS FORTRAN User's Guide.*

Note: The symbol "✦" or "✧" in a section title indicates that related information specific to R1 or VX mode, respectively, is described under the same heading at the end of the chapter. Modes are described in Chapter 1, "Introduction."

Contents:

Chapter 1 — "Introduction" gives a general overview of RT PC VS FORTRAN and introduces the terms and concepts of the language.

Chapter 2 — "Characters, Lines, Statements, and Execution Sequence" describes the RT PC VS FORTRAN character set, types of lines, types of statements, and the execution sequence.

Chapter 3 — "Data Types and Constants" describes the data types and the kinds of constants used in RT PC VS FORTRAN.

---

1 RT PC is a trademark of IBM Corporation

2 AIX is a trademark of IBM Corporation

Chapter 4 — "Names, Arrays, and Substrings" includes rules for naming variables and constants, how to define and reference arrays, and a description of character variables and character substrings.

Chapter 5 — "Expressions" describes the four types of expressions used in statements.

Chapter 6 — "Assignment Statements" describes the four types of assignment statements.

Chapter 7 — "Specification Statements" describes the various types of specification statements.

Chapter 8 — "Control Statements" describes the various types of control statements.

Chapter 9 — "Program and Subprogram Structure" describes the structure of RT PC VS FORTRAN programs in terms of program units, which include the main program and three types of subprograms.

Chapter 10 — "Input and Output" describes the main features of input and output in RT PC VS FORTRAN, including the file system and the various input/output statements.

Chapter 11 — "Format Specifications" describes formatted input/output and the FORMAT statements available in RT PC VS FORTRAN.

Appendix A — "Intrinsic Functions" contains a table of FORTRAN intrinsic functions, which are "built-in" functions that perform mathematical computations, bit-manipulation routines, and nonstandard library routines.

Appendix B — "Information for the FORTRAN 66 Programmer" contains information to assist the FORTRAN 66 programmer in using FORTRAN 77.

Related Publications

You may want to refer to the following IBM RT PC publications for additional information:

- *VS FORTRAN User's Guide*, SH23-0129, describes the procedures for compiling and running RT PC VS FORTRAN programs under the AIX Operating System.
- *VS Pascal User's Guide*, SH23-0127, describes the procedures for compiling and running RT PC VS Pascal programs under the AIX Operating System.
- *VS Pascal Reference Manual*, SH23-0128, describes the Pascal programming language as implemented on the RT PC.
- *VS Language/Operating System Interface Library*, SH23-0131, describes the system routines that can be called from FORTRAN and Pascal programs.
- *Concepts*, GC23-0784, gives an overview of the RT PC hardware, the AIX Operating System, and supporting publications.
- *Installing and Customizing the AIX Operating System*, SV21-8001, provides step-by-step instructions for installing and customizing the AIX Operating System, including instructions for adding devices to and deleting them from the system and for defining device characteristics. This book also explains how to create, delete, and change AIX and non-AIX minidisks.
- *Messages Reference*, SV21-8002, lists messages displayed by the RT PC and explains how to respond to the messages.
- *Usability Services Guide* and *Usability Services Reference*, SV21-8003, show how to create and print text files, work with directories, start application programs, and do other basic tasks.
- *Using and Managing the AIX Operating System*, SV21-8004, contains information on using AIX Operating System commands, working with the file system, developing shell procedures, and performing such

system-management tasks as creating and mounting file systems, backing up the system, and repairing file-system damage.

- *AIX Operating System Commands Reference*, SV21-8005, lists and describes the AIX Operating System commands.
- *C Language Guide and Reference*, SV21-8008, provides information for writing, compiling, and running C language programs.
- *AIX Operating System Technical Reference*, SV21-8009, describes the system calls and subroutines a programmer would use to write application programs. This book also provides information about the AIX Operating System file system, special files, miscellaneous files, and the writing of device drivers.
- *AIX Operating System Programming Tools and Interfaces*, SV21-8010, describes the programming environment of the AIX Operating System and includes information about the use of operating system tools to develop, compile, and debug programs.
- *AIX Operating System DOS Services Reference*, SV21-8012, provides step-by-step information for using the AIX Operating System shell. In addition, this book describes the DOS system services.
- *User Setup Guide*, SV21-8020, provides instructions for setting up and connecting devices to system units. It also gives procedures for installing the AIX Operating System and for testing the setup.
- *Guide to Operations*, SV21-8021, describes system units, displays, console keyboard, and other devices that can be attached to the RT PC. This guide also includes procedures for operating the hardware and for moving system units.
- *Problem Determination Guide*, SV21-8022, provides instructions for running diagnostic routines for hardware and problem-determination procedures for software.

You may also want to consult the IBM RT PC FORTRAN 77 Version 1.1 publications.

Contents

Chapter 1. Introduction

IBM RT PC VS FORTRAN is an easy-to-use, high-level programming language for the RT Personal Computer. It compiles source code in FORTRAN as defined by IBM VS FORTRAN Version 2, IBM RT PC FORTRAN 77 Version 1.1, ANSI Standard FORTRAN 77, and VAX[1] FORTRAN Version 3.

In addition to excellent performance, IBM RT PC VS FORTRAN offers these enhanced functions:

- Automated installation
- Source compatibility with IBM VS FORTRAN Version 2[2]
- Source compatibility with IBM RT PC FORTRAN 77 Version 1.1[2]
- Source compatibility with ANSI Standard FORTRAN 77
- Source compatibility with VAX FORTRAN Version 3[2]
- Optimized executable code
- Excellent compile-time performance
- An operating system interface library
- No significant limit on program size
- No significant limit on data size
- Separate unit compilation
- Access to command-line options
- Common development/debugging environment
- Detailed screen messages
- Easy inter-language linkages with Pascal and C.

You may select one of four compiler modes: IBM mode, R1 mode, AN mode, or VX mode. You may work in the mode you need or with which you are most familiar.

---

1 Trademark of Digital Equipment Corporation

2 See the *RT PC VS FORTRAN User's Guide* for limitations.

IBM Mode

This is the default mode of the compiler, and it allows you to compile code written in IBM VS FORTRAN Version 2 (see the *RT PC VS FORTRAN User's Guide* for limitations).

You may develop and run IBM mode programs entirely on the RT PC. As a cost-effective development tool, you may develop and run IBM mode programs on an independent RT PC workstation and then move the programs to a mainframe that uses VS FORTRAN Version 2.

You may also take programs written in IBM VS FORTRAN Version 2 from a mainframe and run them on your RT PC.

IBM mode contains all of the ANSI Standard FORTRAN 77 facilities; you may use ANSI Standard FORTRAN 77 code in IBM mode, and can improve it using IBM mode enhancements.

R1 Mode

This mode allows you to compile code written in IBM RT PC FORTRAN 77 Version 1.1 (see the *RT PC VS FORTRAN User's Guide* for limitations). You can take code written in this version of FORTRAN and recompile it in IBM RT PC VS FORTRAN in order to take advantage of its improvements and additional features.

AN Mode

This mode allows you to compile code written in ANSI Standard FORTRAN 77. Code that is to adhere to this definition of FORTRAN can be compiled in this mode; during program compilation, you are warned when any extension to this definition is used.

VX Mode

This mode allows you to compile code written in VAX FORTRAN Version 3 (see the *RT PC VS FORTRAN User's Guide* for limitations). You may take programs written in VAX FORTRAN Version 3 and run them on your RT PC.

An additional advantage of IBM RT PC VS FORTRAN is that you have the ability to mix modes in creating an executable program. However, each separate unit compilation may use only a single mode.

You should note that some programs may produce different results when run on the RT PC compared to other machines because of differences in machine architecture, operating systems, or compiler implementations. These differences, along with the limitations of each mode, are noted in the *RT PC VS FORTRAN User's Guide*.

Methods of Presentation

In this manual:

- Italicized letters and words represent variables, for which user-supplied information is substituted. For example, the form of an integer edit-descriptor is I*w*, in which *w* is a nonzero unsigned integer constant. In an actual FORMAT statement, the specification might be I5 or I21.

- Brackets ([ ]) indicate that an item is optional. For example, the form of a unit specifier is "[UNIT =] *u*". In an actual input/output statement, the UNIT= keyword could be either used or omitted.

- An ellipsis (...) indicates that the preceding specification can, optionally, be repeated. For example, the form of an INTRINSIC statement is "INTRINSIC *name* [, *name*] ...", in which *name* is an intrinsic function name. An actual INTRINSIC statement could be coded as "INTRINSIC SIN" or "INTRINSIC SIN, COS" or "INTRINSIC SIN, COS, TAN" as needed.

- All other words, letters, and symbols are to be coded as shown.

- The general rule in FORTRAN for spaces (blanks) is that they have no significance in statements, and are used to improve readability. Space and blank are synonymous in this manual.

- The phrase "FORTRAN 66" refers to ANSI Standard FORTRAN 66.

- The phrase "FORTRAN 77" refers to ANSI Standard FORTRAN 77.
- Blue type indicates an enhancement to ANSI Standard FORTRAN 77.
- The symbol "✦" or "✧" in a section title indicates that related information specific to R1 or VX mode, respectively, is described under the same heading at the end of the chapter.

Terms and Concepts

A FORTRAN program is composed of characters that are grouped into lines. These lines are grouped into program units, which make up a program.

Lines consist of up to 72 characters. The first character of a line is considered to be in column 1, the second in column 2, and so on. The column position of characters in a line is often significant, especially with fixed-form input. Fixed-form input needs to be entered according to a predefined format, while free-form input (available in IBM mode) permits greater freedom in arranging input text of a program. A line can be a comment line, an initial line of a statement, or a continuation line of a statement.

"Comment lines" are lines of text that may be helpful in reviewing source program listings. They have no effect other than to be reproduced in program listings.

The "initial line" of a statement is the statement's first line, which may contain a statement label. A "statement label" tags a statement so that it can be referenced by other statements.

"Continuation lines" are used to continue a statement beyond its initial line. This may be done because a statement consists of more characters than will fit onto a single line, or may be done just to improve readability of a source program.

Data objects include constants and variables. A "constant" is a string of digits or other characters defining a value that does not change. A "variable" can have its value changed during program execution.

Variables and constants can have both a name and a data type. The "name" identifies that data object in a program. The "data type" of a data object defines its structural characteristics, features, and properties, such as the amount of storage it occupies, its range and precision, and in some cases the operations that can be performed on it. FORTRAN names can use default data types derived from a naming convention or the default can be overridden by explicit specifications.

A variable can be a single data object or an aggregate data object. There are two types of aggregate data objects — array variables and character variables. An "array variable" is a collection of data occupying consecutive storage units and can have multiple dimensions. A "character variable" represents string data and is a sequence of characters that can be accessed individually or collectively by a substring reference.

A variable can be given more than one name by a process of "association". There are several ways to associate names. The COMMON statement can be used to associate names among separate program units. The EQUIVALENCE statement can associate names in the same program unit. Variable names can also be associated through the argument-passing mechanism when functions or subroutine subprograms are referenced.

Names of variables have a "scope", which is the portion of a program in which the name is known or can be referenced. In general, a variable name has a scope that is local to the program unit in which it is defined. However, a main program name and subroutine and function subprogram names have a global scope. Names of variables in a common area are local to the program unit in which they are declared; the name of the common area itself has a global scope. Dummy arguments to a statement function have a scope that is local to statement function definition.

"Expressions" in statements combine data objects and operators to create new values. FORTRAN supports arithmetic, character, logical, and relational expressions. Mixed-type expressions are also permitted with well-defined rules for conversions between operands and for generation of results.

Statements may be categorized as either executable or nonexecutable. "Executable statements" specify action to be taken by a program; for example, assigning values to variables, evaluating expressions, affecting flow of execution, and performing data transmission. "Nonexecutable statements" describe the use or extent of the program unit, the characteristics of the data objects, data management, editing information, and statement functions.

"Specification statements" are nonexecutable statements that declare variables and symbolic constants. These include the type statements, which define the data types of variables; the DIMENSION statement, which defines the size of array variables; the COMMON and EQUIVALENCE statements, which associate variables; the PARAMETER statement, which gives a symbolic name to a constant; the EXTERNAL and INTRINSIC statements, which define attributes of other program units; and the DATA statement, which sets the starting value of data.

"Assignment statements" are executable statements that assign values to variables. The four types of assignment statements are arithmetic, character, logical, and statement label (ASSIGN) assignment statements. The ASSIGN statement assigns the value of a statement label or format label to an integer variable.

"Control statements" are executable statements that control the execution of a program, and include the IF, DO, CALL, RETURN, and GOTO statements, among others. The IF statement is conditional; it specifies a logical or arithmetic expression to be tested and the action to be taken depending on the result. The DO statement is used in a DO loop for repetitive execution of the same statement or statements. The CALL and RETURN statements are used for the execution of subroutine and function subprograms. Several forms of the GOTO statement are used for transfer of control within a program unit.

"Input/output statements" are executable statements that are used in transferring data between main storage and files and devices, and include the READ, WRITE, and PRINT statements, among others. The READ statement retrieves data from files and devices, and the WRITE and PRINT statements output data to files and devices.

A "statement function definition" is a single statement in a program unit containing an operation on dummy arguments. A "statement function reference" in the same program unit contains actual arguments and refers to

the statement function definition. The actual arguments are combined according to the statement function definition to yield a result that can be used in an expression.

Program units make up a FORTRAN program and include the main program and any number of subprograms. Subprograms fall into three categories: subroutine, function, and block data subprograms.

A "subroutine subprogram" is invoked by a CALL statement in another program unit and performs its own set of FORTRAN statements, optionally returning one or more parameters to the calling program unit. A "function subprogram" computes and returns a value in the context of an expression to the calling program unit. A "block data subprogram" is nonexecutable and is used to initialize (set the starting values of) data declared in common blocks.

User-defined subroutine and function subprograms are also known as "external procedures". Subroutine subprograms are declared with a SUBROUTINE statement and function subprograms are declared with a FUNCTION statement. Subroutine and function subprograms can have "arguments", which are variable names of input and output objects passed between the calling or referencing program unit and the subprogram being called or referenced. At the time a function or subroutine subprogram is declared, its "dummy arguments" are declared. At the time the subprogram is called or referenced, "actual arguments" are substituted for the dummy arguments.

A subroutine or function subprogram can have multiple entry points. An ENTRY statement allows execution to begin in a subroutine or function subprogram at statements other than the first executable statement.

Control is returned from a subroutine or function subprogram when an END or RETURN statement is encountered. In FORTRAN, an "alternate-return specification" is provided for subroutine subprograms. Instead of returning to the calling program unit at the statement following the CALL statement, a return can be made to an alternate statement.

FORTRAN supplies a comprehensive set of "intrinsic functions", which perform data type conversion and provide an extensive collection of arithmetic and transcendental functions.

Files can be external or internal, formatted or unformatted, and accessed sequentially or randomly, giving FORTRAN a powerful input/output capability.

External files are connected to an external device such as a disk file or a console. Internal files provide input/output to character variables.

Formatted files can be the subject of data conversion from internal storage representation to external character string representation and from external character string representation to internal storage representation. Format conversion may be performed via READ, WRITE, and PRINT statements. There is an extensive set of format specifications to control the form and layout of converted data. FORTRAN also has a list-directed input/output capability in which default formatting rules are applied in the data conversion process.

Chapter 2. Characters, Lines, Statements, and Execution Sequence

Character Set ✦ ✧

The FORTRAN character set consists of 27 letters, 10 digits, and 14 special characters.

A "letter" is one of these 27 characters:

A B C D E F G H I J K L M
N O P Q R S T U V W X Y Z $

A "digit" is one of these 10 characters:

0 1 2 3 4 5 6 7 8 9

An "alphanumeric character" is a letter or a digit.

The "special characters" used in FORTRAN are:

Special Character	Name
	Space (blank)
"	Double quote
$	Dollar sign
'	Apostrophe
(	Left parenthesis
)	Right parenthesis
*	Asterisk
+	Plus sign
,	Comma
-	Minus sign
.	Decimal or period
/	Slash
:	Colon
=	Equal sign

Figure 2-1. FORTRAN Special Characters

FORTRAN assumes that the characters have an order known as a collating sequence. The collating sequence from lowest to highest is:

- space (blank)
- double quote (")
- dollar sign ($)
- apostrophe (')
- left parenthesis (()
- right parenthesis ())
- asterisk (*)
- plus sign (+)
- comma (,)
- minus sign (-)
- decimal point or period (.)
- slash (/)

- digits from 0 to 9
- colon (:)
- equal sign (=)
- letters from A to Z.

Within the ordered sets of digits and letters in a sequence, the characters are contiguous; that is, there are no holes in the ordered sets.

RT PC VS FORTRAN uses the ASCII representation for characters. The ASCII representation for each character is listed in the *RT PC VS FORTRAN User's Guide*.

Spaces (blanks) have no meaning in FORTRAN programs, except:

- within string constants, Hollerith constants, and Hollerith fields
- in compiler directives, described in the *RT PC VS FORTRAN User's Guide*
- in column 6, when a space distinguishes between an initial line and a continuation line
- in counting the total number of characters per line and per statement.

Otherwise, spaces can and should be used freely to improve the layout and readability of your FORTRAN programs.

All characters used in statements must belong to the FORTRAN character set described in this section. For comment lines, character constants, and Hollerith fields, any of the printable ASCII characters can be used.

Lines ✦ ✧

FORTRAN has certain requirements pertaining to line length and to the information that can appear in specific columns of lines.

Source code can appear in columns 1-72 and has a limit of 1320 characters per statement, which corresponds to 20 lines by 66 columns.

Tabs are preset for eight-character advances.

FORTRAN acknowledges four types of lines: comment lines, initial lines of statements, continuation lines of statements, and compiler-directive lines.

Source input is accepted in either of two formats:

- fixed-form input format
- free-form input format

A program unit must be written in either fixed-form or free-form input format, but not both. The way lines are coded in both formats is described in the following sections.

Comment Lines ✦ ✧

A "comment line" often contains information that can be helpful in reviewing a program listing. Comment lines have no effect other than to be reproduced on program listings.

Comment lines can appear anywhere before a program unit's END statement, including before its first statement.

In Fixed-Form Input Format: A comment line is identified by a "C" in column 1 or by a blank line.

Comment lines can also appear between an initial line and its first continuation line and between two continuation lines.

Examples:

```
C     This line is a comment line.
C     The next line is blank ...

C     ... and is also a comment line.
```

In Free-Form Input Format: A comment line begins with a double quote (") in column 1. A comment line cannot follow a line that is continued and cannot itself be continued. Blank lines are not allowed in free-form input.

Examples:

```
"This line is a comment line.
"The following blank line will cause an error.

"This line is also a comment line.
```

Initial Lines of Statements

An "initial line" of a statement indicates the start of a statement line.

The initial line of a statement can have a "statement label", described in "Statement Labels" on page 2-11.

In Fixed-Form Input Format: An initial line of a statement is any non-comment line containing a space, a tab, or a 0 in column 6.

Examples:

```
C     These are two initial lines of
C     statements without statement labels.
C
      GOTO 999
      X=A

C     These are two initial lines of
C     statements with statement labels.
C
  379 GOTO 999
  485 X=A
```

In Free-Form Input Format: The first character of the statement text must be alphabetic. If a statement does not have a label, then the statement text must begin on the initial line. Blank lines are not allowed.

Continuation Lines ✦ ✧

"Continuation lines" are used to continue a statement beyond its initial line. This may be done for readability of the source program or because the statement may consist of more characters than will fit onto a single line.

In Fixed-Form Input Format: A continuation line is indicated by any character other than a 0 or a space in column 6. A continuation line cannot have a statement label, but columns 1-5 on a continuation line can contain characters (which are ignored). A statement can have up to 19 continuation lines.

When a character constant extends across two lines, its value is the same as if the character in column 7 of the continuation line abuts the last character on the preceding line.

Examples:

```
      PRINT *,'This string
     + uses two
     + continuation lines.'
```

The plus signs are in column 6 and indicate continuation lines. The output from this example is:

```
This string uses two continuation lines.
```

In Free-Form Input Format: A line to be continued is indicated by terminating the line with a minus sign (-). A comment line cannot be continued.

If the last character in a line is a minus sign, the compiler assumes it indicates continuation and discards it. If the last two characters in a line are minus signs, only the last one is taken as a continuation character, and the preceding one is preserved as a minus sign.

The statement text of continuation lines can start in any position. Up to 19 continuation lines are permitted in a single statement.

Compiler-Directive Lines

A "compiler-directive line" supplies information to the compiler to affect its action but does not result in executable code.

Spaces are significant in compiler-directive lines and are used to delimit keywords and file names. For more information on compiler directives, see the *RT PC VS FORTRAN User's Guide.*

In Fixed-Form Input Format: The first character of a compiler directive is entered in column 7 or after.

Example:

```
C     This compiler directive instructs the
C     compiler to include the body of the file
C     RASP.FOR into the program source code.
C
      INCLUDE (RASP.FOR)
```

In Free-Form Input Format: A compiler directive can start in any column.

Statements ✦ ✧

In general, statements must begin on new lines; that is, a statement cannot begin on the same line as another statement. The exception to this rule is the logical IF statement, described in "Logical IF Statements" on page 8-8.

In statements, spaces have no effect except within character constants and Hollerith constants, where they indicate blank characters.

An END statement must appear on an initial line of its own. No other statement in a program unit can have an initial line that looks like an END statement.

In Fixed-Form Input Format: Statements are written in columns 7-72 and can have up to 19 continuation lines.

Examples:

```
C     This is an assignment statement.
C
      A = 5.0

C     This is a CALL statement to a subroutine.
C
      CALL COLLECT (PAY,PHONE)

C     This is a logical IF statement.
C
      IF (DAY .EQ. 'FRIDAY') RETURN
```

In Free-Form Input Format: Statements are written in columns 1-80 and can have up to 19 continuation lines. The first character of a statement (after a label, if any) must be alphabetic. Multiple statements per line are not allowed.

A statement is terminated by an initial or continuation line that does not end with a minus sign.

Example:

```
"SAMPLE TEXT
  .
  .
10D=010.5
GO TO 56
150 A=B+C*(D+E**F+-
G+H-2.*(G+P))
 C=3.
 .
 .
```

Statements may be categorized as either executable or nonexecutable. "Executable statements" specify action to be taken by a program; for example, assigning values to variables, evaluating expressions, affecting flow of execution, and performing data transmission. "Nonexecutable statements" describe the use or extent of the program unit, the characteristics of the data objects, data management, editing information, and statement functions.

"Assignment statements" are executable statements that assign values to variables, and are described in Chapter 6, "Assignment Statements."

"Specification statements" are nonexecutable statements that define properties of variables, arrays, and functions. The following specification statements are described in Chapter 7, "Specification Statements" unless otherwise noted:

- BLOCK DATA (Chapter 9)
- COMMON
- DATA
- DIMENSION
- EQUIVALENCE
- EXTERNAL
- FUNCTION (Chapter 9)
- IMPLICIT
- INTRINSIC
- NAMELIST
- PARAMETER
- PROGRAM (Chapter 9)
- SAVE
- SUBROUTINE (Chapter 9)
- Type

"Control statements" are executable statements that control a program's flow of execution. The following control statements are described in Chapter 8, "Control Statements" unless otherwise noted:

- CALL (Chapter 9)
- CONTINUE
- DO
- END
- ENTRY (Chapter 9)

- Assigned GOTO
- Computed GOTO
- Unconditional GOTO
- Arithmetic IF
- Block IF-THEN-ELSE
  - Block IF
  - ELSEIF
  - ELSE
  - ENDIF
- Logical IF
- PAUSE
- RETURN (Chapter 9)
- STOP

"Input/output statements" are executable statements that are used in transferring data between main storage and input/output devices. The following input/output statements are described in Chapter 10, "Input and Output":

- BACKSPACE
- CLOSE
- ENDFILE
- FORMAT
- INQUIRE
- OPEN
- PRINT
  - format-directed
  - list-directed
- READ
  - format-directed
  - list-directed
  - namelist-directed
- WRITE
  - format-directed
  - list-directed
  - namelist-directed

Statement Labels

A "statement label" tags a statement so that it can be referenced by other statements. Statement labels are 1-5 digits in length and appear in columns 1-5 of initial lines of statements. At least one of the digits in a statement label must be nonzero.

Any statement can be labeled, but only executable statement labels and FORMAT statement labels can be referenced by other statements.

Each statement label in a program unit must be unique. Duplication of statement labels produces an error.

Examples:

```
C     This is a labeled FORMAT statement.
C
 123  FORMAT('The result is -- ',I5)

C     This is a DO block that uses a statement label.
C
      DO 110 ICON=1,100
          DESK (ICON)=0.0
 110  CONTINUE
```

Order of Statements ✦ ✧

The order in which statements appear in a program unit must comply with the following rules:

- If a PROGRAM statement is used, it must appear as the first noncomment statement of the main program. The first noncomment statement of a subprogram must be either a FUNCTION, SUBROUTINE, or a BLOCK DATA statement.
- FORMAT and ENTRY statements can appear anywhere between the first noncomment statement and the END statement.

- In ordering specification statements in a program unit, IMPLICIT statements must come before all other specification statements except PARAMETER statements.

- Any specification statement that defines the data type of a symbolic name must come before a PARAMETER statement that defines the symbolic name of a constant. Also, a PARAMETER statement that defines the symbolic name of a constant must come before any use of the name.

- PARAMETER and DATA statements may be interspersed with other specification statements, and the other specification statements must come before statement function definitions and executable statements.

- A NAMELIST statement declaring a namelist name must precede the use of the name in any input/output statement. It's placement is the same as that for other specification statements.

- Statement function definitions must come before executable statements.

- ENTRY statements can appear anywhere except within an IF block (between IF and ENDIF) and within a DO block (between a DO statement and the terminal statement of its DO loop).

- The final line of a program unit must be an END statement.

Figure 2-2 illustrates the manner in which you can intersperse statements and comment lines. Vertical lines separate statements that can be mixed. Horizontal lines separate statements that cannot be mixed.

For example, FORMAT statements can be mixed with statement function definitions and DATA statements. But statement function definitions cannot be mixed with executable statements.

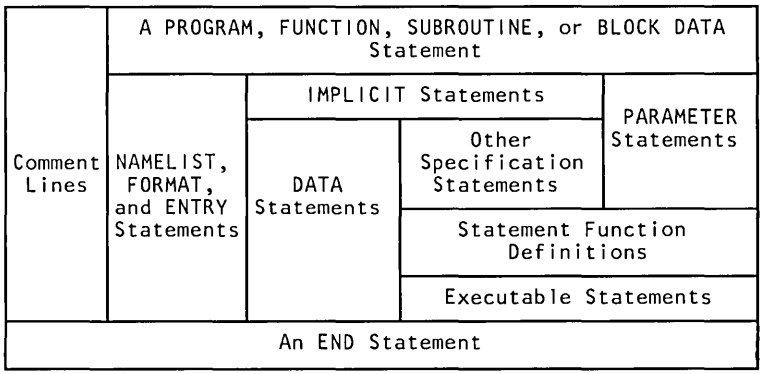


Figure 2-2. Order of Statements and Comments in a FORTRAN Program

Normal Execution Sequence and Control Transfer ✦ ✧

The "normal execution sequence" of a program unit occurs when its executable statements are processed in the order that they appear. Normal execution sequences begin with the first executable statement in the main program. When an external procedure is referenced, execution continues with the first executable statement following the FUNCTION, SUBROUTINE, or ENTRY statement in the referenced subprogram.

Normal execution sequences are not affected by nonexecutable statements, comment lines, or compiler directives appearing between executable statements.

"Control transfer" occurs when the normal execution sequence is altered. A control transfer is caused by:

- an unconditional, computed, or assigned GOTO statement
- an arithmetic IF statement
- a RETURN statement

- a STOP statement
- an input/output statement containing an error specifier or end-of-file specifier
- a call (using the CALL statement) that has an alternate-return specifier
- a logical IF statement containing any of the above as a subordinate statement
- a block IF or ELSEIF statement
- the last statement (if any) of an IF block or ELSEIF block
- a DO statement
- the terminal statement of a DO loop
- an END statement.

Function and subroutine subprograms cannot be invoked recursively; that is, a subprogram cannot call itself directly or be called by another subprogram that it has called. Also, subprograms cannot reference routines written in other languages.

R1 Mode Specifics

This section describes the instances in which R1 mode differs from IBM mode (the default mode).

Character Set

- The dollar sign ($) is not valid as a letter.
- These 26 lowercase letters are included in the character set:

  a b c d e f g h i j k l m
  n o p q r s t u v w x y z

- Special characters also include:

Special Character	Name
!	Exclamation point
%	Percent sign
&	Ampersand
<	Left angle bracket
>	Right angle bracket
\	Backslash
_	Underscore

- The compiler converts all lowercase characters to uppercase except those inside character constants.

Lines

- Free-form input format is not allowed.

Comment Lines

- A "c" in column 1 also indicates a comment line.

Continuation Lines

- Up to nine continuation lines are allowed per statement.

Statements

- Up to nine continuation lines are allowed per statement.

Order of Statements

- The NAMELIST statement is not allowed.

Normal Execution Sequence and Control Transfer

- Function and subroutine subprograms can be invoked recursively; that is, a subprogram can call itself directly or be called by another subprogram that it has called.
- Subprograms can also reference routines written in other languages (such as Pascal), which can be recursive.

VX Mode Specifics

This section describes the instances in which VX mode differs from IBM mode (the default mode).

Character Set

- The dollar sign ($) is not valid as a letter.
- These 26 lowercase letters are included in the character set:

  a b c d e f g h i j k l m
  n o p q r s t u v w x y z

- Special characters also include:

Special Character	Name
!	Exclamation point
%	Percent sign
&	Ampersand
<	Left angle bracket
>	Right angle bracket
_	Underscore

- The compiler converts all lowercase characters to uppercase except those inside character constants.

Lines

- Free-form input format is not allowed.
- Conditionally compiled lines are allowed.

  A "conditionally compiled line" is a line that is only compiled when the conditional compile switch is activated. Otherwise, the line is treated as a comment line.

  To indicate that a line is to be conditionally compiled, place an uppercase or lowercase D in column 1 of that line. If the conditionally compiled statement continues for more than one line, each continuation line also needs an uppercase or lowercase D in column 1, as well as a continuation indicator in column 6.

  To activate the conditional compile switch, specify the c+ command-line option when invoking the compiler. For a description of command-line options, see the *RT PC VS FORTRAN User's Guide*.

Example:

```
C     Suppose the conditional-compile option
C     is activated.
C
C     The values of the variables are written
C     out for debugging purposes.
C
D     WRITE(*,10) COUNT, VAL, I, J
D 10  FORMAT('Count = ',I4,'Val = ',F8.4,
D    +       'I = ',I4,'J = ',I4)
```

Comment Lines

- A "c" or "!" in column 1 also indicates a comment line.
- Comments can also be put on any line of a program unit by placing an exclamation point (!) in any column except column 6. The rest of the line is then considered a comment.

Example:

```
C     A=B+C          ! The two values are added.
C     PRINT *,A      ! The result is printed.
```

Continuation Lines

- Up to 99 continuation lines are allowed per statement.

Statements

- Up to 99 continuation lines are allowed per statement.

Order of Statements

- A NAMELIST statement declaring a namelist name can appear anywhere between the first noncomment statement and the END statement.
- Figure 2-2 is changed to:

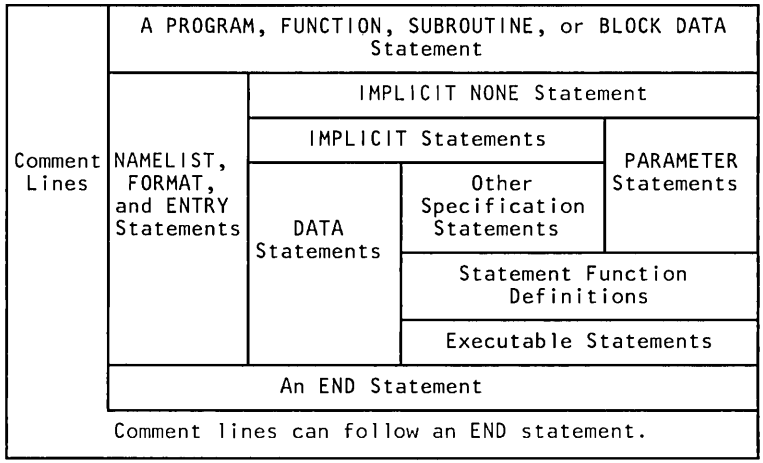


Figure 2-3. Ordering of Statements and Comments in a FORTRAN Program (VX mode)

Normal Execution Sequence and Control Transfer

- DO WHILE statements also cause control transfers.

Chapter 3. Data Types and Constants

"Data types" define the structural characteristics, features, and properties of data. FORTRAN has six data types:

- integer
- real
- double-precision
- complex
- character
- logical.

"Constants" are strings of digits or other characters defining values that do not change. The representation of a constant specifies both its data type and its value. A constant can have a logical, character, or arithmetic data type. An arithmetic constant can have an integer, real, double-precision, complex, or double-complex value.

Other ways to represent constants are also allowed and are described in "Constants ✧ ✧" on page 3-8.

An "unsigned" constant has no leading sign (+ or -). A "signed" constant has a leading sign. A constant that is "optionally signed" can be either signed or unsigned. Constants with integer, real, or double-precision values are optionally signed except where otherwise noted.

Data Type Rules

A symbolic name associated with a constant, variable, array, external function or statement function can have its data type specified in a type statement.

If no type statement is supplied for a program element, a default data type is supplied, which is determined by the first letter of the name. If the first letter of a name is I, J, K, L, M, or N, the integer data type is the default. Any other first letter defaults to the real data type. These data type defaults can be overridden either by explicit type statements or by IMPLICIT statements.

The data type of an array element is the same as that of the array.

The data type of a function name specifies the data type of the value that the function returns.

Intrinsic functions have data types that are specified in the chart in Appendix A, "Intrinsic Functions." Generic intrinsic functions do not have default data types. Their data types depend on the data types of their arguments.

External function references are given default data types based upon the first letters in their names, just like variables and arrays.

Integer Data Type

The "integer data type" is intended to represent elements of the infinite set of integers but can represent only a finite subset because of word size limitations in the computer.

The standard integer data type (INTEGER) occupies 1 word (4 bytes, or 32 bits) of storage and can represent values from -2,147,483,648 through 2,147,483,647.

Integers are represented internally in twos complement notation. As a consequence, the negative integer range is one integer greater than the positive integer range.

An "integer constant" is a string of decimal digits preceded by an optional sign and no decimal point.

You are able to choose the amount of storage an integer data type is to occupy:

INTEGER*2 occupies a halfword (2 bytes, or 16 bits) of storage and can assume values from -32768 through 32767.

INTEGER*4 is the same as the standard integer data type (INTEGER).

Real Data Type

The "real data type" is intended to represent the set of real values that comprise the continuum. Because of word size limitations in the computer, the real data type can represent only a finite subset of the infinite set of real numbers.

The standard real data type (REAL) occupies 1 word (4 bytes, or 32 bits) of storage. It can represent values from -3.402824E+38 through -1.175494E-38, 0, and from 1.175494E-38 through 3.402824E+38, with a precision of about seven decimal places.

The representation of real data allows for +infinity and -infinity and for indeterminate values. This is important when you are formatting such values for output. For more information on this subject, see Chapter 11, "Format Specifications."

A "real constant" has an optional sign, an integer part, a decimal point, a fractional part, and an optional exponent part. Both the integer and fractional parts are strings of digits. Either part can be omitted, but not both.

Examples:

```
3.14159       +2.236         -1.4142
 .7071         +.5            -.618034
5.            +8.            -6.
0.0            0.              .0

These are all valid real constants.
```

The optional exponent part of a real constant consists of the letter E followed by the optionally signed integer constant. The exponent part indicates a power of 10.

Examples:

```
E14       E+12       E-10       E0

These are all valid real exponents.
```

A real constant with an exponent part is the product of the constant preceding the E and 10 raised to the power indicated by the integer following the E.

Examples:

```
+7.52E-1       299793.5E3        20E-3

These are all valid real constants with exponent parts.
```

You are able to choose the amount of storage a real data type is to occupy:

REAL*4	is the same as the standard real data type (REAL).
REAL*8	is the same as the double-precision data type (DOUBLE PRECISION).

Double-Precision Data Type

The "double-precision data type" (DOUBLE PRECISION) is an extended-precision real data type and is used when single-precision data is inadequate.

The double-precision data type occupies 2 words (8 bytes, or 64 bits) of storage. It can represent values from -1.797693D+308 through -2.225074D-308, 0, and from 2.225074D-308 through 1.797693D+308, with a precision of about 16 decimal places.

A "double-precision exponent" consists of the letter D followed by an optionally signed integer constant. A double-precision exponent indicates a power of 10.

Examples:

```
D13      D+2      D-9      D0

These are all valid double-precision exponents.
```

A "double-precision constant" has an optional sign, a real or integer constant, and a double-precision exponent.

Examples:

```
+7.5D-1    299793.5D3    20D-3     -8D14

These are all valid double-precision constants.
```

Complex Data Type ✦ ✧

The "complex data type" is used to represent elements of the complex number domain. A complex number consists of an ordered pair of real numbers in which the first number represents the "real" part of the complex number and the second number represents the "imaginary" part.

The standard complex data type (COMPLEX) occupies 2 words (8 bytes, or 64 bits) of storage.

A "complex constant" is written as two integers or two real numbers separated by a comma and enclosed in parentheses. The first number represents the "real" part of the complex number, and the second number represents the "imaginary" part.

If the constants of the ordered pair representing the complex constant differ in precision, the constant of lower precision is converted to a constant of the higher precision. If the constants differ in type, the integer constant is converted to a real constant of the same precision as the original real constant.

Examples:

```
(1, 1)       (.707, -0.707)

(-1, 2.)     (-1.5E10, 2.6E-5)

(4.7D+2, -1.0D-5)

These are all valid complex constants.
```

You are able to choose the amount of storage a complex data type is to occupy:

COMPLEX*8 is the same as the standard complex data type (COMPLEX).

COMPLEX*16 occupies 4 words (16 bytes, or 128 bits) of storage.

Character Data Type ✦

The "character data type" (CHARACTER) is used to represent a string of characters. A character string can contain any of the printable ASCII characters, including spaces.

A "character constant" is a character string enclosed in apostrophes ('xxx'). The delimiting apostrophes are not part of the character string. An apostrophe that is to be part of a character string is indicated by two consecutive apostrophes ('').

The "length" of a character constant is the number of characters in the string, counting each set of two consecutive apostrophes as one character. The delimiting apostrophes are not counted in the string length as they are not part of the string.

Empty (null) character strings are not allowed.

Examples:

```
'x'    ' '    'FORTRAN'

'The time is 1 o''clock'

'''embedded apostrophes'''

These are all valid character constants.
The last two examples have embedded apostrophes.
```

Logical Data Type ✦ ✧

The "logical data type" represents boolean quantities and can take on only the values of true and false. The standard logical data type (LOGICAL) occupies 1 word (4 bytes, or 32 bits) of storage.

A "logical constant" represents a truth value and is either .TRUE. (for the true value) or .FALSE. (for the false value).

The abbreviations T and F (without the periods) may be used for .TRUE. and .FALSE., respectively, only for the initialization of logical variables or logical arrays in the DATA statement or in the explicit type statement. For use as input/output data, see "L — Logical Editing" on page 11-15.

You are able to choose the amount of storage a logical data type is to occupy:

LOGICAL*1 occupies 1 byte (8 bits) of storage.

LOGICAL*4 is the same as the standard logical data type (LOGICAL).

Constants ✦ ✧

In addition to integer, real, double-precision, complex, double-complex, character, and logical constants, Hollerith and hexadecimal constants are also allowed.

Hollerith Constants ✧

A "Hollerith constant" is a string of printable characters preceded by a character count and the letter "H". The form of a Hollerith constant is:

*w*Hc[*c*]...

w

specifies the number of characters in the string, which may not be less than 1 or greater than 255 (including spaces).

c

is a printable character.

Each character requires 1 byte of storage.

Hollerith constants have no data type. They can be used in place of character-string constants, and can also be used to initialize non-character variables in DATA statements.

Examples:

```
24H INPUT/OUTPUT AREA NO. 2
DATA IVAL /4HSAMP/
```

Hexadecimal Constants ✧ ✧

A "hexadecimal constant" is expressed as the letter "Z" followed by a string of hexadecimal digits, and can be used as a data or type initialization value for any type of variable or array. The form of a hexadecimal constant is:

Z*c*[*c*]...

c

is a number from 0–9 or a letter from A–F; each pair takes 1 byte of storage.

A hexadecimal constant specifies as much as 8 bytes of data. When the data type implies that the length of the constant is more than the number of digits specified, the leftmost digits have a value of 0. When the data type implies that the length of the constant is less than the number of digits specified, the constant is truncated on the left and a warning message is issued.

Examples:

`Z12ABF06C` represents the bit string `00010010101010111111000001101100`

`ZB1DFADE` represents the bit string `00001011000111011111101011011110`

The first four "0" bits are implied because an odd number of hexadecimal digits are written.

R1 Mode Specifics

This section describes the instances in which R1 mode differs from IBM mode (the default mode).

Complex Data Type

- The "real" and "imaginary" parts of a complex constant can be symbolic names as defined in a PARAMETER statement.
- The "double-complex data type" (DOUBLE COMPLEX) is also allowed, and is the same as COMPLEX*16.

Character Data Type

- A character constant can be delimited by double quotes ("xxx") as well as apostrophes ('xxx').
- If an apostrophe is to be part of a string that is delimited by apostrophes, or if a double quote is to be part of a string that is delimited by double quotes, it can be indicated either by two consecutive marks ('' or "") or by being placed after a backslash (\' or \").

  Note: Strings delimited by apostrophes require no special coding for inserting double quotes; strings delimited by double quotes require no special coding for inserting apostrophes.
- In determining the length of a character constant, two consecutive apostrophes are counted as one character when the string is delimited by apostrophes. Two consecutive double quotes are counted as one character when the string is delimited by double quotes.
- The compiler places a null character (\0) at the end of each character-string constant appearing outside a DATA statement. This is to ease communication with C routines.

- For compatibility with C-language usage, these "backslash escapes" are recognized in character strings:

Escape	Meaning
\n	New line
\t	Tab
\b	Backspace
\f	Form feed
\0	Null
\'	Apostrophe
\"	Double quote
\\	Backslash
\x	x, where x is any other character (\ is ignored)

Figure 3-1. Backslash Escapes

Logical Data Type

- The abbreviations T and F are not allowed.

Constants

- In addition to hexadecimal constants, binary and octal constants are also allowed.

  As much as 4 bytes of data can be specified by binary, octal, and hexadecimal constants. They are used to initialize logical or integer variables in a DATA statement, and are denoted by a letter followed by a string enclosed in apostrophes:

- If the letter is an uppercase or lowercase "B", then the string is binary, and only digits 0 and 1 are permitted.

- If the letter is an uppercase or lowercase "O", then the string is octal, with digits 0–7 permitted.

- If the letter is an uppercase or lowercase "Z" or "X", then the string is hexadecimal, with digits 0–9 and letters A–F permitted.

When the data type implies that the length of the constant is more than the number of digits specified, the constant is zero-filled on the left. When the data type implies that the length of the constant is less than the number of digits specified, the constant is truncated on the left and a warning message is issued.

Examples:

```
C     All three elements of A are initialized to 10.
      INTEGER A(3)
      DATA A/B'1010',O'12',Z'A'/
```

Hexadecimal Constants

• See "Constants" on page 3-12.

VX Mode Specifics

This section describes the instances in which VX mode differs from IBM mode (the default mode).

Complex Data Type

- The real and imaginary parts of a complex constant can be symbolic names as defined in a PARAMETER statement.
- The "double-complex data type" (DOUBLE COMPLEX) is also allowed, and is the same as COMPLEX*16.

Logical Data Type

- The abbreviations T and F are not allowed.
- Logical entities used in an arithmetic context are treated as integers.
- The LOGICAL*2 data type is also allowed, which occupies a halfword (2 bytes, or 16 bits) of storage.
- The "byte data type" (BYTE) is also allowed; like LOGICAL*1, it occupies 1 byte (8 bits) of storage and can contain the logical values .TRUE. or .FALSE.. Unlike LOGICAL*1, the byte data type can contain a single character or a value from -128 through 127 instead of a logical value.

Constants

- In addition to hexadecimal constants, octal constants are also allowed.

  You can use octal and hexadecimal constants wherever numeric constants are allowed. They are denoted by a string enclosed in apostrophes followed by a letter:

- If the letter is an uppercase or lowercase "O", then the string is octal, with digits 0–7 permitted.

- If the letter is an uppercase or lowercase "X", then the string is hexadecimal, with digits 0–9 and letters A–F permitted.

Leading zeros are ignored in octal and hexadecimal constants. You can specify up to 64 bits (22 octal digits, 16 hexadecimal digits).

Octal and hexadecimal constants are numeric constants that assume data types according to the following rules:

- An octal or hexadecimal constant used with a binary operator assumes the data type of the other operand.

  Example:

```
INTEGER*2 INUM
REAL*8 RNUM
INUM='23'O
RNUM*'1F2'X
```

  The constant '23'O assumes an integer data type with a 2-byte length, and the constant '1F2'X assumes a real data type with an 8-byte length.

- An octal or hexadecimal constant requiring a certain data type assumes that data type.

  Example:

```
REAL ARR(10)
ARR(1)=ARR('4'O)+1
ARR(1)=ARR('A'X)+1
```

  The constants '4'O and 'A'X both assume an integer data type with a 4-byte length.

- An octal or hexadecimal constant used as an actual argument assumes a 4-byte length and no data type.

Example:

```
CALL SUB('752'O)
X=FUNC('D1'X)
```

The constants '752'O and 'D1'X assume a 4-byte length and no data type.

– An octal or hexadecimal constant used in any other context assumes an INTEGER*4 data type.

Example:

```
K= .NOT. '14B'X
IF ('7772'O) 10,20,30
```

The constants '14B'X and '7772'O both assume an integer data type with a 4-byte length.

An octal or hexadecimal constant specifies as much as 8 bytes of data. When the data type implies that the length of the constant is more than the number of digits specified, the constant is zero-filled on the left. When the data type implies that the length of the constant is less than the number of digits specified, the constant is truncated on the left. A warning message is issued when a variable initialized in a DATA statement is truncated.

Hollerith Constants

- Hollerith constants can be used in numeric expressions, and assume data types according to the following rules:

  – A Hollerith constant used with a binary operator assumes the data type of the other operand.

Example:

```
INTEGER*2 INUM
REAL*8 RNUM
INUM=1HA
RNUM*2HXY
```

The constant 1HA assumes an integer data type with a 2-byte length, and the constant 2HXY assumes a real data type with an 8-byte length.

- A Hollerith constant requiring a certain data type assumes that data type.

  Example:

  ```
  REAL ARR(10)
  ARR(1)=ARR(1HA)+1
  ```

  The constant 1HA assumes an integer data type with a 4-byte length.

- A Hollerith constant used as an actual argument assumes no data type.

  Example:

  ```
  X=FUNC(3HNUM)
  ```

  The constant 3HNUM assumes no data type and has a 4-byte length.

- A Hollerith constant used in any other context assumes an INTEGER*4 data type.

  Example:

  ```
  K= .NOT. 1HA
  IF (6HSCALAR) 10,20,30
  ```

The constants 1HA and 6HSCALAR both assume an integer data type with a 4-byte length.

When the length of the constant is less than the length implied by the data type, spaces are appended to the constant on the right. When the length of the constant is greater than the length implied by the data type, the constant is truncated on the right. A warning message is issued if any non-space characters are truncated.

Hexadecimal Constants

- See "Constants" on page 3-14.

Chapter 4. Names, Arrays, and Substrings

This chapter includes FORTRAN rules for names, how to define and reference arrays, and a description of character variables and substrings of character variables.

Names ✦ ✧

A "name" (or "identifier") denotes a program unit, common block, variable, array, constant, argument, or statement function.

A name consists of up to six alphanumeric characters. The dollar sign ($) can also be used in a name (as its first character) and is treated as a letter. A name can also have embedded spaces, although they have no significance and are ignored by the compiler.

FORTRAN does not have any reserved words since the compiler recognizes keywords in context. To make your programs clear and readable, use names that are easily distinguishable from FORTRAN keywords.

Examples:

```
SHELL    $FIRST    S515
FIVE     KOUNT     H2O

These are all valid names.
```

When a user-defined name that has not previously appeared in a program unit is referenced in an executable statement, FORTRAN decides how to classify that name from the context in which it appears.

A name's data type is determined from its first letter. If a variable starts with an I, J, K, L, M, or N, it is given the integer data type. Variables with any other first letters, including $, are given the real data type. If an undeclared name appears in the context of a function reference, its data type is determined from its name in the same manner. In both cases, the default data types can be overridden with IMPLICIT statements. For a description of the IMPLICIT statement, see "IMPLICIT Statements — Assigning Default Data Types ✦ ✧" on page 7-15.

A name appearing in the context of a call to a subroutine has a symbol-table entry created for it. If an entry for that name already exists, its attributes are coordinated with those of the new entry. Inconsistencies such as a subroutine name used in the context of a function result in error messages.

You are encouraged to declare all names used in each program unit since it helps ensure that FORTRAN associates the proper definition with the name. Allowing FORTRAN to decide on defaults can sometimes result in logic errors, whose causes are difficult to locate.

Scope and Definition Status

The "scope" of a name is the portion of a program in which the name is known or can be referred to. A general rule about the scope of a name is that it is either local to a program unit or global to the entire program.

Names with a "global scope" include the name of the main program, the names of all subprograms (subroutine, function, and block data), and the names of common blocks. A name with a global scope can be used in any number of program units and still refer to the same thing, and therefore must be used in a single, consistent manner in a program. For example, a subroutine cannot have the the same name as a function subprogram or a common block. Similarly, two function subprograms cannot have the same name.

Names with a "local scope" include names of variables, arrays, constants, arguments, and statement functions. A name with a local scope is only known within a single program unit and therefore can be used in other program units with a different meaning. Within a program unit, of course, a name must have a consistent meaning.

A name with local scope can be used in the same compilation as the same name with global scope as long as the global name is not referenced within the program unit containing the local name. For example, there can be both a function subprogram named SYSTEM and a local variable named SYSTEM in another program unit. But the program unit containing the variable SYSTEM cannot reference the function named SYSTEM. If it is referenced, the compiler issues error messages.

Exceptions to the scope rules exist and are important to remember when creating FORTRAN programs. One of the exceptions to the scope rules is common block names. It is possible to reference a common block name (which has a global scope) in a program unit that contains a locally scoped item with the same name as the common block. This is allowed because common block names always appear in slashes (/), such as /COLD/, and therefore the compiler can distinguish between the names.

Another exception to the scope rules is names of dummy arguments of statement functions. The scope of names of dummy arguments of statement functions is limited to the statement function definition. Any other use of these names in the statement function or outside the statement function is not allowed. For example, if a dummy argument to a statement function has the same name as a function subprogram, that function subprogram cannot be referenced from within the statement function definition.

References to names of dummy arguments of a statement function from outside the statement function definition do not refer to those objects.

A third exception to the scope rules is that a name used as an implied DO control variable in a DATA statement or input/output statement has a scope that is local to the DATA statement or input/output statement.

The definition status of variables is described in Chapter 9, "Program and Subprogram Structure."

Array Declarations ✦

An "array" is an ordered set of data items with identical attributes, identified by a single name.

An "array declaration" specifies an array's name, which identifies the array in a program unit. An array declaration also indicates the number of dimensions it contains and the size of each dimension. It can also specify the data type of its elements. The form of an array declaration is:

```
aryname ( dim [ , dim ] ... )
```

aryname
is the name of an array.

dim
is a dimension declaration, described in "Dimension Declarations."

The number of dimensions an array has depends upon the number of dimension declarations given when the array is declared (the maximum number of dimension declarations is seven). Only one array declaration per array is allowed in a program unit. Duplicate declarations produce an error message.

Dimension Declarations

A "dimension declaration" defines the lower and upper bounds of a specific dimension in an array. The form of a dimension declaration is:

```
[ lowerbnd : ] upperbnd
```

lowerbnd
is an optional dimension-bound expression that defines the lower bound of the array.

upperbnd
is a dimension-bound expression the defines the upper bound of the array. The upper bound of the last dimension declaration can be an asterisk (*), which is described in "Kinds of Array Declarations."

A "dimension-bound expression" is an arithmetic expression, in which all variables, constants, and symbolic constant names are of the integer data type.

A dimension-bound expression cannot contain any function or array element references. Integer variables can appear in dimension-bound expressions only in adjustable array declarations, described in "Kinds of Array Declarations." If a symbolic constant name or variable in a dimension-bound expression is not of the default-implied integer data type, it must be specified in a type statement or an IMPLICIT statement as integer before it can be used in a dimension-bound expression.

A dimension bound can have a positive, negative, or zero value, but the upper bound must not be less than the lower bound. If only the upper bound is specified, the lower bound has a value of 1. An upper bound specified by an asterisk (*), described in "Kinds of Array Declarations," is always greater than or equal to the lower bound.

Kinds of Array Declarations

The three basic kinds of array declarations are constant array, adjustable array, and assumed-size array declarations.

A "constant array declaration" is a declaration in which all the dimension-bound expressions are integer constants.

An "adjustable array declaration" is a declaration in which the dimension-bound expressions contain integer variables. Adjustable arrays can be used as dummy arguments in subroutine and function subprograms. Variables which define the bounds of adjustable arrays must either be dummy arguments themselves or in common blocks.

An "assumed-size array declaration" is a declaration in which the upper bound of the last dimension is an asterisk (*). Like adjustable arrays, assumed-size arrays can be used as dummy arguments in subroutine and function subprograms. Procedures using assumed-size arrays circumvent any range checking that the FORTRAN compiler can perform.

Actual Array and Dummy Array Declarations

An "actual array declaration" immediately declares an array and must be a constant array declaration. Actual array declarations can be used in type statements, DIMENSION statements, and COMMON statements. These statements are described in Chapter 7, "Specification Statements."

A "dummy array declaration" defines a dummy argument in a subroutine or function subprogram. Dummy array declarations can be constant, adjustable, or assumed-size array declarations. They can only appear in subroutine and function subprograms and cannot appear in COMMON statements.

Examples:

```
C     These are five constant array
C     declarations in type statements.

C     This is a 100-element vector with
C     bounds of 1 and 100.
C
      INTEGER VECTOR(100)

C     This is a 20-element matrix of
C     five rows and four columns.
C
      REAL MATRIX(5,4)

C     This is a 256-element array with
C     bounds of 0 and 255.
C
      CHARACTER*2 CHARS(0:255)
```

```
C     This is a three-element array of
C     logical values.
C
      LOGICAL BOOLS(-1:+1)

C     This is a constant expression
C     declaring an eight-element array.
C
      REAL WOOD(2*4)

C     This is an adjustable array declaration
C     in a DIMENSION statement, which can
C     appear in a subprogram.  CHARS and LINES
C     must be integer and dummy arguments.
C     SCREEN must be a dummy argument.
C
      DIMENSION SCREEN(1:CHARS, 1:LINES)

C     This is an assumed-size array
C     declaration in a type statement,
C     which can appear in a subprogram.
C
      REAL VARIAB(5,*)
```

Referencing Array Elements — Array Subscripts

An "array subscript" references an element of an array. The form of an array subscript is:

(*subexpr* [, *subexpr*] ...)

subexpr
is a subscript expression. Note that the parentheses enclosing an array subscript's list of subscript expressions are required.

A "subscript expression" is any valid arithmetic expression. Subscript expressions can contain array element references and function references. If a subscript expression contains a function reference, the function must not change the value of any other subscript expression in the same sub-

script. If the array being referenced is not initialized in a DATA statement, the subscript expression can be non-integer with fractional parts truncated.

The value of a subscript expression must not be less than the lower bound or greater than the upper bound of that array's dimension. If the upper bound of the dimension is an asterisk (*), the subscript expression must not be greater than the size of the actual array.

The compiler does not check for improper indexing of arrays.

Examples:

```
      SCREEN(2,3)

      VARIAB(N+1, MAX(3,4))

      SCREEN(IPTR(N),JPTR(N))

These are all valid arrays with subscripts.
```

Array Storage Sequence

FORTRAN array data is organized in computer memory by column (column major order); therefore, the first subscript in a multi-dimensional array varies fastest.

Examples:

```
C     This is a one-dimensional array.
C
      DIMENSION A(-2:3)
C
C     The elements of this array are stored in this order:
C     A(-2), A(-1), A(0), A(1), A(2), and A(3).
```

```
C     This is a two-dimensional array.
C
      DIMENSION B(4,3)
C
C     The elements of this array are stored in this order:
C     B(1,1), B(2,1), B(3,1), B(4,1), B(1,2), B(2,2),
C     B(3,2), B(4,2), B(1,3), B(2,3), B(3,3), and B(4,3).

C     This is another two-dimensional array.
C
      DIMENSION C(-1:1,0:1)
C
C     The elements of this array are stored in this order:
C     C(-1,0), C(0,0), C(1,0), C(-1,1), C(0,1), and C(1,1).
```

A general formula for the relative offset of an arbitrary element, D(I,J), of a two-dimensional array declared as DIMENSION D(1:*m*,1:*n*) is:

relative offset = I + (J-1)***m*

The *m* and *n* are positive integers greater than or equal to 1. Note that the relative offset of the row 1, column 1 element [D(1,1)] in this array is 1.

Using Unsubscripted Array Names

Array names are generally followed by subscripts, but the exceptions in which the array name alone can be used are:

- a list of dummy arguments for a subroutine or function subprogram
- a COMMON statement when declaring that the array resides in that common block
- a type statement when the data type of the array is established
- an array declaration when the array dimensions are being established
- an EQUIVALENCE statement
- a DATA statement

- the list of actual arguments in a reference to an external procedure
- the list of an input/output statement if the array is not an assumed-size dummy array
- a unit identifier for an internal file in an input/output statement if the array is not an assumed-size dummy array
- the format identifier in an input/output statement if the array is not an assumed-size dummy array
- a SAVE statement.

Character Substrings ✧

A "character substring" is a portion of a character string and has a character data type. A character substring is identified by a substring name, which can be referenced and have values assigned to it. The form of a substring name is:

```
charvar ( [ start ] : [ finish ] )
```

charvar
is a character variable and can be an element of a character array.

start
finish
are optional substring expressions; *start* specifies the leftmost character position of the substring.

The values of *start* and *finish* must meet this condition (*length* is the length of the character variable or character array element):

$1 \leq start \leq finish \leq length$

If *start* is omitted in a substring name, the value 1 is used. If *finish* is omitted, the value *length* is used. Both *start* and *finish* can be omitted. If this is the case, the substring reference has the form *charvar*(:), which is equivalent to *charvar*.

The length of a character substring is *finish* - *start* + 1.

A "substring expression" is any integer expression and can contain array element references and function references. The restrictions that apply to array subscripts also apply to substring expressions.

Examples:

```
ROPEY(1:3)

THELOT(:)

ACHAR(5:5)

FOURCH(:4)

LINE(I)(1:20)

These are all valid substring expressions.
```

Incorrect usage of substrings usually results in compile-time errors. For example, substrings of single-character entities *do not* exist; therefore, this code is incorrect and produces a compile-time error:

```
CHARACTER TEXT(80)
.
.
TEXT(1:4) = 'WORD'
```

If the "1:4" is meant to refer to the first four characters of the data in TEXT, the proper use of this character substring is:

```
CHARACTER*80 TEXT
.
.
TEXT(1:4) = 'WORD'
```

R1 Mode Specifics

This section describes the instances in which R1 mode differs from IBM mode (the default mode).

Names

- The dollar sign ($) is not considered a letter and cannot appear in a name.
- The compiler makes no distinction between uppercase letters and lowercase letters. For example, the names FORTRAN, FORtRAN, fOrTrAn, and fortran are all equivalent as names in a FORTRAN program.

Array Declarations

- The maximum number of dimension declarations in an array declaration is 11.

VX Mode Specifics

This section describes the instances in which VX mode differs from IBM mode (the default mode).

Names

- A name consists of up to 31 alphanumeric, dollar sign ($), and underscore (_) characters and must not start with a digit or dollar sign.
- The dollar sign is not considered a letter.
- The compiler makes no distinction between uppercase letters and lowercase letters. For example, the names FORTRAN, FORtRAN, fOrTrAn, and fortran are all equivalent as names in a FORTRAN program.
- Although names can consist of up to 31 characters, names with a global scope are truncated to eight characters before being passed to the linker. Names with a global scope include program unit names and common block names. You need to use names unique to eight characters in these cases.

Character Substrings

- A "substring expression" is any valid arithmetic expression, with non-integer substring expressions being converted to integer values by truncating any fractional part before use.

Chapter 5. Expressions

"Expressions" combine data objects and operators to create new values, and consist of operators, operands, and parentheses. In FORTRAN, there are four types of expressions:

- arithmetic
- character
- relational
- logical.

Arithmetic Expressions ✧

An "arithmetic expression" is a numeric computation that generates a numeric value.

Arithmetic Operators

The arithmetic operators are:

Operator	Meaning
**	Exponentiation
/	Division
*	Multiplication
-	Subtraction or Negation
+	Addition or Identity

Figure 5-1. Arithmetic Operators

The **, /, and * operators are binary operators. The + and - operators can be unary or binary operators.

The ** operator has the highest precedence, then the * and / operators, and lastly the + and - operators. Whenever an ambiguity exists, the expressions are evaluated from left to right. Parentheses can be used to change the order of evaluation. Items in parentheses are considered first, from the innermost set of parentheses outward, no matter what operators are used.

Arithmetic Operands

An "arithmetic operand" can be a primary operand, a factor operand, a term operand, or an arithmetic expression.

"Primary operands" include:

- unsigned arithmetic constants
- symbolic names of arithmetic constants
- arithmetic variable references
- arithmetic array element references
- arithmetic function references
- arithmetic expressions enclosed in parentheses.

"Factor operands" are:

- primary
- primary ** factor.

A factor consists of one or more primaries separated by the exponentiation operator. The "primary ** factor" means that an expression such as 2**3**4 is interpreted as 2**(3**4).

"Term operands" are:

- factor
- term / factor
- term * factor.

A term consists of one or more factors separated by the multiplication (*) or division (/) operator. Factors are combined left to right.

"Arithmetic expressions" are:

- term
- +term or -term
- arithmetic expression + term
- arithmetic expression - term.

An arithmetic expression consists of a series of terms separated by addition (+) or subtraction (-) operators. A plus or minus sign can precede the first term in an expression. Terms are combined left to right.

Note that two consecutive operators form an incorrect expression; a**-b is not allowed, but a**(-b) is allowed.

Constant Expressions

A "constant expression" is either an arithmetic constant expression or an integer constant expression. Constant expressions are used in many FORTRAN constructs, especially in specification statements.

An "arithmetic constant expression" is an expression in which each primary is an arithmetic constant, the symbolic name of an arithmetic constant, or a constant expression enclosed in parentheses. Exponentiation is allowed only if the exponent is of the integer data type.

Examples:

```
5.0*2       2**31-1         2*(4.5, 9.8)
-16/4       3.141592/2      5**(3+2)

These are all valid arithmetic constant expressions.
```

An "integer constant expression" is an arithmetic constant expression in which each constant is of the integer data type.

Examples:

```
3*5        -10          4+5*(9-2)

These are all valid integer constant expressions.
```

Data Type Conversion Rules for Arithmetic Expressions ✦ ✧

When operands of mixed data types appear in an expression, FORTRAN performs implicit data type conversion on the operands according to well-defined rules. The data type of an expression is ultimately derived from the data types of its operands according to the following rules.

When the addition (+) operator or the subtraction (-) operator acts upon a single operand (that is, used as an unary operator), the data type of the result is the same as the data type of the operand.

When an arithmetic operator acts upon a pair of operands (that is, used as a binary operator), the data type of the result is as shown in either Figure 5-2 or Figure 5-3.

In these figures:

I indicates the integer data type.

R indicates the real data type.

D indicates the double-precision data type.

C indicates the standard complex data type (COMPLEX).

CD indicates the COMPLEX*16 data type.

RP indicates the real part of a complex number.

IP indicates the imaginary part of a complex number.

The functions REAL, DBLE, and CMPLX used in the figures are defined in Appendix A, "Intrinsic Functions."

Rules are given in the tables in the form of assignments. The data type of the result is indicated by the letter to the left of the equal sign (=), and the derivation of that result is given by the expression to the right of the equal sign.

Figure 5-2 shows the conversion rules for +, -, *, and / operations. For example, to obtain the rule for I1+C2 where I1 is an integer number and C2 is a complex number, find the I1 entry under X1 and the C2 entry across from X2. The conversion rule is:

C = CMPLX (REAL (I1), 0.0) + C2

The result is of the complex data type. The first operand is obtained by converting the integer number to a real number and then converting that real number to a complex number with the imaginary part equal to 0.0. The two complex numbers are then added.

For the -, *, and / conversion rules, replace the + in Figure 5-2 with the desired operator.

X1 \ X2	I2
I1	I = I1 + I2
R1	R = R1 + REAL(I2)
D1	D = D1 + DBLE(I2)
C1	C = C1 + CMPLX(REAL(I2),0.0)
CD1	CD = CD1 + DCMPLX(DBLE(I2),0.0)
X1 \ X2	R2
I1	R = REAL(I1) + R2
R1	R = R1 + R2
D1	D = D1 + DBLE(R2)
C1	C = C1 + CMPLX(R2,0.0)
CD1	CD = CD1 + DCMPLX(DBLE(R2),0.0)
X1 \ X2	D2
I1	D = DBLE(I1) + D2
R1	D = DBLE(R1) + D2
D1	D = D1 + D2
C1	CD = DCMPLX(DBLE(RP1) + DBLE(IP1)) + DCMPLX(D2,0.0)
CD1	CD = CD1 + DCMPLX(D2,0.0)
X1 \ X2	C2
I1	C = CMPLX(REAL(I1),0.0) + C2
R1	C = CMPLX(R1,0.0) + C2
D1	CD = DCMPLX(D1,0.0) + DCMPLX(DBLE(RP2),DBLE(IP2))
C1	C = C1 + C2
CD1	CD = CD1 + DCMPLX(DBLE(RP2),DBLE(IP2))
X1 \ X2	CD2
I1	CD = DCMPLX(DBLE(I1),0.0) + CD2
R1	CD = DCMPLX(DBLE(R1),0.0) + CD2
D1	CD = DCMPLX(D1,0.0) + CD2
C1	CD = DCMPLX(DBLE(RP1),DBLE(IP1) + CD2
CD1	CD = CD1 + CD2

Figure 5-2. Conversion Rules for +, -, *, and / Operations

Figure 5-3 shows the conversion rules for exponentiation (**) operations. It is interpreted in the same manner as Figure 5-2.

X1 \ X2	I2
I1	I = I1 ** I2
R1	R = R1 ** I2
D1	D = D1 ** I2
C1	C = C1 ** I2
CD1	CD = CD1 ** I1

X1 \ X2	R2
I1	R = REAL(I1) ** R2
R1	R = R1 ** R2
D1	D = D1 ** DBLE(R2)
C1	C = C1 ** CMPLX(R2,0.0)
CD1	CD = CD1 ** DCMPLX(DBLE(R2),0.0)

X1 \ X2	D2
I1	D = DBLE(I1) ** D2
R1	D = DBLE(R1) ** D2
D1	D = D1 ** D2
C1	CD = DCMPLX(DBLE(RP1),DBLE(IP1)) ** DCMPLX(D2,0.0)
CD1	CD = CD1 ** DCMPLX(D2,0.0)

X1 \ X2	C2
I1	C = CMPLX(REAL(I1),0.0) ** C2
R1	C = CMPLX(R1,0.0) ** C2
D1	CD = DCMPLX(D1,0.0) ** DCMPLX(DBLE(RP2),DBLE(IP2))
C1	C = C1 ** C2
CD1	CD = CD1 ** DCMPLX(DBLE(RP2),DBLE(IP2))

X1 \ X2	CD2
I1	CD = DCMPLX(DBLE(I1),0.0) ** CD2
R1	CD = DCMPLX(DBLE(R1),0.0) ** CD2
D1	CD = DCMPLX(D1,0.0) ** CD2
C1	CD = DCMPLX(DBLE(RP1),DBLE(IP1) ** CD2
CD1	CD = CD1 ** CD2

Figure 5-3. Conversion Rules for ** Operations

Some of the entries in Figure 5-3 show what happens when a complex argument is raised to a complex power. In these cases, the value of the expression is the principal value, which is determined by the formula:

X1 ** X2 = EXP (X2 * LOG (X1))

The EXP and LOG are the exponential and natural logarithm intrinsic functions described in Appendix A, "Intrinsic Functions."

Except for values raised to an integer power in mixed data type expressions, the operand that differs from the data type of the result is converted to the data type of the result according to the rules given in the previous tables. The operator then acts upon a pair of operands of the same data type.

When a primary is raised to an integer power, the integer does not need to be converted.

Data Type Conversion Rules for Integers of Different Size

In expressions containing mixtures of INTEGER*2 and INTEGER*4 (or INTEGER) variables, the operands with the smaller fields are promoted to the size of the larger fields.

If you assign the result of an expression to a variable with a smaller field, an undefined result is produced if the value of the result exceeds the range of values allowed for that variable.

Note also that many FORTRAN statements and functions specifically require arguments of the standard integer data type. In such cases, you cannot use arguments of INTEGER*2 size.

Some of the FORTRAN intrinsic bit-manipulation routines also require arguments of specific integer sizes.

Example:

```
      INTEGER*2 I,K
      INTEGER*4 L,J
      I=32767
      J=2
      K=I+J
      L=I+J
      WRITE(*,100) K,L
  100 FORMAT(2I10)
      STOP
      END
```

This program prints the numbers -32767 and 32769 and illustrates that INTEGER*2 variables are promoted to INTEGER*4 variables when those expressions are evaluated. If this were not the case, the value -32767 would have been printed twice. The program also illustrates what happens when an expression is assigned to a variable whose data type range is not large enough to hold the value.

Integer Division

If an integer operand is divided by another integer operand, the result is not the strict mathematical quotient. Instead, the quotient is obtained by truncating toward 0. For example, the integer 1 divided by the integer 2 results in a quotient of 0. Similarly, the integer -8 divided by the integer 3 results in a quotient of -2.

Character Expressions

A "character expression" operates on character strings and generates character values. The simplest forms of character expressions are:

- character constants
- character variables
- character array element references
- character substring references
- character function references
- character expressions enclosed in parentheses.

The only character operator is // (concatenation). Concatenation joins two strings in the order specified, thereby forming one string whose length is equal to the sum of the lengths of the two strings. The form of the character concatenation is:

```
x1 // x2
```

This produces a new value that is the value of *x1* concatenated on the right with the value of *x2*.

Example:

```
C     This code prints "Big Ben".
C
      CHARACTER*3 FNAME,LNAME
      DATA FNAME,LNAME /'Big','Ben'/
      PRINT *,FNAME // ' ' // LNAME
```

Dummy arguments to procedures can be character strings whose length is specified by an asterisk (*). The asterisk designates the string as an assumed-size character string whose length is determined at the time an actual string argument is associated with that dummy argument.

A character string expression involving concatenation of such a string argument cannot be passed as an actual argument to any procedure, nor can it appear in the format specification of an input/output statement or as an item in the input/output list of an input/output statement.

Examples:

```
C     This code results in two compile-time errors
C     because of incorrect usage of string expressions.
C
      PROGRAM ERROR
      CHARACTER*20 TRIAL
      TRIAL='Will this go?'
      CALL SUBERR(TRIAL)
      STOP
      END
```

```
      SUBROUTINE SUBERR(STRING)
      CHARACTER*(*) STRING
C     The first error occurs in this CALL statement.
      CALL WILD(STRING//' Nope!')
C     The second error occurs in this WRITE statement.
      WRITE(*,10) STRING//' Nope again!'
 10   FORMAT(A)
      RETURN
      END
```

These examples illustrate two of the three character string expression restrictions. The errors can be corrected by assigning the concatenated strings to new character variables and using the new variables in the CALL and WRITE statements.

Relational Expressions

"Relational expressions" compare arithmetic expression values or character expression values and yield logical values. The relational operators are:

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

Figure 5-4. Relational Operators

Arithmetic Relational Expressions

An "arithmetic relational expression" shows a relationship between arithmetic operands. The form of an arithmetic relational expression is:

```
e1 relop e2
```

e1
e2
are arithmetic operands.

relop
is a relational operator.

Only the .EQ. (equal to) and .NE. (not equal to) operators are allowed for operands of the complex data type.

If the operands are of different data types, the relational expression is treated as if it were in the form:

```
( ( e1 ) - ( e2 ) ) relop 0
```

The 0 (zero) has the same data type as the expression.

Comparison of a double-precision value with a complex value is not allowed.

Character Relational Expressions

A "character relational expression" shows a relationship between character operands. The form of a character relational expression is:

```
e1 relop e2
```

e1
e2
are character expressions.

relop
is a relational operator.

The character order, or collating sequence, used by FORTRAN is described in "Character Set ✦ ✧" on page 2-1. RT PC VS FORTRAN uses the ASCII representation for character data. The .EQ. (equal to) and .NE. (not equal to) operators do not use the ordering; they do a bit comparison.

If the operands in a character relational expression have different lengths, the shorter operand is padded on the right with spaces until the lengths are equal.

Logical Expressions ✧

A "logical expression" operates on logical values and generates a logical result. The simplest forms of logical expressions are:

- logical constants
- logical variable references
- logical array element references
- logical function references
- relational expressions.

Other logical expressions expand upon these simple forms by using parentheses and these logical operators:

Logical Operator	Meaning
.NOT.	Logical Negation
.AND.	Logical Conjunction
.OR.	Inclusive Disjunction
.EQV.	Logical Equivalence
.NEQV.	Logical Nonequivalence

Figure 5-5. Logical Operators

The relative precedence of logical operators is (from highest to lowest):

1. .NOT.
2. .AND.
3. .OR.
4. .EQV. and .NEQV.

The .AND. and .OR. operators are binary operators and appear between their operands. The .NOT. operator is an unary operator and appears before its operand. Operators of equal precedence associate left to right.

Examples:

`(A .AND. B .AND. C)` is equivalent to `((A .AND. B) .AND. C)`

`(.NOT. A .OR. B .AND. C)` is equivalent to
`((.NOT. A) .OR. (B .AND. C))`

Two .NOT. operators cannot appear next to each other, but an adjacent .AND. and .NOT. is allowed. For example, (A .AND. .NOT. B) is an allowable expression.

Precedence of Operators

When arithmetic, relational, and logical operators appear in the same expression, their relative precedence is (from highest to lowest):

1. arithmetic operators
2. relational operators
3. logical operators.

Expression Programming Errors

Any variable, array element, or function referenced in an expression must be defined by the time of the reference. Integer variables must be defined with an arithmetic assignment statement rather than a statement label (ASSIGN) assignment statement. If a character string or substring is referenced in an expression, all the referenced characters should be defined by the time of the reference.

Unpredictable results occur if you divide by zero or raise a zero value to a zero or negative power. Unpredictable results also occur if you raise a negative value to a real or double-precision power.

It is a programming error when a function reference within a statement changes any other object in that statement.

If a function reference causes an actual argument to the function to become defined, unpredictable results occur if that object is referenced anywhere else in the statement containing the function reference.

Example:

```
C     This program is faulty because the variable A in
C     the second assignment statement of the main
C     program ERROR is changed by the function reference,
C     producing unpredictable results.

      PROGRAM ERROR
      A = 4.
      X = FUNC(A) + A
      END

      FUNCTION FUNC(X)
      X = X + 1
      FUNC = X
      RETURN
      END
```

R1 Mode Specifics

This section describes the instances in which R1 mode differs from IBM mode (the default mode).

Data Type Conversion Rules for Arithmetic Expressions

- The "CD" in Figure 5-2 and Figure 5-3 also indicates the double-complex (DOUBLE COMPLEX) data type.

VX Mode Specifics

This section describes the instances in which VX mode differs from IBM mode (the default mode).

Arithmetic Expressions

- The term "numeric" includes logical data because logical data is treated as integer data when used in an arithmetic context.

Data Type Conversion Rules for Arithmetic Expressions

- The "CD" in Figure 5-2 and Figure 5-3 also indicates the double-complex (DOUBLE COMPLEX) data type.

Logical Expressions

- The simplest forms of logical expressions also include integer constants, integer variable references, integer array element references, integer function references, and integer expressions enclosed in parentheses.
- When a logical operator operates on integer elements (which is possible under VX mode only), the logical operation is carried out bit-by-bit on the corresponding bits of the internal (binary) representation of the integer elements. The resulting data type is integer.
- When a logical operator combines integer and logical values (which is possible under VX mode only), the logical value is first converted to an integer value, then the operation is carried out as for two integer elements. The resulting data type is integer.
- The logical operator ".XOR." is also allowed, and is the same as the .NEQV. operator.

Chapter 6. Assignment Statements

"Assignment statements" compute values that are then assigned to data objects. Since FORTRAN does not require that variables be declared before their use, assignment statements can cause the allocation of storage for data objects. There are four types of assignment statements:

- arithmetic
- logical
- statement label (ASSIGN)
- character.

Arithmetic Assignment Statements ✦ ✧

An "arithmetic assignment statement" evaluates an arithmetic expression and assigns the result to a variable. The form of an arithmetic assignment statement is:

```
var = expr
```

var
is a variable or an array element name whose data type is integer, real, double-precision, or complex.

expr
is an expression that is compatible with the data type of *var*.

If the data type of *var* and the *expr* are not compatible, the value of *expr* is automatically converted to the data type of *var* according to the following table:

var	expr
Integer	INT(expression)
Real	REAL(expression)
Double-Precision	DBLE(expression)
Complex	CMPLX(expression)

Figure 6-1. Data Type Conversion for Arithmetic Assignment Statements

The functions in the "Value Assigned" column in Figure 6-1 are generic intrinsic functions described in Appendix A, "Intrinsic Functions."

Example:

```
      COMPLEX C
      REAL X,Y
      INTEGER I,J,K
      C = (1.0,2.0)
      Y = 5.
      J = 1
      K = 2
 5    X = C*Y
10    I = (X*J) / (2.0*Y) * (K**2)
15    X = 4.0 * DATAN(1.0D+0)
```

In statement 5, the expression is evaluated as a complex data type. However, X is of the real data type, so it is assigned the real part of the complex expression.

In statement 10, the expression is evaluated as a real data type. However, since I is declared as an integer, the expression is truncated toward 0 before the assignment.

Statement 15 illustrates the preferred method of obtaining π with double-precision accuracy. But since X is only a single-precision variable, approximately nine decimal places of precision are lost during the assignment.

Keep in mind that the evaluation of an integer expression containing INTEGER*2 variables occurs after the variables are promoted to INTEGER*4, and the result for the expression is of the INTEGER*4 data type.

The assignment of an integer expression to an integer variable of a larger data type occurs without problems. However, assignment of an integer expression to an integer variable of a shorter data type may produce erroneous results if the value of the expression is outside the range of the data type of the variable. Error messages are not produced if this occurs.

Example:

```
      INTEGER*2 I
      INTEGER*4 J
      J=32770
      I=J
      WRITE(*,100) I,J
100   FORMAT(2I10)
      STOP
      END
```

The execution of this program causes the numbers -32766 and 32770 to be printed. This occurs because 32770 is outside the range of numbers that can be represented by the INTEGER*2 data type.

Logical Assignment Statements ✧

A "logical assignment statement" assigns the value of an expression to a logical variable. The form of a logical assignment statement is:

logvar = logexpr

logvar
is a logical variable.

logexpr
is a logical expression that must evaluate to either true (.TRUE.) or false (.FALSE.).

If a logical variable's data type is LOGICAL, LOGICAL*1, or LOGICAL*4, its logical expression is automatically converted to the correct data type before the assignment.

Examples:

```
      LOGICAL TELLME, NONO, BIG

      TELLME = .TRUE.

      NONO = .FALSE.

      BIG = (I .GT. 100)

These are all valid logical assignment statements.
```

Statement Label (ASSIGN) Assignment Statements

A "statement label assignment statement" assigns the value of a format label or a statement label to an integer variable. The form of statement label assignment statement is:

ASSIGN *label* **TO** *intvar*

label
is a format label or a statement label.

intvar
is an integer variable that must have a data type of either INTEGER or INTEGER*4.

Examples:

```
*     H, I, and J have been declared as INTEGER.
*
      ASSIGN 424 TO H
      ASSIGN 675 TO I
      ASSIGN 905 TO J

These are all valid statement label assignment statements.
```

Note that ASSIGN 100 TO LAB is not the same as LAB = 100. The former assigns the address of the statement labeled 100 to LAB. The latter assigns the value 100 to LAB.

Statement label assignment statements are used with assigned GOTO statements, described in Chapter 8, "Control Statements," and as format specifiers in formatted input/output, which are described in Chapter 10, "Input and Output."

Character Assignment Statements

A "character assignment statement" evaluates a character expression and assigns the result to a character variable, character array element, or character substring. The form of a character assignment statement is:

charvar = *charexpr*

charvar
is a character variable.

charexpr
is a character expression.

The character field being used on the left side of the statement should not appear on the right side of the statement. If it does, the results may be undefined.

The left and right sides of a character assignment statement can have different lengths. If the left side is longer, the right side is extended to the right with spaces until the lengths are equal. If the left side is shorter, a substring equal to the length of the left side is taken from the right side starting at position 1.

Only as much of the right side need be defined as is necessary to define the left side. For example, consider this program fragment:

```
      CHARACTER MARK*4, BILL*8

      MARK = BILL
```

This assignment of BILL to MARK requires that the substring BILL(1:4) be defined since that is enough to define MARK. The rest of BILL, BILL(5:8), does not have to be defined.

If the left side of a character assignment statement is a substring reference, the right side is assigned only to the substring. The definition status of the character positions not specified on the left side does not change.

Examples:

```
      CHARACTER*10 STR10
      CHARACTER*20 STR20
10    STR10 = 'abcdefghij'
20    STR10 = 'Too long of a string'
30    STR20 = '123456789ABCDEFGHIJK'
40    STR10 = STR20(1:10)
50    STR20(2:11) = STR20(1:10)
```

Statement 10 causes the entire 10-character string to be assigned to the character variable STR10.

Statement 20 causes the characters "Too long o" to be assigned to STR10. The rightmost characters in the character constant are ignored.

Statement 40 assigns only the first 10 characters of STR20 to STR10.

Statement 50 gives undefined results because character positions in the expression are being assigned values.

R1 Mode Specifics

This section describes the instances in which R1 mode differs from IBM mode (the default mode).

Arithmetic Assignment Statements

- The *var* is a variable or an array element name whose data type is integer, real, double-precision, complex, or double-complex.
- With the inclusion of the double-complex data type, Figure 6-1 is changed to:

var	expr
Integer	INT(expression)
Real	REAL(expression)
Double-Precision	DBLE(expression)
Complex	CMPLX(expression)
Double-Complex	DCMPLX(expression)

Figure 6-2. Data Type Conversion for Arithmetic Assignment Statements (R1 and VX Modes)

VX Mode Specifics

This section describes the instances in which VX mode differs from IBM mode (the default mode).

Arithmetic Assignment Statements

- The *var* is a variable or an array element name whose data type is integer, real, double-precision, complex, or double-complex.
- With the inclusion of the double-complex data type, Figure 6-1 is changed. See Figure 6-2 on page 6-8.

Logical Assignment Statements

- If a logical variable's data type is LOGICAL, LOGICAL*1, LOGICAL*2, or LOGICAL*4, its logical expression is automatically converted to the correct data type before the assignment.

Chapter 7. Specification Statements

"Specification statements" are nonexecutable statements that define properties of variables, arrays, and functions. The specification statements described in this chapter are:

- Type
- DIMENSION
- COMMON
- DATA
- PARAMETER
- IMPLICIT
- EXTERNAL
- INTRINSIC
- SAVE
- EQUIVALENCE
- NAMELIST

The BLOCK DATA, FUNCTION, PROGRAM, and SUBROUTINE specification statements are described in Chapter 9, "Program and Subprogram Structure."

For proper ordering of specification statements in a program unit, see "Order of Statements ✦ ✧" on page 2-11.

Type Statements — Declaring Data Types ✦

"Type statements" specify the data types of user-defined names and can either confirm or override the default data types. Type statements can also define the dimensions of an array.

A type statement can contain a user-defined name for a variable, array, dummy argument, external function, or statement function. When a type

statement does contain one of these names, the data type of that name is defined for the program unit containing the type statement. In any given program unit, a user-defined name can appear in only one type statement.

A type statement can confirm the data type of an intrinsic function, but it is not required to do so. Type statements cannot contain main program names or subroutine subprogram names.

You can initialize data in any form of type declaration statement by placing values bounded by slashes (/) immediately after the symbolic name of the variable or array to be initialized.

Arithmetic Type Statements ✦ ✧

An "arithmetic type statement" declares arithmetic data objects. The form of an arithmetic type statement is:

```
type  var [ *l ] [ /clist/ ] [ , var [ *l ] [ /clist/ ] ] ...
```

type
is INTEGER, INTEGER*2, INTEGER*4, REAL, REAL*4, REAL*8, DOUBLE PRECISION, COMPLEX, COMPLEX*8, or COMPLEX*16.

var
is a variable name, array name, dummy argument name, function name, or array declarator.

l
is a data type length specifier, which must be one of the acceptable lengths for the data type being declared. This specification overrides the length attribute that the statement implies, and assigns a new length to the specified item. If you specify a data type length specifier with an array declarator, the data type length specifier goes immediately after the array name.

clist
is a list of constants, as in a DATA statement.

Examples:

```
C     These are declared integer variables.
C
      INTEGER CLOCK, HANDS(2), TIME(24)

C     These are declared real and
C     double-precision variables.
C
      REAL RADIO, K101, VARBLS(10, 10, 5)
      DOUBLE PRECISION TWOS(50), TWICE, SECOND

C     These are declared complex data items.
C
      COMPLEX FUNK, ROCK, BACH(48)

C     This is a declared real variable
C     with initialization.
C
      REAL*8  ARRAY(10)/5*0.0, 5*1.0/

These are all valid arithmetic type statements.
```

Character Type Statements ✦ ✧

A "character type statement" declares character data objects. The form of a character type statement is:

CHARACTER [**n* [,]] *var* [**n*] [*/clist/*] [, *var* [**n*] [*/clist/*]] ...

var
is a variable name, array name, dummy argument name, or an array declarator.

n
is the length in characters of a character variable or a character array element.

clist
is a list of constants, as in a DATA statement.

The length *n* must be an unsigned integer constant in the range 1–500, or it can be a constant expression that is enclosed in parentheses and has a value of 1–500. (The length of *n* can be from 1–32767 if the n command-line option is specified. Command-line options are described in the *RT PC VS FORTRAN User's Guide.*) If the name is being defined as a dummy argument or if a data type is being established for later use in a PARAMETER statement, the length *n* can also be specified by an asterisk enclosed in parentheses (*).

The length *n* following the type name CHARACTER is the default length for any name in the list that does not have its length specified explicitly. In the absence of a length specification, the default length is 1. A length immediately following a variable or array element overrides the default length for that item only. For an array, the length specifies the length of each element of that array.

A dummy argument defined as CHARACTER*(*) cannot be used as an actual argument to a procedure if it is concatenated in a character string expression. However, a symbolic name of a constant can be used in such a case.

Examples:

```
CHARACTER FLIP*10, FLOP*20

CHARACTER WILD(15)*20

CHARACTER*80 LINE(24)

CHARACTER*(10*20) LSTR
```

```
CHARACTER*(*) VARBLE

CHARACTER NAME*10/'ABCDEFGHIJ'/
```

These are all valid character type statements.

Logical Type Statements ✦ ✧

A "logical type statement" declares logical data objects. The form of a logical type statement is:

type var [**l*] [*/clist/*] [, *var* [**l*] [*/clist/*]] ...

type
is LOGICAL, LOGICAL*1, or LOGICAL*4.

var
is a variable name, array name, dummy argument name, function name, or an array declarator.

l
is a data type length specifier of 1 or 4. This specification overrides the length attribute that the statement implies, and assigns a new length to the specified item. If you specify a data type length specifier with an array declarator, the data type length specifier goes immediately after the array name.

clist
is a list of constants, as in a DATA statement.

Examples:

```
LOGICAL SONG

LOGICAL BOOLS(40)
```

```
      LOGICAL*4  BLACK, WHITE

      LOGICAL FLAG /.TRUE./
```

These are all valid logical type statements.

DIMENSION Statements – Declaring Array Dimensions ✦ ✧

A "DIMENSION statement" specifies the number of dimensions in a user-defined array. The form of a DIMENSION statement is:

DIMENSION *var* (*dim*) [, *var* (*dim*)]

var (*dim*)
is an array declarator that has the form:

name (*d* [, *d*] ...)

name
is the user-defined name of the array.

d
is a dimension declarator.

The number of dimensions in an array is the number of dimension declarators in its array declarator. The maximum number of dimensions for an array is seven. Array and dimension declarators are described in Chapter 4, "Names, Arrays, and Substrings."

Examples:

```
DIMENSION FORTH (10,5:15,0:99)

DIMENSION AXIS (6)
```

The first example declares an array of three dimensions. The first dimension has a range of 1-10, the second dimension has a range of 5-15, and the third dimension has a range of 0-99. The second example declares a single-dimension array of six elements.

COMMON Statements — Declaring Common Blocks ✦ ✧

A "COMMON statement" defines a common block and its contents. Common blocks are storage areas that allow variables to be shared among program units. The form of the COMMON statement is:

COMMON [/ [*cname*] /] *nlist* [[,] [/ [*cname*] /] *nlist*]] ...

cname
is a common block name.

nlist
is a list of variable names, array names, and array declarators, all separated by commas.

Dummy argument names and function names must not appear in COMMON statements.

All variables and arrays appearing in an *nlist* that follows a common block name are declared to be in that common block. If the *cname* is omitted, all elements appearing in the *nlist* are declared to be in the blank common block.

A common block name can appear more than once in one or more COMMON statements in the same program unit. The *nlist* following each successive appearance of the same common block name is treated as a continuation of the list for that common block name. Elements in an *nlist* of a common block are allocated storage in the order of their declaration.

The size of a common block is equal to the number of bytes of storage needed to hold all the elements in that common block.

A named common block with the z command-line option specifies the named common area to be allocated at execution time. (Command-line options are described in the *RT PC VS FORTRAN User's Guide*.) Note that this type of common block cannot be initialized at compile time.

Examples:

```
INTEGER MONTH,DAY,YEAR
COMMON /DATE/ MONTH,DAY,YEAR
```

The variables MONTH, DAY, and YEAR reside in the common block DATE. DATE is 12 bytes in length with MONTH occupying the first 4 bytes, DAY occupying the second 4 bytes, and YEAR occupying the last 4 bytes.

```
INTEGER STAND(2,4)
REAL X(4)
COMMON /CBLOCK/ STAND,X
```

The arrays STAND and X reside in the common block CBLOCK. CBLOCK is 48 bytes in length with the arrays stored in this order: STAND(1,1), STAND(2,1), STAND(1,2), STAND(2,2), STAND(1,3), STAND(2,3), STAND(1,4), STAND(2,4), X(1), X(2), X(3), and X(4).

```
PROGRAM EXAMPL
REAL MATRIX(3,3)
COMMON /ABC/ MATRIX
.
.
END

SUBROUTINE SUB
REAL COL1(3),COL2(3),COL3(3)
COMMON /ABC/ COL1, COL2, COL3
.
.
END
```

The common block ABC is defined in the main program and contains the two-dimensional array MATRIX. In the subroutine subprogram SUB, the common block ABC is referenced, but with different variables. COL1 references MATRIX(1,1), MATRIX(2,1), and MATRIX(3,1). COL2 references MATRIX(1,2), MATRIX(2,2), and MATRIX(3,2). COL3 references MATRIX(1,3), MATRIX(2,3), and MATRIX(3,3).

DATA Statements – Declaring Initial Values ✦ ✧

A "DATA statement" is a nonexecutable statement that statically initializes data variables. DATA statements can be interspersed with PARAMETER statements and other specification statements. The form of a DATA statement is:

DATA *nlist /clist/* **[[,]** *nlist /clist/* **] ...**

nlist
is a list of variables, arrays, array element names, substring names, and implied DO lists to be initialized.

When using array element names, subscript expressions must be integer expressions containing only integer constants or names of

integer constants. They must not contain variables, array elements, or function references.

clist

is a list of constants used to initialize the *nlist*. The constants can be preceded by an integer constant repeat-factor and an asterisk (*).

Examples:

```
5*3.14159       3*'Help'       100*0

These are constants preceded by an integer
constant repeat-factor and an asterisk.
```

The number of values in a *clist* must equal the number of variables or array elements in the accompanying *nlist*. An array in an *nlist* is equivalent to a list of all the elements in that array in the order they are stored. Array elements and substrings can be indexed by integer constant expressions.

The data type of each element in a *clist* must be the same as the data type of the corresponding variable or array element in the accompanying *nlist*. If necessary, the *clist* constant is converted to the data type of the *nlist* object according to the data type conversion rules for arithmetic assignment statements, which are described in Chapter 6, "Assignment Statements."

A DATA statement can initialize any variable, array element, or substring unless it is:

- a dummy argument
- an object in a blank common block
- an object that is associated with an object in a blank common block
- a variable in a function subprogram whose name is the same as that of the function or one of its alternate-entry points.

A hexadecimal constant can be used to initialize any type of variable or array element.

If a hexadecimal constant initializes a complex data type, one constant is used that initializes both the "real" and "imaginary" parts, and the constant is not enclosed in parentheses. If the constant is smaller than the length (in bytes) of the entire complex entity, zeros are added on the left. If the constant is larger, the leftmost hexadecimal digits are truncated.

A Hollerith constant can be used to initialize a noncharacter variable or array element.

A logical variable or logical array can be initialized with T instead of .TRUE. and F instead of .FALSE..

Character items can be initialized by character data. Each character constant initializes exactly one variable, one array element, or one substring. If a character constant contains more characters than the item it initializes, the additional rightmost characters in the constant are ignored. If a character constant contains fewer characters than the item it initializes, the additional rightmost characters in the item are initialized with blank characters. (Each character represents 1 byte of storage.)

Objects can be initialized only once in a program unit. DATA statements in block data subprograms can only initialize objects in named common blocks.

Examples:

```
C     Some variables are declared.
C
      REAL FIRST,SECOND
      INTEGER EGG,NOG,WIGWAM
      COMPLEX WEIRD(10)
      DOUBLE PRECISION VECT(5)

C     The reals are initialized.
C
      DATA FIRST,SECOND /1.0,2.0/

C     The integers are initialized.
C
      DATA EGG/12/,NOG/24/,WIGWAM/25/
```

```
C     Two elements of the complex array
C     are initialized.
C
      DATA WEIRD(2),WEIRD(5)
     + / 2 * (90.0,0.0) /

C     All of the double-precision array
C     is initialized.
C
      DATA VECT /0.,1.,2.,3.,4./

These are all valid DATA statements.
```

Implied DO Loops in DATA Statements

A DATA statement can incorporate a form of the DO loop (described in "DO Statements — Loop Control ✧" on page 8-10) that provides a concise static initialization of array variables. This "implied DO loop" is an indexing specification similar to a DO loop but does not specify the word DO and has a list of data elements rather than a set of statements as its range. The form of an implied DO loop is:

(*dlist* , *dovar* = *first* , *last* [, *inc*])

dlist
: is the implied DO list, which contains a list of array element names and can contain embedded implied DO lists.

dovar
: is the name of an integer variable known as the implied DO variable.

first
last
inc
: are integer constant expressions; *first* is the starting value of *dovar*, *last* is its ending value, and the optional *inc* is the value by which *dovar* is incremented for each loop. If *inc* is omitted, its default value of 1 is used.

The *first*, *last*, and *inc* expressions establish the implied DO loop's iteration count (the number of passes through the loop), which is established just as for a DO loop. Unlike the DO loop, the iteration count for the implied DO loop must be greater than 0. The *first*, *last*, and *inc* expressions can contain implied DO variables of other implied DO lists whose range includes this implied DO list.

The *dlist* is the range of an implied DO loop. When an implied DO loop appears in a DATA statement, each item in the *dlist* is specified once and the value of *dovar* is appropriately incremented for each repetition of the implied DO loop.

The implied DO variable can have the same name as a variable in the program unit containing the DATA statement since there is no conflict of such names.

Examples:

```
C     Some large arrays are declared.
C
      INTEGER PRIMES(1000)
      INTEGER UPRTRI(20,20)
      REAL MATRIX(25,80)

C     The large arrays are initialized with DATA
C     statements containing implied DO loops.
C
      DATA (PRIMES(I), I=1,1000) /1000*1/
      DATA ((MATRIX(J,K), J=1,25) K=1,80) /2000*1.0/

C     This DATA statement initializes the
C     upper triangle of the array UPRTRI.
C
      DATA ((UPRTRI(I,J), J=I,20), I=1,20) /210*0/

These are all valid implied DO loop initializations.
```

PARAMETER Statements — Making Symbolic Associations

A "PARAMETER statement" associates a constant value with a symbolic name. The use of the symbolic name thereafter is equivalent to the use of the constant value. The form of a PARAMETER statement is:

PARAMETER (*name* = *expr* [, *name* = *expr*] ...)

name
is the symbolic name being defined. (Symbolic names must be defined in a type or IMPLICIT statement unless the default implied type is desired.)

expr
is a constant expression being associated with the symbolic name.

Examples:

```
PARAMETER (TODAY = 'Friday')

PARAMETER (PI = 3.141592654)
```

In these examples, the character expression "Friday" is associated with the symbolic name TODAY and the arithmetic expression 3.141592654 is associated with the symbolic name PI. These expressions can now be referenced by their respective symbolic names throughout the program.

IMPLICIT Statements – Assigning Default Data Types ✦ ✧

An "IMPLICIT statement" overrides FORTRAN's default data type rules and establishes a new default data type for variables. The form of the IMPLICIT statement is:

IMPLICIT *type* (*letterlist*) [, *type* (*letterlist*)] ...

type
is INTEGER, INTEGER*2, INTEGER*4, REAL, REAL*4, REAL*8, LOGICAL, LOGICAL*1, LOGICAL*4, DOUBLE PRECISION, COMPLEX, COMPLEX*8, COMPLEX*16, or CHARACTER [*n]. (The *n* is the size of the character data type. If *n* is not specified, a value of 1 is assumed.)

letterlist
is a list of letters that, when they begin symbolic names, cause the symbolic names to receive the data type and size of *type.*

IMPLICIT statements only apply to the program units in which they appear and do not change the data type of any intrinsic function. Implicit data types can be overridden or confirmed for any specific user-defined name if that name appears in a type statement. Similarly, an explicit data type in a FUNCTION statement takes precedence over an IMPLICIT statement. Also, a character data type length can be overridden by a later data type specification.

A program unit can have more than one IMPLICIT statement, but all IMPLICIT statements must precede all other specification statements.

Examples:

```
C     All names beginning with A are declared
C     to be of the integer data type.
C
      IMPLICIT INTEGER (A)
```

```
C     All names beginning with Q, X, Y, or Z
C     are declared to be of the complex data type.
C
      IMPLICIT COMPLEX (Q, X-Z)

C     All names beginning with A, B, C,
C     D, E, F, G, H, P, Q, R, S, T, U,
C     V, W, X, Y, or Z are declared to be
C     of the double-precision data type.
C
      IMPLICIT REAL*8 (A-H, P-Z)

These are all valid IMPLICIT statements.
```

EXTERNAL Statements – Declaring External or Dummy Procedures

An "EXTERNAL statement" specifies the name of an external or dummy procedure, and allows the name to be used as an actual argument in a subroutine or function reference. The form of an EXTERNAL statement is:

EXTERNAL *procname* [, *procname*] ...

procname
is the name of an external procedure, dummy procedure, or block data subprogram.

If an intrinsic function name appears in an EXTERNAL statement, that name becomes the name of an external procedure and the corresponding intrinsic function can no longer be called from that program unit.

Statement function names cannot appear in EXTERNAL statements, and user-defined names can appear only once per EXTERNAL statement.

Examples:

```
C     This illustrates how to pass the name
C     of a function in an argument list to
C     a function subprogram.
C
      SUBROUTINE ROOTS
      EXTERNAL POS,NEG
      .
      .
      IF (I.LT.0) THEN
         X = QUAD(A,B,C,NEG)
      ELSE
         X = QUAD(A,B,C,POS)
      ENDIF
      .
      .
      RETURN
      END

      FUNCTION QUAD(A,B,C,FUNCT)
      .
      .
      VAL = FUNCT(A,B,C)
      .
      .
      RETURN
      END

      FUNCTION POS(A,B,C)
      .
      .
      RETURN
      END

      FUNCTION NEG(A,B,C)
      .
      .
      RETURN
      END
```

INTRINSIC Statements – Declaring Intrinsic Functions

An "INTRINSIC statement" specifies the name of an intrinsic function, and allows the name to be used as an actual argument in a subroutine or function reference. The form of an INTRINSIC statement is:

INTRINSIC *name* [, *name*] ...

name
is the name of an intrinsic function.

A name can appear only once in an INTRINSIC statement and in only one INTRINSIC statement per program unit. If a name appears in an INTRINSIC statement, it cannot appear in an EXTERNAL statement.

All names used in INTRINSIC statements must be system-defined intrinsic functions. For a list of intrinsic functions, see Appendix A, "Intrinsic Functions."

If a specific name of an intrinsic function is used as an actual argument in a program unit, that name must be declared in an INTRINSIC statement in that program unit. If a generic function name of an intrinsic function appears in an INTRINSIC statement, that function still retains its generic properties.

The intrinsic functions (in IBM mode) that cannot be used as actual arguments are:

- the type-conversion functions — INT, IFIX, IDINT, FLOAT, SNGL, REAL, DBLE, CMPLX, DCMPLX, ICHAR, and CHAR
- the lexical relationship functions — LGE, LGT, LLE, and LLT
- the functions for choosing largest or smallest values — MAX, MAX0, AMAX1, DMAX1, AMAX0, MAX1, MIN, MIN0, AMIN1, DMIN0, AMIN0, and MIN1.

Example:

```
C     This illustrates how to pass the
C     name of an intrinsic function to
C     a subroutine subprogram.

      INTRINSIC SIN
      CALL DOIT(0.5,SIN,X)
      .
      .
      END

      SUBROUTINE DOIT(A,F,Y)
      REAL A,F
      .
      .
      Y = F(A) + F(3.14159)
      .
      .
      END
```

SAVE Statements — Retaining Definition Status

A "SAVE statement" retains the definitions of data objects after control is returned from the procedures that define them. A data object that is specified in a SAVE statement in a subroutine or function subprogram remains defined after exiting the subprogram. The form of a SAVE statement is:

SAVE [*aname* [, *aname*] ...]

aname
is a common block name enclosed in slashes, a variable name, or an array name.

A name can appear only once in a SAVE statement. The names of dummy arguments, procedures, and data objects appearing in common blocks cannot appear in SAVE statements.

A SAVE statement appearing without an associated *aname* saves every data object in the program unit that can be saved. It has the same effect as if every data object in the program unit appeared in the SAVE statement.

Specifying a common block name in a SAVE statement is the same as saving all the elements in that common block. A common block mentioned in a SAVE statement must be mentioned in a SAVE statement in every subprogram in which that common block appears. A SAVE statement has no effect in the main program and is therefore optional.

Examples:

```
C     Everything in the subprogram is saved.
C
      SAVE

C     Some variables are saved.
C
      SAVE DIMES, NICKELS, PENNIES

C     All of common block STAMPS is saved.
C
      COMMON /STAMPS/ S(50)
      SAVE STAMPS

These are all valid SAVE statements.
```

```
C     This code shows one use of the SAVE statement.
C     The program prints the integers 1-10.  If the
C     SAVE statement was not used, a 1 would be printed
C     and the rest of the values would be undefined.
      .
      .
      DO 10 I=1,10
         CALL X(I)
10    CONTINUE
      STOP
      END

      SUBROUTINE X(ICOUNT)
      SAVE K
      IF (ICOUNT.EQ.1) THEN
         K = 1
      ELSE
         K = K + 1
      ENDIF
      PRINT *,K
      END
```

Definition status and the use of the SAVE statement are described further in Chapter 9, "Program and Subprogram Structure."

EQUIVALENCE Statements — Sharing Storage Between Elements ✦ ✧

An "EQUIVALENCE statement" specifies that two or more variables or arrays are to share the same storage. The EQUIVALENCE statement does not cause any automatic data type conversion if the shared elements are of different data types. The form of an EQUIVALENCE statement is:

EQUIVALENCE (*nlist*) [, (*nlist*)] ...

nlist

is a list of at least two variable names, array names, constant array element names, or character substring names.

When using array element names, subscript expressions must be integer expressions containing only integer constants or names of integer constants. They must not contain variables, array elements, or function references.

Argument names cannot appear in EQUIVALENCE statements.

Equivalencing character variables with noncharacter variables produces unpredictable results.

An EQUIVALENCE statement specifies that the storage sequences of the elements that appear in the *nlist* have the same first storage location. Two or more variables are considered to be associated if they refer to the same actual storage. Thus an EQUIVALENCE statement causes its list of variables to become associated.

An EQUIVALENCE statement cannot specify that the same storage location is to appear more than once.

Example:

```
      REAL R, S(10)
C     An error occurs in the next statement.
      EQUIVALENCE (R, S(1)), (R, S(5))
```

An error occurs because the variable R is forced to appear in two memory locations — S(1) and S(5).

An EQUIVALENCE statement cannot specify that consecutive array elements be stored out of sequential order.

Example:

```
      REAL R(10), S(10)
C     An error occurs in the next statement.
      EQUIVALENCE (R(1), S(1)), (R(5), S(6))
```

An error occurs because the statement attempts to associate R(5) and S(6) after associating R(1) and S(1). This means that the array R is stretched, which is not allowed.

Names of dummy arguments cannot appear in EQUIVALENCE statements. Also, if a variable name is also a function name, that name cannot appear in an EQUIVALENCE statement.

When EQUIVALENCE statements are used in conjunction with COMMON statements, further restrictions apply. An EQUIVALENCE statement cannot associate storage elements in different common blocks. An EQUIVALENCE statement can extend a common block by adding storage elements after the common block, but not before the common block.

Example:

```
      COMMON /MASSES/ R(10)
      REAL S(10)
C     An error occurs in the next statement.
      EQUIVALENCE (R(1), S(10))
```

An error occurs because the EQUIVALENCE statement tries to extend the common block by adding storage before the start of the block. When R(1) and S(10) are associated, S(1) is nine storage elements before the start of the common block.

NAMELIST Statements — Specifying Names ✧ ✧

A NAMELIST statement specifies a list of variables or array names and associates that list with a unique namelist name. The namelist name is used in the namelist-directed READ and WRITE statements to define the variables or arrays that are to be read or written. The form of a NAMELIST statement is:

NAMELIST / *name* / *list* [/ *name* / *list*] ...

name
is a namelist name, and cannot be the same as a variable or array name.

list
is a list of variable or array names, separated by commas.

The list of variable or array names belonging to a namelist name ends with a new namelist name enclosed in slashes (/) or with the end of the NAMELIST statement. A variable name or an array name may belong to one or more namelist lists.

Neither a dummy variable nor a dummy array name may appear in a namelist list.

The NAMELIST statement must precede any statement function definitions and all executable statements. A namelist name must be declared in a NAMELIST statement and may be declared only once. The name may appear only in input/output statements.

The NAMELIST statement declares a name *name* to refer to a particular list of variables or array names. Thereafter, the forms READ (*unit,name*) and WRITE (*unit,name*) are used to transmit data between the unit *unit* and the variables specified by the namelist name *name*.

The rules for input/output conversion of namelist data are the same as the rules for data conversion described in Chapter 5, "Expressions." The namelist data must be in a special form, described in "Namelist Input Data" on page 10-34.

R1 Mode Specifics

This section describes the instances in which R1 mode differs from IBM mode (the default mode).

Type Statements — Declaring Data Types

- Initializing data in a type declaration statement by placing values bounded by slashes (/) after the symbolic name of the variable is not allowed.
- A storage class type statement is also allowed, which declares a variable to be STATIC or AUTOMATIC. The form of a storage class type statement is:

```
[ STATIC ] var [ , var ] ...

or

[ AUTOMATIC ] var [ , var ] ...
```

STATIC
indicates there is exactly one copy of the datum, and its value is retained between calls.

AUTOMATIC
indicates there is one copy of each variable declared automatic for each invocation of the procedure. This is the default.

var
is a variable name, array name, or array declarator, but may not be a dummy argument name or function name.

Storage class type statements must follow these rules:

- Variables explicitly declared as AUTOMATIC cannot appear in DATA statements.
- Variables explicitly declared as STATIC can appear in DATA statements.
- Variables explicitly declared as AUTOMATIC or STATIC cannot appear in SAVE statements.
- Variables cannot be declared as AUTOMATIC or STATIC more than once.
- AUTOMATIC variables can appear in EQUIVALENCE statements but they cannot be equivalenced with STATIC variables or variables in COMMON blocks.
- STATIC variables cannot be equivalenced with AUTOMATIC variables or variables in COMMON blocks.
- Both explicitly defined AUTOMATIC variables and STATIC variables cannot appear in COMMON statements.
- The same variable cannot be defined or saved as both AUTOMATIC and STATIC.

Arithmetic Type Statements

- The *type* is INTEGER, INTEGER*2, INTEGER*4, REAL, REAL*4, REAL*8, DOUBLE PRECISION, COMPLEX, COMPLEX*8, COMPLEX*16, or DOUBLE COMPLEX.
- The constant list *clist* is not allowed.

Character Type Statements

- The constant list *clist* is not allowed.
- The n command-line option is not allowed.

Logical Type Statements

- The constant list *clist* is not allowed.

DIMENSION Statements — Declaring Array Dimensions

- The maximum number of dimensions for an array is 11.

COMMON Statements — Declaring Common Blocks

- Elements in a common block must be either all or none of the character data type.
- The z command-line option is not allowed.

DATA Statements — Declaring Initial Values

- A hexadecimal constant can only be used to initialize an integer variable, logical variable, or array element.
- A binary or octal constant can be used to initialize an integer variable, logical variable, or array element.
- The abbreviations T and F are not allowed.

PARAMETER Statements — Making Symbolic Associations

- Symbolic names of constants may be used to replace one or both parts of a complex constant.

IMPLICIT Statements — Assigning Default Data Types

- The *type* can also be DOUBLE COMPLEX, UNDEFINED, STATIC, or AUTOMATIC.

  UNDEFINED turns off the automatic data typing mechanism and instructs the compiler to issue a diagnostic for each variable used that does not appear in a type statement. Specifying the u- command-line option is equivalent to beginning each procedure with UNDEFINED.

  STATIC indicates there is exactly one copy of the datum, and its value is retained between calls.

  AUTOMATIC indicates there is one copy of each variable declared automatic for each invocation of the procedure. This is the default.

  Static and automatic variables in IMPLICIT statements must follow these rules:

  - An implicitly defined automatic variable cannot appear in a DATA statement.
  - A SAVE statement or explicit STATIC type statement overrides an IMPLICIT AUTOMATIC statement.
  - An explicit AUTOMATIC type statement overrides an IMPLICIT STATIC statement.
  - A COMMON statement overrides a previous IMPLICIT statement; that is, both implicitly defined automatic and static variables can appear in a COMMON statement and lose their automatic or static attribute.

EQUIVALENCE Statements — Sharing Storage Between Elements

- Single subscripts are permitted for multiple-dimension arrays. All missing subscripts are considered to be equal to 1. A warning message is given for each incomplete subscript.

NAMELIST Statements — Specifying Names

- The NAMELIST statement is not allowed.

VX Mode Specifics

This section describes the instances in which VX mode differs from IBM mode (the default mode).

Arithmetic Type Statements

- The *type* is INTEGER, INTEGER*2, INTEGER*4, REAL, REAL*4, REAL*8, DOUBLE PRECISION, COMPLEX, COMPLEX*8, COMPLEX*16, or DOUBLE COMPLEX.

Character Type Statements

- The n command-line option is not allowed.

Logical Type Statements

- The *type* is LOGICAL, LOGICAL*1, LOGICAL*2, or LOGICAL*4.
- The data type length specifier **l* can be 1, 2, or 4.

DIMENSION Statements — Declaring Array Dimensions

- A "VIRTUAL statement" is also allowed, which has the same effect as a DIMENSION statement. The form of a VIRTUAL statement is:

VIRTUAL *var* (*dim*) [, *var* (*dim*)]

See "DIMENSION Statements — Declaring Array Dimensions ✧ ✧" on page 7-6 for variable descriptions.

COMMON Statements — Declaring Common Blocks

- Elements in a common block must be either all or none of the character data type.
- The z command-line option is not allowed.

DATA Statements — Declaring Initial Values

- An octal constant can also be used to initialize any type of variable or array element.
- The abbreviations T and F are not allowed.
- Nonprintable characters such as line feed and form feed can be assigned to character variables. This is done by giving the *nlist* item a CHARACTER*1 (or CHARACTER) data type and making the *clist* item an integer in the range 0-255. Integers in hexadecimal radix form can also be used. For a list of nonprintable characters and their numerical representations, see the *RT PC VS FORTRAN User's Guide.*

Examples:

```
      CHARACTER STRING*8, SVECT(6)*10,
     + CARRIAGE_RETURN*1, LINE_FEED*1

      DATA STRING /'Old Rope'/

      DATA SVECT /6 * 'Attached'/

      DATA CARRIAGE_RETURN /13/

C     The integer in this DATA statement is
C     in hexadecimal radix form.
      DATA LINE_FEED /Z0A/
```

This program sends a form feed to the printer:

```
      PROGRAM FF
      CHARACTER*1 FORM_FEED
      DATA FORM_FEED /12/
C     The integer could also be 'Z0C'.
      OPEN(4,FILE='/dev/lp')
      WRITE(4,'(A)') FORM_FEED
      STOP
      END
```

PARAMETER Statements — Making Symbolic Associations

- Symbolic names of constants may be used to replace one or both parts of a complex constant.
- The *expr* can contain these functions:
  - IAND, IOR, NOT, IEOR, ISHFT, LGE, LGT, LLE, and LLT with constant operands
  - CHAR with a constant operand
  - MIN, MAX, ABS, MOD, ICHAR, NINT, DIM, DPROD, CMPLX, CONJG, and IMAG with constant operands.

IMPLICIT Statements — Assigning Default Data Types

- The *type* can also be LOGICAL*2, DOUBLE COMPLEX, or NONE.

  NONE turns off the automatic data typing mechanism and instructs the compiler to issue a diagnostic for each variable used that does not appear in a type statement. Specifying the u- command-line option is equivalent to beginning each procedure with NONE.

  If you specify IMPLICIT NONE, no other IMPLICIT statement can be included in the program unit.

EQUIVALENCE Statements — Sharing Storage Between Elements

- Single subscripts are permitted for multiple-dimension arrays. The single subscript represents the linear element number.

NAMELIST Statements — Specifying Names

- A NAMELIST statement can appear anywhere in a program unit after the PROGRAM, FUNCTION, or SUBROUTINE statement. However, a namelist name must be declared in a NAMELIST statement before being referenced in a namelist-directed READ, WRITE, or PRINT statement.

Chapter 8. Control Statements

"Control statements" control the execution of a program. The control statements described in this chapter are:

- Block IF-THEN-ELSE
  - Block IF
  - ELSEIF
  - ELSE
  - ENDIF
- Logical IF
- Arithmetic IF
- DO
- CONTINUE
- STOP
- PAUSE
- Unconditional GOTO
- Assigned GOTO
- Computed GOTO
- END

The CALL, ENTRY, and RETURN control statements are described in Chapter 9, "Program and Subprogram Structure."

Block IF-THEN-ELSE Statement Group

The "block IF-THEN-ELSE statement group" is a structured-coding construction that controls program execution without using jumps via GOTO statements.

To understand the block IF statement and its associated statements more fully, you need to understand the concept of the IF-level. The IF-level is used to define the nesting rules for the block IF statement and its associated

statements and to define the extent of IF blocks, ELSEIF blocks, and ELSE blocks. The IF-level of any statement is:

n1 - n2

The *n1* is the number of block IF statements from the beginning of the current program unit, including this statement. The *n2* is the number of ENDIF statements from the beginning of the current program unit, not including this statement.

The IF-level of every statement must be greater than or equal to 0, and the IF-level of every IF, ELSEIF, ELSE, and ENDIF block must be greater than 0. The IF-level of every END statement must be 0.

The following examples illustrate the basic concepts of the IF-THEN-ELSE statement group.

Examples:

```
C     This is a block IF statement that allows
C     a group of statements to be executed
C     only if the expression is true.
C
      IF (I .LT. 10) THEN
C          The next statements are executed
C          only if I < 10.
           PATH=50
           TIME=4.
      ENDIF



C     This is a block IF statement with a
C     series of ELSEIF statements.
C
      IF (J .GT. 1000) THEN
C          The next statement is executed
C          only if J > 1000.
           PATH=50
      ELSEIF (J .GT. 100) THEN
C          The next statement is executed
C          only if 100 < J ≤ 1000.
```

```
            PATH=5
      ELSEIF (J .GT. 10) THEN
C           The next statement is executed
C           only if 10 < J ≤ 100.
            PATH=1
      ELSE
C           The next statement is executed
C           only if J < 10.
            PATH=0
      ENDIF



C      This example shows that the constructs
C      can be nested and that an ELSEIF state-
C      ment can follow a block IF statement
C      without requiring an ELSE statement.
C
      IF (I .LT. 100) THEN
C         The next statement is executed
C         only if I < 100.
          PATH=6
          IF (J .LT. 10) THEN
C             The next statement is executed
C             only if I < 100 and J < 10.
              PATH=PATH+1
          ENDIF
C         The next statement is executed
C         only if I < 100.
          INPUT=0
      ELSEIF (I .LT. 1000) THEN
C         The next statement is executed
C         only if 100 ≤ I < 1000.
          IF (J .LT. 10) THEN
C             The next statement is executed
C             only if 100 ≤ I < 1000 and
C             J < 10.
              INPUT=1
          ENDIF
C         The next statement is executed
C         only if 100 ≤ I < 1000.
          PATH=5
      ENDIF
```

Block IF Statements

The form of the block IF statement is:

```
IF ( logexpr ) THEN
```

logexpr
is a logical expression.

In executing the block IF statement, the logical expression is evaluated first. If the value of the logical expression is true and if the IF block has at least one executable statement, the next statement executed is the first executable statement of the IF block.

The IF block can have any number of statements and consists of all statements after the IF statement up to but not including the next ENDIF statement that has the same IF-level as this IF statement. The next statement to be executed after an IF block is the next ENDIF statement with the same IF-level.

If the value of the logical expression is true and there are no executable statements in the IF block, the next statement to be executed is the next ENDIF statement at the same IF-level as this IF statement.

If the value of the logical expression is false, the next statement to be executed is the next ELSEIF, ELSE, or ENDIF statement at the same IF-level as this IF statement.

Note that transfer of control into an IF block from outside the IF block is not allowed.

The block IF statement has the same appearance as the logical IF statement, which is described in "Logical IF Statements" on page 8-8; however, the block IF statement can be identified by its subsequent THEN keyword.

ELSEIF Statements

The form of the ELSEIF statement is:

```
ELSEIF ( logexpr ) THEN
```

logexpr
is a logical expression.

In executing the ELSEIF statement, the logical expression is evaluated first. If the value of the logical expression is true and if the ELSEIF block has at least one executable statement, the the next statement executed is the first executable statement of the ELSEIF block. The ELSEIF block can have any number of statements and consists of all statements up to but not including the next ELSEIF, ELSE, or ENDIF statement that has the same IF-level as this ELSEIF statement. The next statement to be executed after an ELSEIF block is the next ENDIF statement with the same IF-level.

If the value of the logical expression is true and there are no executable statements in the ELSEIF block, the next statement to be executed is the next ENDIF statement that has the same IF-level as this ELSEIF statement.

If the value of the logical expression is false, the next statement to be executed is the next ELSEIF, ELSE, or ENDIF statement that has the same IF-level as this ELSEIF statement.

Note that transfer of control into an ELSEIF block from outside the ELSEIF block is not allowed.

ELSE Statements

The form of an ELSE statement is:

```
ELSE
```

The ELSE block associated with an ELSE statement can have any number of statements (possibly none) and consists of all statements up to but not including the next ENDIF statement that has the same IF-level as this ELSE statement.

The ENDIF statement with the same IF-level as the ELSE statement must appear before any other ELSE or ELSEIF statements at this IF-level. In other words, only one ELSE statement can be in a block IF statement. There is no effect in executing an ELSE statement.

Note that transfer of control into an ELSE block from outside the ELSE block is not allowed.

ENDIF Statements

An "ENDIF statement" marks the end of a block IF statement group. The form of the ENDIF statement is:

```
ENDIF
```

To specify which statements are in a particular block IF statement, an ENDIF statement must be at the same IF-level for every block IF statement in a program unit. There is no effect in executing an ENDIF statement.

Sample Block IF-THEN-ELSE Program

This program performs some simple banking operations. First, the program determines what type of account the customer has (either savings or checking), which is the outermost IF block. If the account type is invalid, an "Error in type" message results.

Two IF blocks are nested within the outer IF block, one for each type of account. These blocks determine how much money the customer has in the account and sets arguments accordingly.

```
PROGRAM IFTHEN
REAL AMOUNT,INT
CHARACTER*10 GIFT,TYPE,STATUS
.
.
IF (TYPE.EQ.'SAVINGS   ') THEN
    IF (AMOUNT.GT.25000.00) THEN
        INT = 8.75
        STATUS = 'Money Mrkt'
        GIFT = 'Watch      '
    ELSEIF (AMOUNT.GT.500.00) THEN
        INT = 5.25
        STATUS = 'Regular    '
        GIFT = 'Blender    '
    ELSEIF (AMOUNT.GT.0.0) THEN
        INT = 5.25
        STATUS = 'Regular    '
        GIFT = 'Nothing    '
    ELSE
        INT = 0.0
        STATUS = 'Overdrawn '
        GIFT = 'Nothing    '
    ENDIF
ELSEIF (TYPE.EQ.'CHECKING  ') THEN
    IF (AMOUNT.GT.500.00) THEN
        INT = 5.25
        STATUS = 'NOWaccount'
        GIFT = 'Blender    '
    ELSEIF (AMOUNT.GT.0.0) THEN
        INT = 0.0
        STATUS = 'Regular    '
        GIFT = 'Nothing    '
    ELSE
        INT = 0.0
        STATUS = 'Bounced    '
        GIFT = 'Nothing    '
    ENDIF
ELSE
    PRINT *,'Error in type'
ENDIF
.
.
END
```

Logical IF Statements

A "logical IF statement" evaluates an expression to be either true or false and, based on the evaluation, executes or does not execute a following statement. If the value of the expression is true, the statement is executed; otherwise, the statement is not executed, and the execution sequence proceeds as if a CONTINUE statement (described in "CONTINUE Statements" on page 8-14) had been encountered. The form of the logical IF statement is:

IF (*logexpr*) *statement*

logexpr
is a logical expression.

statement
is any executable statement except a DO, block IF, ELSEIF, ELSE, ENDIF, END, or logical IF statement.

Note that functions evaluated in the logical expression can affect the value of variables in the executable statement.

Examples:

```
IF (ERR.NE.0) CALL ERROR(ERR)
```

This logical IF statement determines if ERR is equal to 0. If it is not equal to 0, the subroutine ERROR is called. If it is equal to 0, the program continues with the next statement.

```
REAL VAL
.
.
IF (VAL.EQ.1.0) GOTO 100
```

This logical IF statement determines if the real number VAL is equal to the whole number 1. However, the construction of this code is questionable because a real number in FORTRAN is rarely equal to a whole number since a real number's internal precision is limited. A large calculation involving real numbers that theoretically should come out to be a whole number can be internally produced by the computer to be a number slightly less than or slightly greater than expected. It is more effective to determine if VAL is equal to 1 in this manner:

```
REAL VAL,ERROR
ERROR = .00001
.
.
IF (ABS(VAL-1.0).LT.ERROR) GOTO 100
```

This program performs the desired test and also takes into account any loss of precision by determining if the absolute value of VAL - 1.0 is less than ERROR.

Arithmetic IF Statements

An "arithmetic IF statement" evaluates an expression and causes a transfer of control to one of three statement labels depending on the value (-, 0, or +) of the expression. The form of an arithmetic IF statement is:

IF (*expr*) *statlab1*, *statlab2*, *statlab3*

expr
is an expression that has an integer, real, or double-precision data type.

statlab1
statlab2
statlab3
are statement labels of executable statements that appear in the same program unit as the arithmetic IF statement. The same statement label can appear more than once among the three labels.

If the value of an expression is less than 0, statement label *statlab1* is executed. If the value of an expression is 0, statement label *statlab2* is executed. If the value of an expression is greater than 0, statement label *statlab3* is executed.

None of the statement labels may appear within the range of a DO loop or inside an IF, ELSEIF, or ELSE block unless the arithmetic IF statement is within the range or block also.

Example:

```
C     In this code, control is transferred
C     either to line 10, 20, or 30,
C     depending on whether K is less than,
C     equal to, or greater than 100.
C
      IF (K-100) 10,20,30
10    PRINT *,'K is less than 100.'
      GOTO 40
20    PRINT *,'K equals 100.'
      GOTO 40
30    PRINT *,'K is greater than 100.'
```

DO Statements — Loop Control ✧

A "DO statement" groups a number of statements in a procedure and is FORTRAN's principle means of loop control. The form of a DO statement is:

DO *statlab* [,] *dovar* = *first* , *last* [, *inc*]

statlab
is the statement label of the terminal statement of the DO loop within the same program unit.

dovar
is the name of an integer, real, or double-precision variable that is known as the DO variable.

first
last
inc
are integer, real, or double-precision expressions; *first* is the starting value of *dovar*, *last* is its ending value, and the optional *inc* is the value by which *dovar* is incremented for each loop. The *inc* cannot have a value of 0, can be negative, and has a default value of 1.

The executable statement with the *statlab* label is the terminal statement of the DO loop. The terminal statement cannot be an unconditional GOTO, assigned GOTO, arithmetic IF, block IF, ELSEIF, ELSE, ENDIF, RETURN, STOP, END, or DO statement.

The range of a DO statement is its associated DO loop, which begins with the statement following the DO statement and ends with the terminal statement. If a DO statement appears in the range of another DO statement, the former DO statement's associated loop must be completely inside the latter DO statement's associated loop. If a DO statement appears inside an IF, ELSEIF, or ELSE block, its DO loop must be entirely contained in that block. If a block IF statement appears inside of a DO loop, its associated ENDIF statement must also appear inside the DO loop. DO loops can share a terminal statement.

The value of the DO variable must not be set or changed by any statement within the the DO loop associated with that control variable. Thus it is a programming error to have nested DO loops with the same index variable.

These events take place when a DO statement is executed:

- the *first*, *last*, and *inc* expressions are evaluated and are converted to the data type of the DO variable if necessary, according to the data type conversion rules for arithmetic assignment statements, which are described in "Arithmetic Assignment Statements ✦ ✧" on page 6-1.

- the DO variable *dovar* is set to the value of *first*.

- the DO loop's iteration count (the number of passes through the loop) is computed using this formula:

  MAX (INT ((*last* - *first* + *inc*) /*inc*) ,0)

  in which MAX (maximum) and INT (integer) are FORTRAN intrinsic functions. The iteration count may evaluate to 0.

- the iteration count is tested; if it is greater than 0, the statements in the DO loop are executed. If the iteration count equals 0, the DO loop is bypassed and control is passed to the next executable statement after the DO loop.

  You can specify that a DO loop is to be executed at least once regardless of the iteration count, as it is in FORTRAN 66. To do so, use the y+ command-line option. For a description of command-line options, see the *RT PC VS FORTRAN User's Guide*.

These events take place when the terminal statement of a DO loop is executed:

- the value of the DO variable *dovar* is incremented by the value of *inc*.

- the iteration count is decremented by 1.

A DO loop is exited under any of these conditions:

- the iteration count is decremented to 0 or a negative number.

- a RETURN statement is executed within the DO loop.

- control is transferred to a statement in the same program unit but outside of the DO loop.

- a subroutine subprogram called from within the DO loop returns via an alternate return specifier to a statement which is outside of the DO loop.
- the program ends.

The value of the DO variable is defined both when the DO loop is exited due to an iteration count of 0 and when the DO loop is exited due to an explicit transfer of control out of the DO loop.

Examples:

```
C     This program fragment prints
C     the numbers 1-11 to the screen.
C
      DO 200 I = 1, 10
200   WRITE(*, '(I5)') I
      WRITE(*, '(I5)') I

C     This is one way of setting
C     an integer array to zeros.
C
      INTEGER ARR(20),I
      .
      .
      DO 10 I=1,20
          ARR(I) = 0
  10  CONTINUE

C     This is one way to compute N factorial.
C     The result is stored in the variable FACT.
C
      INTEGER I,N,FACT
      .
      .
      READ(*,10) N
      FACT = 1.0
      DO 20 I=2,N
         FACT = FACT * N
  20  CONTINUE
```

```
C     This loop is not executed.
C
      DO 10 I=5,1
         WRITE(*,100)
100      FORMAT('Nothing')
 10   CONTINUE
```

CONTINUE Statements

A "CONTINUE statement" is a "null" or "no operation" statement often used to end a DO block. It is primarily used to end DO blocks that would otherwise have an invalid terminal statement. The form of a CONTINUE statement is:

CONTINUE

A CONTINUE statement can appear anywhere any other executable statement can appear in a program and has no effect on program execution.

STOP Statements

A "STOP statement" stops the execution of a program. The form of a STOP statement is:

STOP [*nnn* **]**

nnn
is either a character constant or a string of 1-5 digits.

The *nnn* is displayed on the screen when the STOP statement is executed.

Example:

```
IF (IERR.NE.0) STOP 'Abnormal Termination'
STOP 'Normal Termination'
END
```

This is one possible use of the STOP statement. If the variable IERR is not equal to 0, the program stops and "Abnormal Termination" is displayed. Otherwise, "Normal Termination" is displayed. Note that only one of the two STOP statements in the example is executed.

PAUSE Statements

A "PAUSE statement" suspends a program until the Enter key is pressed. The form of a PAUSE statement is:

PAUSE [*nnn*]

nnn
is either a character constant or a string of 1-5 digits.

The *nnn* is often used as a prompt to request input.

When the Enter key is pressed, program execution resumes as if a CONTINUE statement has been executed. If the Esc (Escape) key is pressed before the Enter key, program execution is stopped.

Examples:

```
PAUSE 'Insert a diskette into the default drive.'
```

This PAUSE statement prompts the user to do a specific task. Execution is stopped until the user presses the Enter key.

```
PAUSE 10
CALL HOT(I,J,K)
CALL COLD(K,L,M)
PAUSE 11
CALL DAY(M,N,O)
CALL NIGHT(I,O)
PAUSE 12
CALL HOME (J,L)
```

Here the PAUSE statement is used for debugging a program. By placing PAUSE statements that print out numbers in strategic places, you are able to determine where the problem area in a program is. The numbers from the PAUSE statements are printed out and the execution of the program can be followed.

Unconditional GOTO Statements

An "unconditional GOTO statement" causes an unconditional transfer of control to another part of the program unit. The form of the unconditional GOTO statement is:

GOTO (*statlab*)

statlab
is the statement label of an executable statement within the same program unit. That executable statement is the target of the GOTO statement.

A GOTO statement cannot cause a transfer of control into a DO, IF, ELSEIF, or ELSE block from outside the block.

Examples:

```
      CALL CHKERR(ERR)
      IF (ERR.NE.0) GOTO 99
      .
      .
  99  STOP 'Error Condition'
      END
```

Here a GOTO is performed if an error condition exists. This is one way to terminate the program due to an error.

```
C     An error occurs here.
 300  GOTO 300
```

This is the simplest form of an infinite loop and, even though it is obvious, the compiler does not flag it as an error. It is up to you to avoid programming infinite loops of any kind.

```
      GOTO 10
      PRINT *,'This statement is never executed.'
  10  PRINT *,'This statement is executed.'
```

The PRINT statement immediately after the GOTO statement is never executed. Not only does execution jump over it because of the GOTO statement, but execution cannot be transferred to it because the PRINT statement does not have a statement label. The compiler does not flag this type of error.

```
      GOTO 100
      .
      .
C     An error occurs here.
 100  FORMAT (2F6.2)
```

A compiler error occurs because a GOTO statement is not allowed to transfer control to a a nonexecutable statement.

Assigned GOTO Statements

An "assigned GOTO statement" uses the value of an integer variable as a statement label and transfers control of the execution to the statement with that statement label. The form of an assigned GOTO statement is:

GOTO *i* **[[,]** **(** *statlab* **[,** *statlab* **]** **...** **)]**

i
: is the name of a variable that has a data type of INTEGER or INTEGER*4. (The *i* cannot have an INTEGER*2 data type.)

statlab
: is a statement label of an executable statement within the same program unit. The same statement label can appear more than once in the list of statement labels.

At the time the assigned GOTO is executed, the integer variable *i* must be defined with the value of a statement label of an executable statement that is in the same program unit as the assigned GOTO statement. This definition must be made with an ASSIGN statement. For a description of this type of assignment, see "Statement Label (ASSIGN) Assignment Statements" on page 6-4.

If the optional list of statement labels is present, the value of the integer variable *i* must be the same as one of the statement labels in the list.

The assigned GOTO statement has the same transfer of control restrictions as the unconditional GOTO statement.

Example:

```
      INTEGER SYSTEM
      ASSIGN 801 TO SYSTEM
      GOTO SYSTEM (360,370,801)
C     Code for the IBM System 360 Series computers.
  360 LENGTH=24
      GOTO 9999
C     Code for the IBM System 370 Series computers.
  370 LENGTH=48
      GOTO 9999
C     Code for the IBM RT PC Series computers.
  801 LENGTH=32
      GOTO 9999
C     Code common to all systems.
 9999 CONTINUE
```

In this example, the code for the IBM RT PC Series computers is executed because SYSTEM refers to the same symbolic address as statement label 801. Remember that SYSTEM does not hold the value 801; it holds an address associated with an executable statement. To change which executable statement the assigned GOTO statement transfers control to, the program unit needs to be changed and recompiled.

Computed GOTO Statements ✧

A "computed GOTO statement" transfers control of execution to one out of a set of labeled statements depending on the value of an expression. The form of a computed GOTO statement is:

GOTO (*statlab* [, *statlab*] ...) [,] *i*

i
is an integer expression.

statlab

is a statement label of an executable statement within the same program unit. The same statement label can appear more than once in the list of statement labels.

The computed GOTO statement evaluates the integer expression *i* to a value *n*. Control is then transferred to the statement whose statement label is in the *n*'th position in the *statlab* list. For example, if the value of an expression is 4, control is transferred to the statement whose statement label is in the fourth position in the *statlab* list.

If the value of *n* is less than 1 or greater than the number of statement labels in the *statlab* list, the computed GOTO statement has no effect, and program execution proceeds as if a CONTINUE statement had been executed.

The computed GOTO statement has the same transfer of control restrictions as the unconditional GOTO statement.

Examples:

```
C     The computed GOTO statement used in this context simulates
C     the CASE statement found in many other high-level languages.
C
      INTEGER NEXT
      .
      .
      GOTO (100,200,300,400,500) NEXT
10    PRINT *,'Execution transfers here if NEXT <> 1,2,3,4,5'
      GOTO 999

100   PRINT *,'Execution transfers here if NEXT = 1'
      GOTO 999

200   PRINT *,'Execution transfers here if NEXT = 2'
      GOTO 999

300   PRINT *,'Execution transfers here if NEXT = 3'
      GOTO 999

400   PRINT *,'Execution transfers here if NEXT = 4'
      GOTO 999
```

```
500   PRINT *,'Execution transfers here if NEXT = 5'
999   END
```

Control transfers to either line 10, 100, 200, 300, 400, or 500 depending on the value of NEXT. If NEXT does not equal 1, 2, 3, 4, or 5, control transfers to line 10. If NEXT = 1, control transfers to line 100; if NEXT = 2, control transfers to line 200, and so on.

```
      INTEGER WHERE
  10  WRITE(*,15)
  15  FORMAT('Enter 1, 2, or 3:',$)
      READ(*,20) WHERE
  20  FORMAT(I5)
      GOTO (100,200,300) WHERE
      GOTO 10

C     This code is executed when WHERE = 1.
 100    .
        .
      GOTO 400

C     This code is executed when WHERE = 2.
 200    .
        .
      GOTO 400

C     This code is executed when WHERE = 3.
 300    .
 400    .
        END
```

Control is transferred to either line 100, 200, or 300 depending on whether the user enters a 1, 2, or 3, respectively. If the user does not enter a 1, 2, or 3, another value is prompted for.

END Statements

An "END statement" indicates the end of a program unit's sequence of statements. The last executable statement in every program unit must be an END statement. The form of an END statement is:

```
END
```

An END statement executed in a function or subroutine subprogram has the same effect as a RETURN statement in a subprogram, described in "RETURN Statements ✦ ✧" on page 9-21. An END statement in a main program terminates the execution of the program.

VX Mode Specifics

This section describes the instances in which VX mode differs from IBM mode (the default mode).

DO Statements — Loop Control

- A DO loop can have an "extended range"; this occurs when a control statement contained within the DO loop transfers control out of the loop, and, after execution of one or more statements, another control statement returns control back into the loop. Thus the range of the loop is extended to include all executable statements between the destination statement of the first transfer and the statement that returns control to the loop.

  Rules governing the use of a DO statement's extended range are:

  – A transfer into the range of a DO statement is permitted only if the transfer is made from the extended range of that DO statement.

  – The extended range of a DO statement must not change the control variable of the DO statement.

  Example:

```
      DO 10 I=1,10
        PRINT *,'In main DO'
        GOTO 20
30      PRINT *,'Back in main DO'
10    CONTINUE
   .
   .
20    PRINT *,'In extended range of DO'
      GOTO 30
   .
   .
      END
```

- The "DO WHILE statement" is also allowed; it is similar to a DO statement, but instead of executing for a fixed number of iterations, it executes for as long as a logical expression contained in the statement continues to be true.

  The form of a DO WHILE statement is:

DO [*statlab* **[,]] WHILE (** *logexpr* **)**

statlab
is the statement label of an executable statement that must follow in the same program unit.

logexpr
is a logical expression.

The DO WHILE statement tests the logical expression at the beginning of each execution of the loop, including the first. If the value of the expression is true, the statements in the body of the loop are executed; if the expression is false, control transfers to the statement following the loop.

If no label appears in a DO WHILE statement, the DO WHILE loop must be terminated with an END DO statement.

Example:

```
CHARACTER*132 LINE
I=1
LINE(132:) = 'x'
DO WHILE (LINE(I:I) .EQ. ' ')
   I=I+1
END DO
```

- The "END DO statement" is also allowed, and is used terminate the range of a DO or DO WHILE statement. An END DO statement must be used to terminate a DO block if the DO or DO WHILE statement

defining the block does not contain a terminal-statement label. An END DO statement may also be used as a labeled terminal statement if the DO or DO WHILE statement does contain a terminal-statement label.

The form of an END DO statement is:

END DO

Examples:

```
DO WHILE (I .GT. J)
   ARRAY(I,J) = 1.0
   I=I-1
END DO

DO 10 WHILE (I .GT. J)
   ARRAY(I,J) = 1.0
   I=I-1
10 END DO
```

- The statement label *statlab* in the DO statement is optional. If no label appears, the DO loop must be terminated by the END DO statement.

Example:

```
C    This prints a message five times.
C
     DO J=1,5
        WRITE(*,10)
        FORMAT('This is a VX mode feature')
     END DO
```

Computed GOTO Statements

- The *i* is any valid arithmetic expression. Non-integer expressions are converted to integer values before use.

Chapter 9. Program and Subprogram Structure

A complete FORTRAN program consists of:

- a main program
- any number of subroutine subprograms
- any number of function subprograms
- any number of block data subprograms.

A "subprogram" is a program unit that begins with a SUBROUTINE, FUNCTION, or BLOCK DATA statement. A subprogram is defined separately and can be compiled independently of the main program.

A "procedure" is a subroutine or function subprogram, an intrinsic function, or a statement function. Subroutine and function subprograms, which are also known as external procedures, can share values and results through argument lists, common blocks, or both.

Main Programs and PROGRAM Statements

A "main program" is a program unit that contains at least one executable statement and does not start with a SUBROUTINE, FUNCTION, or BLOCK DATA statement. Execution of a FORTRAN program starts with the first executable statement in the main program, and therefore only one main program is allowed per FORTRAN program.

A main program can have a PROGRAM statement. If used, the PROGRAM statement can only appear as the first statement of the main program. The form of a PROGRAM statement is:

PROGRAM *progname*

progname
is the user-defined name of the main program.

The *progname* has a global scope and therefore cannot be the same name as any common block or other program unit. The *progname* is also local to the main program and cannot be the same as any other name within the main program.

Examples:

```
      PROGRAM INQUIRE

      PROGRAM CHANGE

      PROGRAM ANALYZE

These are all valid PROGRAM statements.
```

Dummy and Actual Arguments – Passing Values ✦ ✧

Values can be passed between program units in two ways. One method, described in "COMMON Statements — Declaring Common Blocks ✦ ✧" on page 7-7, uses common blocks. Another method, described here, uses argument-passing between program units.

A "dummy argument" is a name by which a value is known during the execution of a subprogram. The name is local to the subprogram and receives its value from an actual argument in the calling program. A dummy argument can be used as an actual argument by making it the calling argument in a subsequent subprogram call.

An "actual argument" contains a value that is associated with a dummy argument. This occurs when a CALL statement containing an actual argument references the subprogram, which contains the dummy argument. CALL statements are described in "CALL Statements ✦" on page 9-9.

The number and the data type of actual arguments must be the same as the number and the data type of their corresponding dummy arguments.

Example:

```
C     X, Y, and Z are the dummy arguments and
C     A, B, and C are the actual arguments.
      .
      .
      CALL SUB(A,B,C)
      .
      .
      END

      SUBROUTINE SUB(X,Y,Z)
      .
      .
      END
```

Actual arguments are associated with dummy arguments when execution is transferred to a subroutine or function subprogram. The arguments remain associated until execution of the subprogram ends. Therefore, if another value is assigned to a dummy argument while a subprogram is being executed, the value of its corresponding actual argument can change. If an actual argument is a constant, function reference, or an expression other than a simple variable, assigning a different value to its corresponding dummy argument may lead to hard-to-diagnose errors and should be avoided.

A dummy argument that is a variable can be associated with an actual argument that is a variable, an array element, or an expression.

The lengths of actual and dummy arguments that have an integer or logical data type must be the same. For example, an actual argument with a data type of INTEGER*2 can only be associated with a dummy argument with a data type of INTEGER*2.

Actual arguments that are integer expressions can only be associated with dummy arguments that have a data type of INTEGER or INTEGER*4. Similarly, actual arguments that are logical expressions can only be associated with dummy arguments that have a data type of LOGICAL or LOGICAL*4. An "expression" in this context is any construct that is not a variable, array, or array element.

Example:

```
C     The results of this code are inaccurate.
      .
      .
      CALL SUBR(523)
      .
      END

      SUBROUTINE SUBR(I)
      INTEGER*2 I
      .
      .
      RETURN
      END
```

In this program, the integer expression 523, which is the actual argument, is evaluated and stored in an integer temporary location of 4 bytes. However, the subroutine expects an argument with an INTEGER*2 data type. This mismatch of argument data types can result in inaccurate results. Similar errors occur due to other mismatches; for example, passing real constants to a subroutine that expects arguments with a double-precision data type.

If an actual argument is an expression, it is evaluated just prior to the association of actual and dummy arguments. If an actual argument is an array element, its subscript expression is also evaluated just prior to the association of actual and dummy arguments. The subscript expression remains constant, even if it contains variables that are re-defined during the execution of the subprogram.

A dummy argument that is an array can be associated with an actual argument that is either an array or an array element. The number and size of dimensions in the dummy argument may differ from those of the actual argument, but any reference to the dummy argument must be within the

limits of the array storage sequence of the actual argument. The compiler does not flag such out-of-bounds references as errors, but the results are generally unpredictable and undesirable. When passing multi-dimensional arrays between program units, pay particular attention to the array storage sequence, which is described in "Array Storage Sequence" on page 4-8.

Example:

```
C     This program may not work properly because the
C     storage sequence of arrays was not considered.
C
      PROGRAM MAIN
C
C     The dimension is made large enough to handle the
C     largest problem.
      REAL MATRIX(100,100)
      .
      .
C     Data is read from the disk file opened with logical
C     unit number 22.
      READ(22,100) MSIZE,((MATRIX(I,J),J=1,MSIZE),I=1,MSIZE)
 100  FORMAT(I6,(F20.10)
      .
C     The matrix is inverted.
      CALL MATINV(MSIZE,MATRIX)
      .
      .
      END

      SUBROUTINE MATINV(N,M)
      INTEGER N
      REAL M(N,N)
      .
      .
      RETURN
      END
```

The subroutine does not work properly if the size MSIZE of the square matrix MATRIX is less than 100 when read from the disk. Mistakes of this nature are extremely difficult to locate. To avoid making them, establish a variable for the size of the matrix, initialize this size variable in a DATA statement (to 100 for this example), and pass it in each CALL statement to the subroutine.

A dummy argument that is an asterisk (*) can only appear in the argument list or an ENTRY statement of a subroutine. The actual argument associated with this dummy argument is an alternate-return specifier in the CALL statement to the subprogram.

Dummy arguments that are arrays or character strings can have adjustable dimensions, which enable you to create program units that can accept objects of varying size. A dummy argument can be an adjustable array; that is, it can have its dimensions specified by variables passed as actual arguments. A dummy argument can be an assumed-size array; that is, the upper bound of its last dimension can be specified by an asterisk (*). In this case, the value of that dimension is not passed as an actual argument, but is determined by the number of elements in the array. If you dimension an array with an asterisk, you must ensure that the calling program unit has provided an array big enough to contain all the elements that are to be accessed by the subprogram.

A character string can have its length specified by an asterisk, which declares that the string has a varying size. The length of such a string is not passed explicitly as an argument; rather, the length is determined from the length of the actual argument by the compiler.

A dummy argument with a character data type cannot have a length greater than the length of its associated actual argument. If the length of the actual argument is greater than that of the dummy argument, the actual argument is truncated on the right.

If a dummy argument with a character data type has a length specified by an asterisk, a character expression that involves concatenation of that argument cannot be used as an actual argument to any other procedure, format specification, or input/output list in an input/output statement.

Subroutine Subprograms

A "subroutine subprogram" (or "subroutine") is a program unit that is called from other program units via the CALL statement. When invoked, it performs the actions defined by its executable statements and then returns control to the calling program unit. A subroutine does not directly return a value, although values can be passed to the calling program unit via the subroutine's arguments or via common variables.

Example:

```
C     This subroutine sorts an array of 10 real
C     numbers in ascending order.  The array X
C     is passed to the subroutine and is passed
C     back to the calling program unit.
C
      SUBROUTINE SORT(X)
      REAL X(10),TEMP
      INTEGER I,J
      DO 20 I=1,9
         DO 10 J=I+1,10
            IF (X(I).GT.X(J)) THEN
               TEMP = X(I)
               X(I) = X(J)
               X(J) = TEMP
            ENDIF
   10    CONTINUE
   20 CONTINUE
      RETURN
      END
```

SUBROUTINE Statements

A subroutine begins with a SUBROUTINE statement, ends with an END statement, and can contain all statements except PROGRAM, FUNCTION, and BLOCK DATA statements. The form of a SUBROUTINE statement is:

SUBROUTINE *subname* [(*darg* [, *darg*] ...)]

subname
is the user-defined name of the subroutine.

darg
is the user-defined name of a dummy argument. A dummy argument can be the user-defined name of a variable, array, or dummy procedure. It can also be an asterisk (*), which designates an alternate-return specifier.

The *subname* has a global scope and therefore cannot be the same as the name of any other program unit. It is also local to the subprogram it names.

The *darg* list defines the number of actual arguments to that subprogram. It also defines the data type of the arguments, as do any subsequent IMPLICIT, type, or DIMENSION statements. Argument names cannot appear in COMMON, DATA, EQUIVALENCE, or INTRINSIC statements.

If a subroutine does not have any dummy arguments, an empty argument list can be indicated by a pair of parentheses () following the name.

Examples:

```
      SUBROUTINE NOARGS

      SUBROUTINE ZILCH()

      SUBROUTINE ONEARG (RILEY)

      SUBROUTINE ALTRET(LIMIT,*)
```

These are all valid SUBROUTINE statements.

CALL Statements ✦

A "CALL statement" is used to reference a subroutine from another program unit. The form of a CALL statement is:

```
CALL subname [ ( [ arg [ , arg ] ... ] ) ]
```

subname
is the name of a subroutine.

arg
is an actual argument.

The actual arguments in the CALL statement must agree in data type and number with the corresponding dummy arguments specified in the SUBROUTINE statement of the referenced subroutine. An actual argument can be:

- an expression
- an array name
- an intrinsic function name
- an external procedure name
- a dummy procedure name
- an "alternate-return specifier" that has the form **s* in which *s* is the statement label of an executable statement in the same program unit as the CALL statement.

If a subroutine has no dummy arguments, a CALL statement referencing it cannot have any actual arguments. In this case, a CALL statement can have an optional pair of parentheses () following the subroutine name.

A CALL statement in a program unit causes these events to occur:

1. All actual arguments that are expressions are evaluated.

2. All actual arguments are associated with their corresponding dummy arguments.

3. The specified subroutine is executed.

4. Control is returned to the calling program unit when a RETURN or an END statement is executed in the subroutine. The statement to which control is returned is either the statement following the CALL statement or the statement whose alternate-return specifier is designated by the RETURN statement. RETURN statements are described in "RETURN Statements ✦ ✧" on page 9-21.

A subroutine can be called from any other program unit within the program. However, recursive subroutine calls are not allowed; that is, a subroutine cannot call itself directly or be called by another subprogram that it has called.

Sample Subroutine Subprogram

In this program, the subroutine SORT sorts an array of 10 real numbers named X. X is passed to SORT through the common block COM. SORT also receives a three-character code that tells it whether to sort in ascending or descending order. The code is received with the dummy argument ORDER, which is a character variable.

```
      PROGRAM SUBS
      REAL X(10)
      COMMON /COM/ X
      READ(*,10) X
10    FORMAT(10F6.2)
      CALL SORT('ASC')
      PRINT *,X
      CALL SORT('DES')
      PRINT *,X
      STOP
      END
```

```
      SUBROUTINE SORT(ORDER)
      COMMON /COM/ X
      CHARACTER*3 ORDER
      REAL TEMP,X(10)
      INTEGER I,J
      DO 20 I=1,9
        DO 10 J=I+1,10
          IF ((X(I).GT.X(J)).AND. (ORDER.EQ.'ASC')) THEN
            TEMP = X(I)
            X(I) = X(J)
            X(J) = TEMP
          ELSEIF ((X(I).LT.X(J)).AND. (ORDER.EQ.'DES')) THEN
            TEMP = X(I)
            X(I) = X(J)
            X(J) = TEMP
          ENDIF
10      CONTINUE
20    CONTINUE
      RETURN
      END
```

Functions

A "function" is referenced in the context of an expression and returns a value that is used as an operand in that expression.

A "function reference" in an expression causes that function to be executed. The form of a function reference is:

funcname ([*arg* [, *arg*] ...])

funcname
is the name of an intrinsic function, a statement function, or a function subprogram.

arg
is an actual argument to that function.

The number of actual arguments must be the same as the number of dummy arguments. Except for generic intrinsic functions, which are described in "Intrinsic Functions" on page 9-14, the data type of an actual argument must agree with the data type of its corresponding dummy argument. An actual argument can be:

- an expression
- an array name
- an external procedure name
- an intrinsic function name
- a dummy procedure name.

The three kinds of functions are function subprograms, intrinsic functions, and statement functions.

Function Subprograms and FUNCTION Statements ✦ ✧

A "function subprogram" (or "external function") begins with a FUNCTION statement, ends with an END statement, and can contain all statements except PROGRAM, FUNCTION, and BLOCK DATA statements. The form of a FUNCTION statement is:

[*type*] [******len1*] **FUNCTION** *funcname* [******len2*] ([*darg* [, *darg*] ...])

type
defines the data type of the value that the function subprogram returns. If *type* is omitted, the data type is determined by default and by any subsequent IMPLICIT statements that determine the data types of ordinary variables. If *type* is specified, the *funcname* cannot appear in any subsequent type statements.

len1
is the length specification, and can be an unsigned nonzero integer constant, an integer constant expression enclosed in parentheses, or an asterisk (*) enclosed in parentheses. The expression can only contain integer constants; it cannot contain names of integer constants. The *len1* has a default value of 1.

funcname
is the user-defined name of the function subprogram.

len2
is an unsigned nonzero integer constant specifying the length of the data type. It must be one of the valid length specifiers for *type*.

darg
is the user-defined name of a dummy argument.

The *funcname* has a global scope and therefore cannot be the same as the name of any other program unit. It is also local to the subprogram it names.

The *darg* list defines the number of actual arguments to that subprogram. It also defines the data type of the arguments, as do any subsequent IMPLICIT, type, or DIMENSION statements. Argument names cannot appear in COMMON, DATA, EQUIVALENCE, or INTRINSIC statements.

Unlike in SUBROUTINE statements, parentheses are required in FUNCTION statements even when there are no arguments.

The *funcname* must appear as a variable in the function subprogram. Each execution of the function must result in an assignment of a value to that variable. The final value of this variable upon executing a RETURN or END statement defines the value of the function. After the variable is defined, its value can be referenced in an expression just like any other variable. In addition to this value, a function subprogram can return other values via assignments to its dummy arguments or through variables in common blocks.

A *funcname* that is declared to have a data type of CHARACTER*(*) has its length derived from the specification of the function in the calling program unit.

Example:

```
C     This program includes a function subprogram
C     that calculates a root of a polynomial using
C     the quadratic equation.
C
      REAL ROOT,X2,X1,X0
      .
      .
C
C     2*(X**2) + 4.5*X + 1
C
      X2 = 2.0
      X1 = 4.5
      X0 = 1.0
      ROOT = QUAD(X2,X1,X0)
      .
      .
      END

      REAL FUNCTION QUAD(A,B,C)
      REAL A,B,C
      QUAD = (-B + SQRT(B**2 - 4*A*C)) / (2 * A)
      RETURN
      END
```

In this example, the actual arguments X2, X1, and X0 are passed to the function subprogram QUAD where they are associated with the dummy arguments A, B, and C, respectively. The variable ROOT takes on the value that the function subprogram returns.

Intrinsic Functions

"Intrinsic functions" are system-supplied functions; that is, they are built into the system because they are otherwise difficult to express. A FORTRAN intrinsic function returns a single value and is referenced in the same way as a user-defined function.

If a variable, array, or statement function is defined with the same name as that of an intrinsic function, the name is local to that program unit in which it is declared and the intrinsic function cannot be used in that program unit.

If a function subprogram is defined with the same name as that of an intrinsic function, the name references the intrinsic function unless the name is declared in an EXTERNAL statement.

Many intrinsic functions have both "generic" and "specific" names. Generally, the specific forms of the intrinsic functions require arguments to be of a particular data type, and an error is generated if an argument of the wrong data type is used as an actual argument in a specific intrinsic function reference. The generic forms usually do not require their arguments to be of a particular data type. When the compiler identifies a generic function reference in a program, the compiler actually calls the specific function that is appropriate for the data type of the actual argument in the reference. In general, the use of generic function names in programs is more convenient. Except for the data type conversion functions, the data type of the argument to a generic function determines the data type of the result.

For example, the generic intrinsic function LOG computes the natural logarithm of its argument, which can have a real, double-precision, or complex data type. The data type of the result is the same as the data type of its argument. The specific intrinsic functions ALOG, DLOG, and CLOG also compute natural logarithms. ALOG computes the logarithm of a real argument and returns a real result. Likewise, DLOG and CLOG accept double-precision and complex arguments and return double-precision and complex results, respectively.

Only a specific intrinsic function name can be used as an actual argument when an intrinsic function name is passed to a user-defined procedure or function.

The intrinsic functions, their generic and specific names, and their argument and result data types are listed in Appendix A, "Intrinsic Functions."

Statement Functions

A "statement function definition" is a single statement in a program unit containing an operation on dummy arguments and is nonexecutable. Any statement function definitions in a program unit can only appear after any specification statements and before any executable statements.

A "statement function reference" in the same program unit contains actual arguments and refers to the statement function definition. The actual argu-

ments are combined according to the statement function definition. A statement function is executed by referencing it just like a function.

The form of a statement function definition is:

```
statfunc ( [ darg [ , darg ] ... ] ) = expression
```

statfunc
is the name of the statement function being defined.

darg
is the user-defined name of a dummy argument.

expression
is an expression that defines how the dummy arguments are to be combined.

The data type of *expression* must be assignment-compatible with the data type of *statfunc*. The *darg* list defines the data types and number of actual arguments to the statement function.

The scope of any dummy arguments is limited to the statement function. Therefore, a name of a dummy argument can also be used as the name of something else in the rest of the program unit. If the name of a dummy argument in a statement function definition is the same as a local name in the program unit, a reference to that name within the statement function always refers to the dummy argument. It never refers to the other usage.

The name of the statement function definition has a scope that is local to its program unit. Therefore, it cannot be used for anything else other than as the name of a common block or as the name of a dummy argument to another statement function definition. In all such uses, the data type of the name must be the same.

References to dummy arguments of the containing subprogram, variables, other functions, array elements, and constants are all allowed within the *expression*. However, statement function references must refer to statement functions defined prior to the statement function in which they appear.

Statement functions are not recursive, either directly or indirectly. Also, a statement function can only be referenced in the program unit in which it is defined.

A statement function name cannot appear in any specification statement other than a type statement or a COMMON statement. If a statement function name appears in a type statement, that name cannot be defined as an array name. If a statement function name appears in a COMMON statement, that name can only be the name of the common block.

Examples:

```
C     QUAD is a statement function that
C     calculates the root of a polynomial
C     using the quadratic equation.
C
      PROGRAM EXAMP1
      .
      .
      REAL QUAD

C     This ends the specification statements.
C
      QUAD(A,B,C) = (-B + SQRT(B**2 - 4*A*C)) / (2 * A)

C     This begins the executable statements.
C
      .
      .
      X = QUAD(1.0,2.5,0.5)
      .
      .
      END
```

As this example illustrates, statement functions must appear after all specification statements and before all executable statements. In this code, the actual arguments take on the real default data type.

```
      PROGRAM EXAMP2
      PARAMETER (PI = 3.141592654)
      REAL AREA,CIRCUM,R,RADIUS
      .
      .
      AREA(R) = PI * (R**2)
      CIRCUM(R) = 2 * PI * R
      .
      .
      READ(*,100) RADIUS
  100 FORMAT(F6.2)
      PRINT *,'The area is: ',AREA(RADIUS)
      PRINT *,'The circumference is: ',CIRCUM(RADIUS)
      .
      END
```

In this example, the two statement functions AREA and CIRCUM are defined. Note that both AREA and CIRCUM refer to PI, which is not an actual argument. This is permissible because the statement function can refer to all accessible variables within its program unit.

ENTRY Statements

A subroutine or function subprogram has a primary entry point that is established via the SUBROUTINE or FUNCTION statement declaring the program unit. A subroutine call or a function reference normally activates a subprogram at its primary entry point, and the first statement that is executed is normally the first executable statement in the subprogram.

An "ENTRY statement" defines an alternate entry point in a subroutine or function subprogram. This alternate entry point is the start of a sequence of statements that is different from the sequence executed by entering the subprogram at its primary entry point. Additionally, an alternate entry point can have a dummy argument list that differs in number and data type from that of the primary entry point and from those of other ENTRY statements in the same subprogram. The form of an ENTRY statement is:

ENTRY *entname* [(*darg* [, *darg*] ...)]

entname
is the user-defined name of the entry point for a subroutine or function subprogram.

darg
is the user-defined name of a dummy argument. A dummy argument can be the user-defined name of a variable, array, or dummy procedure. If the subprogram is a subroutine, the dummy argument can also be an asterisk (*), which designates an alternate-return specifier.

The *entname* has a global scope and is also local to the subprogram it names. The *entname* cannot appear as a dummy argument in a FUNCTION or SUBROUTINE statement or in another ENTRY statement in the same subprogram. It also cannot appear in an EXTERNAL statement. In a function subprogram, the only place *entname* can be used prior to the ENTRY statement is in a type statement.

The *darg* list defines the number of actual arguments to that subroutine. It also defines the data type of the arguments, as do any IMPLICIT, type, or DIMENSION statements. Argument names cannot appear in COMMON, DATA, EQUIVALENCE, or INTRINSIC statements.

An ENTRY statement cannot appear within an IF block or a DO block. If an ENTRY statement does not have any dummy arguments, an empty argument list can be indicated by a pair of parentheses () following the name.

When a subprogram is referenced or called via an alternate entry point, the actual arguments must agree in number, order, and data type with the dummy arguments except for subroutine names and alternate-return specifiers, which do not have a data type.

If a function subprogram has a character data type, all entry points to it must also have a character data type. If the length of a character function subprogram is specified with an asterisk (*), all entry points to that function must also have a length specified with an asterisk; otherwise, all entry points must have the same length specification.

Examples:

```
C     This subroutine opens from one to four data files.
C
      SUBROUTINE OPEN4
      OPEN(4,FILE='File4.Dat')
      ENTRY OPEN3
      OPEN(3,FILE='File3.Dat')
      ENTRY OPEN2
      OPEN(2,FILE='File2.Dat')
      ENTRY OPEN1
      OPEN(1,FILE='File1.Dat')
      RETURN
      END
```

To open four data files, do a `CALL OPEN4`; this causes the four OPEN statements to be executed. To open only one data file, do a `CALL OPEN1`; this causes only the last OPEN statement to be executed.

```
C     This function subprogram calculates the volume of a
C     cylinder, given the radius of the base and the height.
C     Also, the entry point AREA can be referenced to
C     calculate the area of a circle, given the radius.
C
      REAL FUNCTION VOL(RDS,HGT)
      PARAMETER (PI = 3.141592654)
      REAL RDS,HGT
      A(RDS) = PI * RDS**2
      VOL = A(RDS) * HGT
      RETURN
      ENTRY AREA(RDS)
      AREA = A(RDS)
      RETURN
      END
```

The advantage of using this code over using two separate function subprograms is that the equation for the area of a circle needs to be specified only once (with the statement function A) instead of twice. Note that the argument lists for VOL and AREA do not have to match.

RETURN Statements ✦ ✧

A "RETURN statement" ends the execution of its subroutine or function subprogram and returns control to the calling program unit. The execution of a RETURN statement is equivalent to the execution of an END statement. The form of a RETURN statement in a function subprogram is:

```
RETURN
```

The form of a RETURN statement in a subroutine is:

```
RETURN [ int ]
```

int
is an integer constant or an integer expression.

When a RETURN statement is in a function subprogram, the value of the function is the current value of the variable with the same name as the function. If the function variable is not assigned a value prior to the execution of a RETURN or END statement, the function value is undefined.

The *int* appearing in a RETURN statement of a subroutine indicates an alternate return to the calling program unit. If *int* lies between 1 and *n*, where *n* is the number of asterisks (*) in the SUBROUTINE or ENTRY statement, the *int*'th asterisk in the dummy argument list is selected. Control then returns to the calling program unit at the statement whose statement label is specified in the *int*'th alternate-return specifier in the CALL statement.

For example, if the value of *int* is 5, the fifth asterisk in the dummy argument list is selected, and control is returned to the statement whose statement label is specified in the fifth alternate return specifier in the CALL statement (see "CALL Statements ✦" on page 9-9).

If *int* is omitted or if it lies outside the range 1 to *n*, a normal return is executed. Control is returned to the calling program unit at the statement following the CALL statement.

Examples:

```
      REAL FUNCTION ABS(X)
      REAL X
      ABS=X
      IF(X.GT.0) RETURN
      ABS=-X
      END
```

This example illustrates that a RETURN statement does not have to be the last statement in a subprogram. It also shows that an END statement in a subprogram is equivalent to a RETURN statement.

```
      PROGRAM EXAMPL
      REAL ARG1,ARG2
      .
      .
      CALL DOIT(*10,ARG1,ARG2,*20,*30)
      PRINT *,'If WHERE <> 1, 2, or 3, return here!'
      .
10    PRINT *,'If WHERE = 1, return here!'
      .
20    PRINT *,'If WHERE = 2, return here!'
      .
30    PRINT *,'If WHERE = 3, return here!'
      .
      .
      END

      SUBROUTINE DOIT(*,X,Y,*,*)
      REAL X,Y
      INTEGER WHERE
      .
      .
      RETURN WHERE
      END
```

If WHERE = 1, execution is transferred to the statement labeled 10 in the main program after returning from DOIT. If WHERE = 2 or WHERE = 3, execution transfers to the statements labeled 20 or 30, respectively. If WHERE does not equal 1, 2, or 3, execution transfers to the statement immediately following the CALL statement, as would happen even if WHERE was not on the RETURN statement.

In a Main Program: In a main program, a RETURN statement can be used to stop the execution of a program. It acts the same as a STOP statement, described in "STOP Statements" on page 8-14.

Example:

```
PROGRAM MAIN
PRINT*,'This is an IBM mode feature!'
RETURN
PRINT*,'This sentence is not printed.'
END
```

In this example, the second PRINT statement is never executed because the RETURN statement stops the program's execution.

Definition Status

When a RETURN or END statement is executed in a subprogram, all data objects within the subprogram become undefined except:

- data objects specified in SAVE statements
- data objects in blank common blocks
- data objects in a named common block that appear in the current subprogram and also appear in at least one other subprogram that directly or indirectly references the current subprogram
- initially defined data objects that have neither been redefined nor become undefined.

If a named common block appears in the main program, data objects in that common block remain defined.

Examples:

```
c     This program illustrates the different
C     definition statuses of a counter variable
C     in three different subroutines.
c
      PROGRAM EXAMPL
      DO 10 I=1,5
         CALL SUBA(I)
         CALL SUBB(I)
         CALL SUBC(I)
  10  CONTINUE
      END

      SUBROUTINE SUBA(KOUNT)
      IF (KOUNT.EQ.1) I=0
      I=I+1
      WRITE(*,10) I
  10  FORMAT('SUBA has been called 'I2' times. ***ERROR***')
      RETURN
      END

      SUBROUTINE SUBB(KOUNT)
      SAVE J
      IF(KOUNT.EQ.1) J=0
      J=J+1
      WRITE(*,10) J
  10  FORMAT('SUBB has been called 'I2' times.  ***WORKS!!!***')
      RETURN
      END

      SUBROUTINE SUBC(KOUNT)
      DATA K/0/
      K=K+1
      WRITE(*,10) K
  10  FORMAT('SUBC has been called 'I2' times.  ***WORKS!!!***')
      RETURN
      END
```

The value 1 is printed the first time subroutine SUBA is called. In subsequent calls, however, undefined values are printed because the variable I in

SUBA loses its value when SUBA returns control to the main program. The variables J and K in the other two subprograms are effective counter variables since their values are retained even after a return to the main program. The values are retained by using SAVE and DATA statements.

Block Data Subprograms and BLOCK DATA Statements

A "block data subprogram" is nonexecutable and is used to initialize data declared in common blocks. More than one block data subprogram is allowed in a FORTRAN program, but only one can be unnamed. A block data subprogram begins with a BLOCK DATA statement, ends with an END statement, and can contain type, IMPLICIT, PARAMETER, DIMENSION, COMMON, SAVE, EQUIVALENCE, and DATA statements. The form of a BLOCK DATA statement is:

BLOCK DATA [*blockname*]

blockname
is the name of the BLOCK DATA subprogram.

If present, the *blockname* cannot be the same as the name of any common block or other program unit. It also cannot be the same as any local name in the subprogram.

More than one named common block can be initialized in the same block data subprogram. All the variables in a named common block must be specified, even if they are not all initialized. Also, any one common block can be specified in only one block data subprogram.

Example:

```
c     Notice that not all the variables are initialized.
c
      BLOCK DATA
c
      COMMON /SHAPES/ CIRCLE,TRIANGLE,SQUARE
      REAL CIRCLE
      COMPLEX TRIANGLE
      DOUBLE PRECISION SQUARE

      COMMON /FOODS/ BURGER,DOGS,FRIES
      LOGICAL BURGER
      REAL DOGS
      COMPLEX FRIES

      DATA CIRCLE,TRIANGLE / 3.2, (11.5,1.5) /
      DATA BURGER /.TRUE./,DOGS /.45/
c
      END
```

R1 Mode Specifics

This section describes the instances in which R1 mode differs from IBM mode (the default mode).

CALL Statements

- Recursive subroutine calls are allowed; that is, a subroutine can call itself directly or be called by another subprogram that it has called.

Function Subprograms and FUNCTION Statements

- The length specifications *len1* and *len2* are not allowed.

RETURN Statements

- A RETURN statement cannot be used in a main program.

VX Mode Specifics

This section describes the instances in which VX mode differs from IBM mode (the default mode).

Dummy and Actual Arguments — Passing Values

- If the actual argument is a Hollerith constant, the dummy argument must have a numeric data type.

Function Subprograms and FUNCTION Statements

- The length specification *len1* is not allowed.

RETURN Statements

- If necessary, *int* is converted to integer.
- A RETURN statement cannot be used in a main program.

Chapter 10. Input and Output

This chapter contains a description of:

- the concepts of FORTRAN input and output, including files, records, units, and the various forms of file access and record formats
- the parameters common to the FORTRAN input/output statements
- these input/output statements:
  - OPEN
  - CLOSE
  - READ
  - WRITE
  - PRINT
  - BACKSPACE
  - ENDFILE
  - REWIND
  - INQUIRE

The FORMAT statement is described in Chapter 11, "Format Specifications." Additional information on FORTRAN input and output under the RT PC AIX Operating System can be found in the *RT PC FORTRAN User's Guide.*

Concepts of FORTRAN Input and Output

A "record" is a sequence of characters or values and is the building block of the FORTRAN input/output system. The three kinds of records are:

- formatted records
- unformatted records

- endfile records.

A "formatted record" is a sequence of characters ended by an end-of-record control character, which is the line-feed character (ASCII decimal value 10). Therefore, a formatted record is a line of text. Formatted records are interpreted on input in the same manner that the operating system and any text editor interprets characters. In general, formatted files are transportable to other language processors and computers.

An "unformatted record" is a sequence of values. Unformatted records are not altered or interpreted by the input/output system and may not have an end-of-record character like formatted files. In general, unformatted files are not transportable to other language processors and computers because of different internal representations of values.

An "endfile record" should be thought of as a label immediately following the last record of a file, which signals the end of that file. It does not physically exist but the input/output system supplies an indication of one to the underlying operating system.

A "file" is a set of related records treated as a unit. A FORTRAN file is either external or internal.

External Files

An "external file" is a file on a physical peripheral device such as a disk file or is actually a physical device such as a printer or console. An external file being operated upon by a FORTRAN program has a number of properties including:

- a file name
- a file position
- a file record format
- a file access method.

File Name

The "file name" of an external file is a character string identical to that by which it is known to the operating system.

File Position

The "position" of an external file is usually established by the preceding input/output operation. There is a notion of the beginning-of-file, the end-of-file, the current record, the preceding record, and the next record. It is also possible to be between records, in which case the next record is the successor to the previous record, and there is no current record.

The position of the file after a sequential WRITE statement is the end-of-file, but not beyond the endfile record. A READ statement executed at the end-of-file, but not beyond the endfile record, positions the file beyond the endfile record. Executing an ENDFILE statement positions the file beyond the endfile record. A program can branch on reading an endfile record via the END= option in a READ statement.

File Record Format — Formatted or Unformatted

The "file record format" of an external file is formatted or unformatted. Internal files are always formatted.

A "formatted file" consists entirely of formatted records. Each record is in ASCII form and is ended by an end-of-record character.

An "unformatted file" consists of unformatted records; that is, each record is in binary form and is ended by an end-of-record character.

Unformatted files are more efficient than formatted files in input/output overhead and in file space requirements. For example, 8 bytes are needed to write a single-precision real number with six digits of accuracy using a formatted WRITE statement. Using an unformatted WRITE statement, only 4 bytes are needed. Unformatted access can be used if data is to be written and read by FORTRAN on the same processor. However, unformatted files cannot be printed or edited with standard text editors.

If data is to be transferred without any system interpretation, especially if all 256 character combinations are needed, unformatted input/output is necessary since formatted files are limited to the printable character set. For example, unformatted input/output is needed to control a device with a single-byte binary interface. If formatted input/output is used, certain characters (such as the carriage return) have their meanings changed.

File Access Method — Sequential or Direct

The file access method of an external file is either sequential or direct (random).

A "sequential-access file" (or "sequential file") contains records in an order determined by the order in which the records were written. Sequential files cannot be read or written using the REC= option, which specifies a direct access input/output position. The FORTRAN input/output system extends a sequential file if a record is written beyond the endfile record providing that enough space exists on the external device.

A "direct-access file" (or "direct file") can have its records written in any order. Each record is assigned a specific record position at the time it is written, which is used to reference the record. Record positions are numbered sequentially with the first record position having the number 1. All records in a direct file must have the same length, which is specified when the file is opened.

If desired, records can be written out of order and record positions can be skipped. For example, records can be written in positions 9, 5, and 11 in that order without writing the intermediate records. A record cannot be deleted but can be rewritten with a new value. Also, a record position that has not been written cannot be read, even though the FORTRAN input/output system recognizes this as an error only when the attempted read is to a record beyond the highest-numbered record in the file.

In order for a position in a direct file to be meaningful, direct files must reside in block-structured storage devices, also known as blocked devices. Blocked devices, such as disks, have the capability to locate a specific position on the device and can be used for both sequential access and direct access of files. FORTRAN does not allow direct access of sequential devices since there is no notion of seeking an absolute location on an unblocked file.

The FORTRAN input/output system extends a direct file if a write is made to a position beyond the current highest-numbered record in the file providing that enough space exists on the storage device.

Most uses of direct files tend to also require unformatted files. Direct formatted files require special care. FORTRAN formatted files generally comply with the operating system's rules for text files, thereby allowing standard system utilities such as text editors to be used on them. These rules are enforced for sequential formatted files but may not always be enforced for direct formatted files (which may not be valid text files) because they may have "holes" of unwritten records. Text editors may not be able to interpret these unwritten records in a file.

Direct formatted files cannot contain any character-compression information.

Note: The combination of direct and unformatted files is ideal for a data base management facility that is accessed exclusively through the FORTRAN input/output system.

Internal Files ✦ ✧

An "internal file" is a character variable, character array element, character array, or character substring that serves as the source or destination of some input/output action. A record of an internal file is a character variable, character array element, or character substring.

Internal files provide a means for using the formatting capabilities of the FORTRAN input/output system to convert values to and from their external character representations within FORTRAN's internal storage structures. This means that reading from a character variable converts the character values into numeric, logical, or character variables. Writing to a character variable converts values into their external character representation.

If an internal file is a character variable, character array element, or character substring, the file has exactly one element with a length equal to the length of the character variable, character array element, or character substring. If an internal file is a character array, each element of that array is a record of the file with each record having the same length.

If less than an entire record is written by a WRITE statement, the remainder of the record is filled with spaces.

The position of an internal file is always at the beginning-of-file prior to the execution of any input/output statements. Reading and writing records to an internal file can be accomplished by using sequential formatted input/output statements, list-directed input/output statements, and namelist-directed input/output statements. Only READ, WRITE, and PRINT statements can reference an internal file. Other statements such as OPEN, CLOSE, and INQUIRE cannot be used on internal files.

Example:

```
C     This program illustrates how internal files can be used
C     to convert integer and real numbers to character strings,
C     and vice versa.
C
      PROGRAM INFILE
      CHARACTER*10 STR
      REAL VAL

C     A character string is converted to a real number.
C
      STR = '123.456'
      READ(STR,100) VAL
  100 FORMAT(F6.2)
      WRITE(*,*) 'Real number after conversion is: ',VAL

C     A real number is converted to a character string.
C
      VAL = 456.98
      WRITE(STR,100) VAL
      WRITE(*,*) 'Character string after conversion is: ',STR

C     A character string is converted to an integer number.
C
      STR = '789'
      READ(STR,120) IVAL
  120 FORMAT(I3)
      WRITE(*,*) 'Integer number after conversion is: ',IVAL
```

```
C     An integer number is converted to a character string.
C
      IVAL = 890
      WRITE(STR,120) IVAL
      WRITE(*,*) 'Character string after conversion is: ',STR
      STOP
      END
```

Units ✧

A "unit" is a means of specifying a file. A unit is specified in an input/output statement by either an external unit specifier or an internal unit specifier.

An "external unit specifier" is either a positive integer expression or an asterisk (*), which indicates the console on the operating system. In most cases, external unit specifiers are bound by name to external devices or files on those devices when the OPEN statement is executed. Once a value is associated with an external file name, the value is used refer to the external file. The external unit specifier is uniquely associated with the external device or file until a CLOSE statement is executed or until the program ends.

Exceptions to these rules involve the external units 5 and 6, which are initially associated with the console for reading and writing and do not require an OPEN statement. An asterisk as an external unit specifier also references unit 5. The console is a sequential formatted file. When reading from the console, the backspace and line-delete keys have their normal editing functions.

An "internal unit specifier" is the name of a character variable, character array, character array element, or character substring. Internal unit specifiers can only be used in READ, WRITE, and PRINT statements and are not allowed in any of the auxiliary input/output statements such as OPEN, CLOSE, and INQUIRE.

Sample Input/Output Program

This program copies a file containing three columns of integers and switches the first and second columns. Each column is seven characters wide. The input file is named by the user and the output file is OUT.TXT.

```
C     Before executing program --
C       from Bourne shell:  OUTFILE=OUT.TXT
C                           export OUTFILE
C
C       from C shell:  setenv OUTFILE OUT.TXT
C
      PROGRAM SWITCH
      CHARACTER*23 FNAME
      INTEGER FIRST, SECOND, THIRD
      LOGICAL EXISTS

C     A prompt is made to the console
C     by writing to *.
C
 100  WRITE(*,900)
 900  FORMAT ('Input the symbolic name of file: ')

C     The file name is read from the
C     console by reading from *.
C
 150  READ(*,910) FNAME
 910  FORMAT(A)

C     A test is conducted to determine if
C     the file exists.
C     If not, the user is again prompted.
C
      INQUIRE(FILE=FNAME,EXIST=EXISTS)
      IF (.NOT.EXISTS) THEN
         WRITE(*,920)
 920     FORMAT ('The file does not exist.
     +  Enter another file name: ')
         GOTO 150
      ENDIF

C     Unit 3 is used for input.
C
      OPEN(3,FILE=FNAME)
```

```
C     Unit 4 is used for output.
C
      OPEN(4,FILE='OUTFILE',STATUS='NEW')

C     The file is read and written until
C     end-of-file is reached.
C
 200  READ(3,930,END=300) FIRST,SECOND,THIRD
      WRITE(4,930) SECOND,FIRST,THIRD
 930  FORMAT(3I7)
      GOTO 200
C
 300  WRITE(*,910) 'The copying is finished.'
      STOP
      END
```

Parameters of Input/Output Statements ✧

FORTRAN input/output statements require certain parameters that specify sources and destinations of data transfers, as well as other aspects of operations.

Unit Specifiers

A "unit specifier" indicates the unit to be used in an input/output operation. The form of a unit specifier is:

[UNIT =] *u*

u
is an external or internal unit specifier.

An external unit specifier is either an *, 5, or 6, (which refer to the console) or can be a positive integer expression (which refers to an external file with

that unit number). An internal unit specifier is the name of a character variable, character array, character array element, or character substring.

If the optional UNIT= keyword is not used, the *u* must be the first item in the argument list of specifiers. If the UNIT= keyword is used, then the FMT= keyword must be used, and all the specifiers in the argument list can appear in any order.

Format Specifiers ✦

A "format specifier" indicates the format to be used in an input/output operation. The form of the format specifier is:

```
[ FMT = ] f
```

f

can be a statement label, an integer variable, a character expression, an asterisk (*), or an array name.

A statement label refers to the FORMAT statement that has that label. An integer variable refers to the FORMAT label that it is associated with in an ASSIGN statement. A character expression refers to its current value as the format specifier. An asterisk specifies list-directed formatting.

An array name can have a data type of integer, real, double-precision, logical, or complex. It must contain character data whose leftmost characters constitute a valid format specification. The length of the format specification may exceed the length of the first element of the array; it is considered the concatenation of all the elements of the array in the order given by array element ordering.

If the optional FMT= keyword is not used, the *f* must be the second item in the argument list of specifiers. If both UNIT= and FMT= are used, the argument list of specifiers can appear in any order.

For a description of the format list and the elements it can contain, see Chapter 11, "Format Specifications."

Record Number Specifiers ✧

A "record number specifier" indicates the number of the record to be read or written in a direct access input/output operation. The form of the record number specifier is:

```
REC = rn
```

rn

is an integer expression. It represents the relative position of a record within the file associated with UNIT=*u*. The internal record number of the first record is 1.

End-of-File Exit Specifiers

An "end-of-file exit specifier" designates a statement label at which execution is to start when an end-of-file condition occurs while reading from a file. The form of an end-of-file specifier is:

```
END = s
```

s

is a statement label in the same program unit as the READ statement. (The *s* cannot be an integer variable assigned the value of a statement label through an ASSIGN statement.)

Error Exit Specifiers

An "error exit specifier" designates a statement label at which execution is to start when an error occurs during the execution of an input/output statement. The form of an error exit specifier is:

ERR = *s*

s
is a statement label in the same program unit as the input/output statement. (The *s* cannot be an integer variable assigned the value of a statement label through an ASSIGN statement.)

Input/Output Status Specifiers

An "input/output status specifier" designates an integer variable into which the status of an input/output operation is returned. The form of an input/output status specifier is:

IOSTAT = *ios*

ios
is the name of an integer variable or integer array element. The data type of *ios* must be INTEGER or INTEGER*4.

When an input/output statement containing this specifier finishes execution, the *ios* is defined. A zero value indicates that the input/output operation completed normally; that is, there were no errors and an end-of-file was not encountered. A negative value indicates that an end-of-file was encountered during a READ statement. A positive value indicates that an error condition occurred during the execution of the input/output statement. Error numbers and messages are listed in the *RT PC VS FORTRAN User's Guide*.

Note: The meaning of a positive value differs among operating systems.

Input/Output Lists ✧

An "input/output list" specifies the objects whose values are transferred by READ, WRITE, and PRINT statements. An input/output list is a list of elements separated by commas, and can be empty.

Input/output lists can contain input/output objects and implied DO loops, which are described in the following two sections.

Input/Output Objects ✧

"Input/output objects" can be specified as items in input/output lists of READ, WRITE, and PRINT statements. An input or output object can be:

- a variable name
- an array element name
- a character substring name
- an array name, in which all the elements of the array are specified in the order that they are stored internally
- (for output objects only) any other expression except a character expression that concatenates an assumed-length character string.

Example:

```
      WRITE (0,100) 'The results are: ',WIDGET, OMELET(J,4)
 100  FORMAT (A,I5,F10.5)

This is a valid input/output list.
```

Implied DO Loops ✧

"Implied DO loops" can be specified as items in input/output lists of READ, WRITE, and PRINT statements. The form of an implied DO loop is:

```
( dlist , dovar = first , last [ , inc ] )
```

Implied DO loops are described in "Implied DO Loops in DATA Statements" on page 7-12.

In a READ statement, the implied DO variable *dovar* and associated objects cannot appear as input list items in the implied DO list *dlist* but can be read in the same READ statement outside of the implied DO loop. The *dlist* assignment is repeated for each iteration of *dovar*, which is appropriately incremented for each repetition of the implied DO loop.

Example:

```
      ISIZE=10
      READ(3,150) (JINX(I),I=1,ISIZE)
 150  FORMAT(10I7)
```

In this code, notice that an object associated with the implied DO loop is read with the same READ statement but outside of the implied DO loop. Elements of the array JINX are defined for each iteration of the variable I. The format specified in the FORMAT statement causes the values to be read 10 per record.

Input/Output Statements ✧

The FORTRAN input/output statements include OPEN, CLOSE, READ, WRITE, PRINT, BACKSPACE, ENDFILE, and REWIND. During the execution of any input/output statement, expression evaluation can reference functions. Any function so referenced cannot execute any other input/output statement.

OPEN Statements ✦ ✧

An "OPEN statement" associates a unit number with an external device by specifying its file name. If the file is to be a direct file, the RECL=*rl* option specifies the length of the records in that file. If the unit specified in an OPEN statement is already opened, it is closed before being associated with a file. The form of an OPEN statement is:

OPEN (*arglist*)

arglist
is an argument list of specifiers that must contain one unit specifier. It can also contain one of each of the other allowable specifiers.

The forms of the unit specifier and other allowable specifiers are:

[UNIT=] *u*
is an external unit specifier described in "Unit Specifiers" on page 10-9.

IOSTAT = *ios*
is an input/output status specifier described in "Input/Output Status Specifiers" on page 10-12.

ERR = *s*
is an error exit specifier described in "Error Exit Specifiers" on page 10-12.

FILE = *fname*
identifies a shell environment variable whose value is the file name. If this specifier is omitted, unit *n* is connected to FILE.FT*n*F001.

Example:

```
OPEN (9,FILE='MYFILE')
```

This statement looks for a shell environment variable named MYFILE. If the variable exists, it's value is the file name; if there is no such variable, a default file named FILE.MYFILE is used.

STATUS = *sta*
specifies the status of the file when it is opened. The *sta* is a character expression whose value must be 'OLD', 'NEW', 'SCRATCH', or 'UNKNOWN'. 'OLD' is used for reading or writing existing files. 'NEW' is used for writing new files. 'UNKNOWN' is the default.

If 'NEW' is specified and the file already exists, the file is truncated to a zero length and opened. If the file does not exist, it is created. If 'OLD' is specified and the file does not exist, an error results.

If 'OLD' or 'NEW' is specified, the FILE= keyword must be supplied. If 'SCRATCH' is specified, the file is automatically deleted when the file is closed or when the program ends. If 'UNKNOWN' is specified, the file is treated as if 'OLD' had been specified.

ACCESS = *acc*
specifies the access mode for the file. The *acc* is a character expression whose value must be either 'SEQUENTIAL' or 'DIRECT'. 'SEQUENTIAL' is the default.

FORM = *fm*
specifies whether the file is formatted or unformatted. The *fm* is a character expression whose value must be either 'FORMATTED' or 'UNFORMATTED'. 'FORMATTED' is the default.

RECL = *rl*
specifies the record length of the file. The *rl* is an integer expression. This specification is limited to and required for direct files.

BLANK = *blnk*
controls the default treatment of blanks (spaces) in formatted READ statements, which can be altered by a BN (blank null) or BZ (blank zero) edit-descriptor in a format specification. The *blnk* is a character expression whose value must be either 'NULL' or 'ZERO'. The default is 'NULL', in which all spaces are ignored in numeric input fields. If 'ZERO' is specified, all spaces other than leading spaces are treated as zeros.

A file opened in FORTRAN is either old or new, and there is no distinction between "open for reading" and "open for writing." Therefore, an old (existing) file can be opened and written to, thereby changing the file. Similarly, the same file can be alternately read from and written to, providing that no attempts are made to read beyond end-of-file or to read unwritten records from direct files. Writing to a sequential file deletes any records that exist beyond the record being written to.

When a device such as console or printer is opened as a file, it generally does not make a difference whether the file is old or new. With disk files, opening 'NEW' creates a new temporary file. If that file is closed with STATUS='KEEP' or if the program ends before a CLOSE statement is performed on the file, a permanent file is created with the name given when the file was opened. Any file that already exists with that name is deleted. If a file is closed with STATUS='DELETE', the newly created temporary file is deleted and any file already existing with that name remains intact.

Opening a disk file that does not exist with STATUS='OLD' generates a run-time error. If the file does exist, writing to that file changes its contents.

Examples:

```
C     In Bourne shell:
C       DATAFILE=DATA1.DAT
C       export DATAFILE
C
C     In C shell:
C       setenv DATAFILE DATA1.DAT
C
      OPEN(9,FILE='DATAFILE')
```

This example shows the minimum information needed in an OPEN statement to open an existing file. The unit number is 9 and the file name is DATA1.DAT. The default status is 'OLD'; if DATA1.DAT does not exist, file FILE.FT9F001 is created. The default access is 'SEQUENTIAL' and the default file format is 'FORMATTED'.

```
C     In Bourne shell:
C       myfile=OUT.DAT
C       export myfile
C
C     In C shell:
C       setenv myfile OUT.DAT
C
      OPEN(UNIT=9,FILE='myfile',STATUS='NEW',
     +     FORM='FORMATTED',ACCESS='DIRECT')
```

This OPEN statement opens the new file OUT.DAT as unit 9. The file format is 'FORMATTED' and the file access is 'DIRECT'.

```
C     In Bourne shell:
C       printer=/dev/lp
C       export printer
C
C     In C shell:
C       setenv printer /dev/lp
C
      OPEN(8,FILE='printer')
```

This OPEN statement opens the printer so that it can be written to directly from a program. The printer here is assigned to unit 8.

CLOSE Statements

A "CLOSE statement" disassociates a specified unit from a file, thereby preventing input/output from being directed to that unit number until it is reopened. The form of a CLOSE statement is:

CLOSE (*arglist*)

arglist
is an argument list of specifiers that must contain one unit specifier. It can also contain one of each of the other allowable specifiers.

The forms of the unit specifier and other allowable specifiers are:

[UNIT=] *u*
is an external unit specifier described in "Unit Specifiers" on page 10-9.

IOSTAT = *ios*
is an input/output status specifier described in "Input/Output Status Specifiers" on page 10-12.

ERR = *s*
is an error exit specifier described in "Error Exit Specifiers" on page 10-12.

STATUS = *dis*
specifies the disposition of the file after it is closed and applies only to files opened with STATUS='NEW'. The *dis* is a character expression whose value must be either 'KEEP' or 'DELETE'. The default is 'KEEP'.

A file opened with STATUS='NEW' is a temporary file and is discarded if STATUS='DELETE' is specified. A FORTRAN program that ends normally automatically closes all opened files as if a CLOSE statement with STATUS='KEEP' had been specified. STATUS='KEEP' cannot be specified for a file that was opened with STATUS='SCRATCH'.

Examples:

```
CLOSE(8)
```

This example shows the minimum information needed in a CLOSE statement to close an opened file. This statement closes and saves the file opened as unit 8.

```
CLOSE(UNIT=9,STATUS='DELETE')
```

This statement deletes an opened file that had either previously existed or had been created with an OPEN statement (in which case the file was temporary to the program).

READ, WRITE, and PRINT Statements

"READ statements" are used for inputting data. "WRITE and PRINT statements" are used for outputting data. Note that the PRINT statement does not refer to "printing" on the system printer; it is used to write to the standard output device. The READ, WRITE, and PRINT statements can be categorized as format-specified, unformatted, list-directed, or namelist-directed input/output statements.

"Format-specified input/output statements" are used to direct the editing or conversion between the internal representations of the computer and the representations of character strings in a file or data item. The formatting of the data is controlled by the FORMAT statement, which is described in Chapter 11, "Format Specifications."

"Unformatted input/output statements" do not perform editing or conversion between the internal representations of the computer and representations of character strings in a file or character data item. Unformatted input/output statements execute faster than formatted input/output statements; however, unformatted data is generally not transportable to other language processors and computers.

"List-directed input/output statements" transfer data to or from a formatted record; the formatting of the data is controlled by the data types and lengths of the data items in an input/output list. This manner of treating records is more convenient than the structured manner of format-specified input/output statements when precise layout of the data is not important.

"Namelist-directed input/output statements" use a namelist instead of an input/output list to specify the names of variables or arrays to be read or

written. Namelists are declared in NAMELIST statements, described in "NAMELIST Statements — Specifying Names ✧ ✧" on page 7-23.

Format-Specified and Unformatted READ, WRITE, and PRINT Statements

The forms of the format-specified and unformatted READ, WRITE, and PRINT statements are:

READ *f* **[** , *iolist* **]**

READ (*arglist*) **[** *iolist* **]**

WRITE (*arglist*) **[** *iolist* **]**

PRINT *f* **[** , *iolist* **]**

f
is a format specifier described in "Format Specifiers ✦" on page 10-10.

iolist
is an input/output list described in "Input/Output Lists ✧" on page 10-13.

arglist
is an argument list of specifiers that must contain one unit specifier. It can also contain one of each of the other allowable specifiers.

The forms of the unit specifier and other allowable specifiers are:

[UNIT=] *u*
is a unit specifier described in "Unit Specifiers" on page 10-9.

[FMT =] *f*
is a format specifier described in "Format Specifiers ✦" on page 10-10.

IOSTAT = *ios*
is an input/output status specifier described in "Input/Output Status Specifiers" on page 10-12.

ERR = *s*
is an error exit specifier described in "Error Exit Specifiers" on page 10-12.

REC = *rn*
is a record number specifier described in "Record Number Specifiers ✧" on page 10-11. Record number specifiers can only be used on files that have been opened with ACCESS='DIRECT'.

END = *s*
is an end-of-file exit specifier described in "End-of-File Exit Specifiers" on page 10-11. This specifier is only applicable to the READ statement and cannot appear in WRITE or PRINT statements.

If an *arglist* contains a format specifier, the statement is a formatted input/output statement; if it does not, the statement is an unformatted input/output statement.

If the format specifier is an asterisk (*), the statement is a list-directed input/output statement. In this case, the record number specifier cannot appear in the statement's *arglist*. List-directed input/output can be done on an internal file.

An *arglist* cannot contain a record number specifier if the unit specifier designates an internal file. Also, an *arglist* cannot contain both a record number specifier and an end-of-file specifier.

After the last record of a file is read, the file is positioned at end-of-file.

Formatted input/output examples:

```
      INTEGER I,J,K
      .
      .
      READ(*,10) I,J,K
10    FORMAT(3I5)
```

This code reads three integers from the console by using an asterisk and a FORMAT statement labeled 10.

```
      COMPLEX C
      .
      .
      READ(3,100,REC=6,ERR=99) C
 100  FORMAT(2F8.4)
```

This code reads a complex number from the sixth record of the direct file associated with unit 3 using the FORMAT statement labeled 100. If an error occurs, a jump is made to the statement labeled 99.

```
      CHARACTER*20 STR
      .
      .
      READ 50,STR
  50  FORMAT(A)
```

This code reads a character expression from the console to the character variable STR.

```
      INTEGER I(10)
      .
      .
      WRITE(4,100) (I(J),J=1,5)
 100  FORMAT(10I5)
```

This code writes the first five elements of integer array I to unit 4 using the FORMAT statement labeled 100.

```
      WRITE(*,'(A)') 'Input your data: '
```

This code writes a character expression to the console. Note that the format is specified by a character expression and not by the label of a FORMAT statement.

```
      INTEGER RECORD
      REAL X(10)
      .
      .
      WRITE(4,20,REC=RECORD) X
 20   FORMAT(10F6.2)
```

This code writes all 10 elements of real array X to record number RECORD of the direct file associated with unit 4. The FORMAT statement labeled 20 is used.

```
      INTEGER I,J,K
      .
      .
      PRINT 10,I,J,K
 10   FORMAT(3I5)
```

This code prints the three integers I, J, and K to the console using the FORMAT statement labeled 10.

```
      CHARACTER*3 FARM
      DATA FARM /'(A)'/
      .
      .
      PRINT FARM,'Title'
```

This code prints the word "Title" to the console. The format is specified by the character variable FARM.

Unformatted input/output examples:

```
      WRITE(9) 10,3.14159,'HELLO'
```

This code writes the values 10, 3.14159, and "HELLO" to the file opened to unit 9. The file must have been opened with FORM='UNFORMATTED'. Since no format is specified, the output is unformatted; that is, no conversion between the internal representations of the values and the equivalent character expressions takes place. Therefore, the values written to unit 9 are (in hexadecimal radix form):

00001B1040490FD048454C4C4F

The 00001B10 is the internal representation for the integer 10, 40490FD0 is the internal representation for the real number 3.14159, and 48454C4C4F is the internal representation for the character expression "HELLO".

```
INTEGER I,J,K
.
.
READ(8) I,J,K
```

This code reads in three integers from the file opened as unit 8. The file must have been opened with FORM='UNFORMATTED', and the data in the file must be in unformatted form. No conversion between formatted and unformatted data takes place.

List-Directed READ Statements ✦ ✧

The parameters in a list-directed READ statement are the same as those for a formatted and unformatted READ statement, except that the format specifier is always an asterisk (*).

Data is read into storage locations as specified in a READ statement's input list. Input data consists of any number of value separators and input values.

A "value separator" is used to delimit a value in a list-directed input list. A value separator can be:

- a comma (,) with optional spaces on either side
- a slash (/) with optional spaces on either side
- one or more spaces between constants or after the last constant in the list
- an end-of-record, which appears as a space between two constants.

A comma is used to separate values. Two consecutive commas specifies a null value. A null value indicates that the corresponding list element remains unchanged.

A slash discards the remaining items in the input list and substitutes null values in their place.

A value separator adjacent to an end-of-record is not considered a null value.

An "input value" to a list-directed READ statement can be a constant, a null value, or can take one of two other forms.

One form is *r* *, where *r* is an unsigned nonzero integer constant that indicates *r* occurrences of a null value.

The other form is *r* * *c*, where *r* is an unsigned nonzero integer constant and *c* is a constant; this form indicates *r* occurrences of *c*. For example, 3*5 indicates 5, 5, 5.

Values in list-directed input cannot contain embedded spaces, except for character constants in which spaces are significant. A list-directed input value can be a:

null value
: is indicated by two consecutive value separators, by no characters or spaces preceding the first value separator in an input list, or by an *r* * form. List-directed input values that are null have no effect on the definition status of variables in the corresponding input/output list; that is, variables in the input/output list that are defined remain defined and variables that are not defined remain undefined. A null value cannot be used as the real or imaginary part of a complex constant.

integer value
: which must have the form of an integer constant.

real value
: which must have the form of a real constant. The decimal point can be omitted, in which case the number is assumed not to have a fractional part.

complex value
is represented by an ordered pair of real numbers that are separated by a comma and enclosed in parentheses. Spaces can surround the real numbers and an end-of-record can appear before or after the comma separating the numbers.

character string value
is a string of characters enclosed in apostrophes. Any spaces are significant and are part of the constant. An apostrophe that is part of the character string is represented by by two consecutive apostrophes in the string. Note that a character constant not ended by an apostrophe causes erroneous results.

Character constants can span record boundaries; that is, a character constant can be continued over as many records as needed. A record boundary in a character constant does not add any additional characters to the value.

If a character constant as read is longer than the length of its corresponding input/output list item, the character constant is truncated to fit. If a character constant is shorter than its corresponding input/output list item, the character constant is placed left-justified in the variable and the remaining positions in the variable are filled with spaces.

logical value
is represented by a T for .TRUE. or an F for .FALSE.. The T or F character can be preceded by a period (.) and can be followed by other characters except commas and slashes.

Examples:

```
      INTEGER I,J,K
      LOGICAL CHOICE
      .
      .
      READ *,I,J,K,CHOICE
      .
      END

Input data:  110    220, 330  T
```

In this example, I gets the value 110, J gets the value 220, K gets the value 330, and CHOICE gets the value .TRUE. The data is entered from the console. Note that both spaces and commas are used as value separators.

```
      COMPLEX C
      DIMENSION X(2), IN(4)
      CHARACTER*10 STR
      .
      .
      IN(1)=5
      IN(2)=7
      IN(3)=12
      IN(4)=16
      READ (7,*) C,X,STR,IN
      .
      END

Input data:
  (1.4,0.32)  1.11  2.22  'DATA'  2*  2*6
```

In this example, C gets the value (1.4,0.32), X(1) gets the value 1.11, X(2) gets the value 2.22, STR gets the value 'DATA', IN(1) retains the value of 5, IN(2) retains the value of 7, and IN(3) and IN(4) each get the value 6. The data is read from the file opened as unit 7.

List-Directed READ Statements with Internal Files: The READ statement transfers data from one area of internal storage to one or more other areas of internal storage. The area in internal storage that is read from is called an internal file. The type of the items specified in this statement determines the conversion to be performed.

An internal, list-directed READ statement starts data transmission at the beginning of the storage area indicated by the internal unit specifier. One value in the internal file is transferred to each item of the list in the order they are specified. The conversion to be performed depends on the type and length of the name of the item in the list. Data transmission stops when data has been moved to every item of the list or when the end of the storage area is reached.

If the internal unit specifier is a character variable, a character array element name, or a character substring name, it is treated as one record. If

the internal unit specifier is a character array name, each array element is treated as one record.

The length of a record is the length of the character variable, character substring name, or character array element specified by the internal unit specifier when the READ statement is executed.

If a record contains more data than is necessary to satisfy all the items in the list and the associated format identifier, the remaining data is ignored. The next READ statement with list-directed input/output starts with the next record if no other input/output statement is executed on that file.

If a record contains less data than is necessary to satisfy the list and the record does not have a slash after the last element, an error is detected. If the list has not been satisfied when a slash separator is found, the remaining items in the list remain unaltered and execution of the READ is terminated.

If the list indicates that more data items are to be moved and none remain in the character variable, character substring, or last array element of a character array, an end of file is detected. If an array element is not last and the list requires more data items than that element contains, the items are taken from the next array element.

Control is transferred to the statement specified by END when the end of the file is encountered; that is, when there is insufficient data in the character variable or array to satisfy the requirements of the input/output list. No indication is given of the number of list items read into before the end of the file was encountered. If IOSTAT=*ios* is specified, a negative integer value is assigned to *ios* when an end of file is detected. Execution continues with the statement indicated by END (if specified) or with the next statement if END is not specified. If END and IOSTAT are both omitted, program execution ends when the end of the file is encountered.

Example:

```
  1  CHARACTER*50 CHARVR
  2  READ (UNIT=7, FMT=100) CHARVR
100  FORMAT (A50)
  3  READ (UNIT=CHARVR, FMT=*) A1, A2, A3
```

Statement 1 defines a character variable, CHARVR, of fixed-length 50. Statement 2 reads 50 characters of input into CHARVR. Statement 3 reads from CHARVR, performs the conversion (depending on the type and length of the names of the items in the list), and assigns values to A1, A2, and A3.

List-Directed WRITE and PRINT Statements ✦ ✧

The parameters in list-directed WRITE and PRINT are the same as those for formatted and unformatted WRITE and PRINT statements, except that their format specifiers are always asterisks (*).

Data is transferred from the variables specified in the output list to a specified unit. In general, values are written to the output device in a manner consistent with list-directed input. One exception is character string data, which cannot be re-read by a list-directed input statement after being written by a list-directed WRITE or PRINT statement.

FORTRAN starts new records when necessary and never generates slashes or null values on list-directed output. Values in the output are separated by three spaces, except for character values, which are not preceded or followed by any spaces. The data types and how they are generated in list-directed output are:

integer value
: is generated as an I*w* edit-descriptor. The *w* is the minimum number of characters required to print the integer value.

real or double-precision value
: is generated as an F edit-descriptor or an E edit-descriptor depending on the magnitude of the number. The specific edit-descriptors used are described in this table:

Range of Number	Edit-Descriptor Used	
	Real	Double-Precision
1.0 ≤ val < 10.0	0PF9.6	0PF17.14
val < 1.0 or val ≥ 10.0	1PE13.6E2	1PE22.14E3

complex value
is generated as a pair of real numbers enclosed in parentheses with a comma separating the real and imaginary parts.

character string value
is generated as a string of characters. However, it is not contained in apostrophes and therefore cannot be re-read using list-directed input.

logical value
is generated as T for .TRUE. and F for .FALSE..

Examples:

```
PRINT *,1000,0.12345
PRINT *,(1.1,2.2),.TRUE.,'Chars'
```

In this example, the output to the console is:

```
1000   +1.234500E-01
(+1.100000E+00,+2.200000E+00)   TChars
```

```
INTEGER I
REAL X(2)
.
.
I = -12
X(1) = 1.2E-12
X(2) = 2.4E+12
WRITE(9,*) I,X
```

In this example, the output to the file opened as unit 9 is:

```
-12   +1.200000E-12   +2.400000E+12
```

List-Directed WRITE Statements with Internal Files: The WRITE statement transfers data from one or more areas of internal storage to another area of internal storage. The receiving area is called an internal file. This statement can be used to convert numeric data to character data. The type of item specified in the statement determines the conversion to be performed.

An internal WRITE statement starts data transmission at the beginning of the storage area indicated by the internal unit specifier. Each item of the list is transferred to the internal file in the order it is specified. The conversion to be performed depends on the type and length of the name of the item in the list. Data transmission stops when every item of the list has been moved to the internal file or when the end of the internal file is reached.

If the internal unit specifier is a character variable, a character array element name, or a character substring name, it is treated as one record. If the internal unit specifier is a character array name, each array element is treated as one record. If a record is not large enough to hold all the converted items, a new record is started for any noncharacter item that will exceed the record length. For character items, as much as can be put in the record is written there, and the remainder is written at the beginning of the next record.

The length of a record is the length of the character variable, character substring name, character array element specified by the internal unit specifier when the WRITE statement is executed.

Example:

```
1     CHARACTER*120 CHARVR
2     WRITE (UNIT=CHARVR, FMT=*) A1, A2, A3
3     FORMAT (A120)
4     WRITE (UNIT=3, FMT=3) CHARVR
```

Statement 1 defines a character variable, CHARVR, of fixed-length 120. Statement 2 writes the internal file represented by CHARVR by converting the values in A1, A2, and A3. Statement 4 writes the 120 characters of CHARVR to the unit 3 external file.

Namelist-Directed READ Statements ✧ ✧

The forms of namelist-directed READ statements are:

READ *name*

READ (*arglist*)

name
is a namelist name declared in a NAMELIST statement. NAMELIST statements are described in "NAMELIST Statements — Specifying Names ✧ ✧" on page 7-23.

arglist
is an argument list of specifiers that must contain one unit specifier and one format specifier.

The forms of the unit, format, and other allowable specifiers are:

[UNIT =] *u*
is a unit specifier described in "Unit Specifiers" on page 10-9.

[FMT =] *name*
is a namelist name declared in a NAMELIST statement. NAMELIST statements are described in "NAMELIST Statements — Specifying Names ✧ ✧" on page 7-23.

IOSTAT = *ios*
is an input/output status specifier described in "Input/Output Status Specifiers" on page 10-12.

ERR = *s*
is an error exit specifier described in "Error Exit Specifiers" on page 10-12.

END = *s*
is an end-of-file exit specifier described in "End-of-File Exit Specifiers" on page 10-11.

In the "**READ** *name*" form of this statement, the unit specifier defaults to the console.

When the namelist input data is on internal files, only the second form of this statement is allowed because a unit specifier must be used.

Namelist Input Data: To be read using a namelist list, input data must be in this form:

```
&namelist  value-assignment  [ , value-assignment ] ... &END
```

namelist
is a namelist name declared in a NAMELIST statement. It cannot contain spaces and must be contained within a single record. The NAMELIST statement is described in "NAMELIST Statements — Specifying Names ✧ ✧" on page 7-23.

value-assignment
is a value assignment. Value assignments are separated by commas and take either of these forms:

```
name = constant

arrayname = constant  [ , constant ] ...
```

name
is an array element name or a variable name.

constant
is an integer, real, complex, logical or character constant. Embedded spaces are not permitted, except:

- in character constants. Any spaces are part of the constant.
- in complex constants. Spaces can surround the "real" numbers.

A logical constant can be in the form T for .TRUE. or F for .FALSE.. Subscripts must be integer constants, and can be surrounded by spaces.

arrayname
is the name of an array.

Variable names and array names cannot contain spaces and must be contained within a single record.

The first character of each input data record is ignored. Data should be entered starting at the second position of each record.

Any number of spaces can surround the equal sign in value assignment.

The end of a record is equivalent to a space character. However, character constants can span record boundaries, which do not add any additional characters to the value. Numeric constants and logical constants can not span record boundaries.

In a list, constants are separated by commas. Values are assigned to an array in a linear fashion. The first value in the list is assigned to the first element of the array, the second value is assigned to the second element of the array, and so on. The number of constants must be less than or equal to the number of elements in the array. For multi-dimensional array element sequencing, see "Array Storage Sequence" on page 4-8.

A null value is denoted by two consecutive commas, by a comma with no preceding value, or by a trailing comma. A null value indicates that the value of the variable or corresponding array element remains unchanged. A null value cannot be used as the "real" or "imaginary" part of a complex constant.

Successive occurrences of the same constant can be represented in the form *r* * *c*, where *r* is an unsigned nonzero integer constant and *c* is a constant; this form indicates *r* occurrences of *c*. For example, 3*5 indicates 5, 5, 5.

The variable names and array names specified in the input file must appear in the namelist list, but the order is not significant. A name that has been made equivalent to a name in the input data cannot be substituted for that name in the namelist list. The list can contain names of items in common blocks but must not contain dummy argument names.

It is not necessary for the input file to define every name in the namelist list. The values of items that do not appear in the input data are unchanged.

Operations: The namelist-directed READ statement performs the following operations:

1. Reads data sequentially from an external or internal file until it finds an ampersand (&), followed by the specified namelist name, followed by a space.

2. Matches the item names that appear in the input records with the item names specified in the corresponding NAMELIST statement.

   - If a match is found, the READ statement converts, if necessary, the value associated with the item in the input to the data type of the item according to the rules for data type conversion for arithmetic assignment statements (see Figure 6-1 on page 6-2).

     Then, the READ statement assigns the converted value to the variable or corresponding array element. The values are converted and are assigned to the associated items in the order in which the items appear in the input records.

   - If a match is not found, an error is detected.

3. Stops processing when the end of the data group, specified by "&END", is encountered.

An internal namelist-directed READ statement starts data transmission from the beginning of the internal file.

Example:

```
C     Suppose a namelist is declared
C     as:  NAMELIST /NAM1/ A, B, C
C
      READ (UNIT=2, FMT=NAM1)
```

Input data (a ␣ indicates a blank):

```
␣&NAM1 A=1,
␣B=1, C=3,
␣&END
```

In this example, note that the format specifier specifies a namelist name (NAM1) instead of a statement label of a FORMAT statement, an integer variable name, a character expression, an array name, or an asterisk (*). Also note that *iolist* is omitted. The names of variables or arrays to be read are supplied by the namelist name.

Namelist-Directed WRITE and PRINT Statements ✧ ✧

The forms of namelist-directed WRITE and PRINT statements are:

WRITE (*arglist*)

PRINT *name*

name
is a namelist name declared in a NAMELIST statement. NAMELIST statements are described in "NAMELIST Statements — Specifying Names ✧ ✧" on page 7-23.

arglist
is an argument list of specifiers that must contain one unit specifier and one format specifier.

The forms of the unit, format, and other allowable specifiers are:

[UNIT =] *u*
is a unit specifier described in "Unit Specifiers" on page 10-9.

[FMT =] *name*
is a namelist name declared in a NAMELIST statement. NAMELIST statements are described in "NAMELIST Statements — Specifying Names ✧ ✧" on page 7-23.

IOSTAT = *ios*
is an input/output status specifier described in "Input/Output Status Specifiers" on page 10-12.

ERR = *s*
is an error exit specifier described in "Error Exit Specifiers" on page 10-12.

END = *s*
is an end-of-file exit specifier described in "End-of-File Exit Specifiers" on page 10-11.

In the namelist-directed PRINT statement, the unit specifier defaults to the console.

When the namelist output data is directed to internal files, only the WRITE statement is allowed because a unit specifier must be used.

Operations: The namelist-directed WRITE statement performs the following operations:

1. Retrieves data from internal storage (items specified by the namelist).
2. Converts data from binary to character form according to the data types of the items specified in the corresponding NAMELIST statement.
3. Writes the converted data to the external or internal file.

An internal namelist-directed WRITE statement starts data transmission from the beginning of the internal file.

Namelist Output Data: When output data is written using a namelist list, it is written in a form that can be read using a namelist list. All variable and array names specified in the namelist list and their values are written out, each according to its type. Character data is included between apostrophes. The fields for the data are made large enough to contain all the significant digits. The values of a complete array are written out in columns.

The order of data output is controlled by the order in which items are specified in the namelist list. The first item in the list is written out first, the second item is written out second, and so on.

Example:

```
C     Suppose a namelist is declared
C     as:  NAMELIST /NAM1/ A, B, C
C     and the input data is:
C     &NAM1 A=1,B=1,C=3,&END

      WRITE (UNIT=3, FMT=NAM1)
```

Output (assuming A, B, and C are real variables):

```
 &NAM1 A=+1.000000,B=+2.000000,C=+3.000000,&END
```

BACKSPACE, ENDFILE, and REWIND Statements — Positioning Files

Files can be positioned explicitly in FORTRAN using BACKSPACE, ENDFILE, and REWIND statements.

A "BACKSPACE statement" backspaces a sequential file by one record. The two forms of the BACKSPACE statement are:

BACKSPACE *u*

BACKSPACE (*arglist*)

An "ENDFILE statement" writes an endfile record on a sequential file. The two forms of the ENDFILE statement are:

ENDFILE *u*

ENDFILE (*arglist*)

A "REWIND statement" positions or re-positions a sequential file at its first record. The two forms of the REWIND statement are:

REWIND *u*

REWIND (*arglist*)

In all the forms:

u
is a unit number.

arglist
is an argument list of specifiers that must contain one unit specifier. It can also contain one of each of the other allowable specifiers.

The forms of the unit specifier and other allowable specifiers are:

[UNIT=] *u*
is an external unit specifier described in "Unit Specifiers" on page 10-9.

IOSTAT = *ios*
is an input/output status specifier described in "Input/Output Status Specifiers" on page 10-12.

ERR = *s*
is an error exit specifier described in "Error Exit Specifiers" on page 10-12.

BACKSPACE Statements: A BACKSPACE statement positions the file associated with a specified unit before the preceding record. If there is no preceding record, the file position is not changed. If the preceding record is the endfile record, the file is positioned before the endfile record.

BACKSPACE statements can only be applied to sequential files associated with blocked devices, and can be applied to unformatted files.

Examples:

```
BACKSPACE 4
```

This code backs up one record on the sequential file attached to device 4.

```
BACKSPACE (UNIT=4,ERR=99)
```

This code backs up one record on the sequential file attached to device 4 and branches to the statement labeled 99 if an error occurs.

ENDFILE Statements: An ENDFILE statement writes an endfile record as the next record on the sequential file associated with a specified unit. The file is then positioned after the endfile record, thereby prohibiting further sequential data transfers to that file until either a BACKSPACE or a REWIND statement is executed.

Examples:

```
ENDFILE 4
```

This code writes an endfile record on the sequential file associated with unit 4.

```
INTEGER IOS
.
.
ENDFILE (UNIT=4,IOSTAT=IOS)
```

This code writes an endfile record on the sequential file associated with unit 4 and also returns the input/output status code in the integer variable IOS.

REWIND Statements: A REWIND statement positions the sequential file associated with a specified unit at its first record.

Example:

```
REWIND 4
```

This code positions the sequential file associated with unit 4 at its first record.

INQUIRE Statements — Obtaining File Properties

An "INQUIRE statement" obtains information about the properties of a particular named file or about a file's association to a particular unit. INQUIRE statements can be executed before, while, and after a file is associated with a unit. Any values assigned as a result of an INQUIRE statement are values that are current at the time the statement is executed. The two types of INQUIRE statements are "inquire by file" and "inquire by unit."

The form of an "inquire by file" INQUIRE statement is:

INQUIRE (*arglist*)

arglist
is an argument list of specifiers that must contain one file specifier. It can also contain one of each of the other allowable specifiers.

The form of a file specifier is:

FILE = *filname*

filname
is the symbolic name of the file that is the subject of inquiry.

The *filname* is a character expression that, when its trailing spaces are removed, is an environment variable recognizable to the operating system. The named file does not have to exist nor does it have to be associated with a unit.

The form of an "inquire by unit" INQUIRE statement is:

INQUIRE (*arglist*)

arglist
is an argument list of specifiers that must contain one unit specifier. It can also contain one of each of the other allowable specifiers.

The form of a unit (UNIT=) specifier is described in "Unit Specifiers" on page 10-9. The unit specified does not have to exist nor does it have to be associated with a file. If the unit is associated with a file, the inquiry is made about the association and the file it is associated with.

The other allowable specifiers make up the remainder of an INQUIRE statement's argument list. Only one of each type of specifier is allowed per argument list and the same variable name cannot be given to more than one specifier. All specified integer variables must have a data type of either

INTEGER or INTEGER*4. All specified logical variables must have a data type of either LOGICAL or LOGICAL*4.

The forms of the other allowable specifiers are:

IOSTAT = *ios*
is an input/output status specifier described in "Input/Output Status Specifiers" on page 10-12.

ERR = *s*
is an error exit specifier described in "Error Exit Specifiers" on page 10-12. An INQUIRE statement does not cause any error conditions.

EXIST = *ex*
determines if a specified file exists. The *ex* is a logical variable that is set by an INQUIRE statement to either .TRUE. or .FALSE..

OPENED = *od*
determines if a specified file is opened. The *od* is a logical variable that is set by an INQUIRE statement to either .TRUE. or .FALSE..

NUMBER = *num*
determines the number of the external unit currently associated with the file. The *num* is an integer variable that is set to the number of the external unit. If the file is not associated with a unit, the value of *num* is undefined.

NAMED = *nmd*
determines if the file has a name. The *nmd* is a logical variable that is set to either .TRUE. or .FALSE..

NAME = *fn*
determines the name of the file. The *fn* is a character variable that is set to the name. If the file does not have a name, the value of *fn* is undefined.

ACCESS = *acc*
determines whether the file can be accessed sequentially or directly. The *acc* is a character variable that is assigned either the value 'SEQUENTIAL' or 'DIRECT'. If the file is not associated with a unit, the value of *acc* is undefined.

SEQUENTIAL = *seq*
determines if the file can be accessed sequentially. The *seq* is a character variable that is assigned either the value 'YES' or 'NO'. If FORTRAN cannot determine what access methods are allowed for the file, *seq* is assigned the value 'UNKNOWN'.

DIRECT = *dir*
determines if the file can be accessed directly. The *dir* is a character variable that is assigned either the value 'YES' or 'NO'. If FORTRAN cannot determine what access methods are allowed for the file, *dir* is assigned the value 'UNKNOWN'.

FORM = *fm*
determines the format of the file. The *fm* is a character variable that is assigned the value 'FORMATTED' for a formatted file and is assigned the value 'UNFORMATTED' for an unformatted file. If the file is not associated with a unit, the value of *fm* is undefined.

FORMATTED = *fmt*
determines if the file is a formatted file. The *fmt* is a character variable that is assigned either the value 'YES' or 'NO'. If FORTRAN cannot determine if the file is formatted, *fmt* is assigned the value 'UNKNOWN'.

UNFORMATTED = *unf*
determines if the file is an unformatted file. The *unf* is a character variable that is assigned either the value 'YES' or 'NO'. If FORTRAN cannot determine if the file is unformatted, *unf* is assigned the value 'UNKNOWN'.

RECL = *rcl*
determines the record length of a direct file. The *rcl* is an integer variable that is assigned the value of the record length. If the file is not associated with a unit or if the file is not a direct file, the value of *rcl* is undefined.

NEXTREC = *nr*
determines the number of the next record to be read or written on a direct file. The *nr* is an integer variable that is assigned the value of the next record number. If the file is associated with a unit but no data transfer has taken place, *nr* is assigned the value 1. If the file is not a direct file or if the position cannot be determined (possibly due

to an error), the value of *nr* is undefined. If a NEXTREC specifier is used in an INQUIRE statement, a RECL specifier must also be used.

BLANK = *blnk*
determines the file's default treatment of blanks (spaces). The *blnk* is a character variable that is assigned the value 'NULL' if all spaces in numeric input fields are ignored and is assigned the value 'ZERO' if all spaces other than leading spaces are treated as zeros. If the file is not associated with a unit or if the file is not a formatted file, the value of *blnk* is undefined.

Example:

```
c     This program reads in a symbolic file
c     named entered by the user and writes
c     the values of X, Y, and Z to that file.
c
      INTEGER NUM
      LOGICAL EX,OD
      CHARACTER*20 FNAME
      .
      .
      READ(*,'(A)') FNAME
      INQUIRE(FILE=FNAME,EXIST=EX,OPENED=OD,
     +        NUMBER=NUM)
      IF (.NOT.EX) THEN
          OPEN(4,FILE=FNAME,STATUS='NEW')
          NUM = 4
      ELSEIF (.NOT.OD) THEN
          OPEN(4,FILE=FNAME,STATUS='OLD')
          NUM = 4
      ENDIF
      WRITE(NUM,100) X,Y,Z
      .
      .
      END
```

In this example, the INQUIRE statement first tests to see if the file exists. If it does not, the file is opened with a status of 'NEW'. If the file does exist then a test is conducted to see if it is open. If it is not open, it is opened with a status of 'OLD'. The unit number is used to write out the values of X, Y, and Z.

R1 Mode Specifics

This section describes the instances in which R1 mode differs from IBM mode (the default mode).

Internal Files

- List-directed input/output statements with internal files are not allowed.
- Namelist-directed input/output statements with internal files are not allowed.

Format Specifiers

- The *f* cannot be an array name.

OPEN Statements

- **FILE** = *fname*
  The *fname* is a character expression and, if omitted, unit 5 is connected to 'stdin', unit 6 is connected to 'stdout', and unit *n* (except units 0, 5, and 6) is connected to the file 'fort.*n*'. The *fname* is a file name and no shell variables are needed.
- When opened, a sequential file is positioned at end-of-file.

List-Directed READ Statements

- List-directed READ statements with internal files are not allowed.

List-Directed WRITE and PRINT Statements

- List-directed WRITE statements with internal files are not allowed.

Namelist-Directed READ Statements

- The NAMELIST statement is not allowed.

Namelist-Directed WRITE and PRINT Statements

- The NAMELIST statement is not allowed.

VX Mode Specifics

This section describes the instances in which VX mode differs from IBM mode (the default mode).

Internal Files

- List-directed input/output statements with internal files are not allowed.
- Namelist-directed input/output statements with internal files are not allowed.
- Internal files can also be referenced by ACCEPT and TYPE statements.

Units

- Internal unit specifiers can also be used in ACCEPT and TYPE statements.

Parameters of Input/Output Statements

- A "namelist specifier" indicates that namelist-directed input/output is being used, and identifies the namelist name of the list of variables or array names to be read or written. The form of the namelist specifier is:

```
[ NML = ] nmlname
```

nmlname
is a namelist name.

The keyword NML= is optional only if the namelist specifier is the second item in the argument list of specifiers and the first item is a unit

specifier without the optional UNIT= keyword. A namelist specifier cannot be used in a statement that contains a format specifier.

Record Number Specifiers

The record number specifier can also take the form:

```
'rn
```

The variable *rn* is described in "Record Number Specifiers ✧" on page 10-11.

Input/Output Lists

- An "input/output list" specifies the objects whose values are transferred by READ, WRITE, PRINT, ACCEPT, and TYPE statements.

Input/Output Objects

- "Input/output objects" can be specified as items in input/output lists of READ, WRITE, PRINT, ACCEPT, and TYPE statements.

Implied DO Loops

- "Implied DO loops" can be specified as items in input/output lists of READ, WRITE, PRINT, ACCEPT, and TYPE statements.

Input/Output Statements

- The "ACCEPT statement" is also allowed, and has the same effect as a READ statement. The ACCEPT statement is only used for sequential-access and can be used only as a PRINT statement. The forms of an ACCEPT statement are:

```
ACCEPT f [ , iolist ]

ACCEPT * [ , iolist ]   (for list-directed input/output)

ACCEPT nl               (for namelist-directed input/output)
```

nl
is a namelist specifier without the "NML=" keyword specified. Namelist specifiers are described in "Parameters of Input/Output Statements" on page 10-49.

See "READ, WRITE, and PRINT Statements" on page 10-20 for descriptions of the other variables.

- The "TYPE statement" is also allowed, and has the same effect as a PRINT statement. The forms of a TYPE statement are:

```
TYPE f [ , iolist ]

TYPE * [ , iolist ]   (for list-directed input/output)

TYPE nl               (for namelist-directed input/output)
```

nl
is a namelist specifier without the "NML=" keyword specified. Namelist specifiers are described in "Parameters of Input/Output Statements" on page 10-49.

See "READ, WRITE, and PRINT Statements" on page 10-20 for descriptions of the other variables.

OPEN Statements

- **FILE** = *fname*
  The *fname* is a character expression and, if omitted, an anonymous file with a status of 'SCRATCH' is created, which is automatically deleted when the file is closed or when the program ends. The *fname* is a file name and no shell variables are needed.

List-Directed READ Statements

- List-directed READ statements with internal files are not allowed.

List-Directed WRITE and PRINT Statements

- List-directed WRITE statements with internal files are not allowed.

Namelist-Directed READ Statements

- Namelist-directed input/output with internal files is not allowed.
- A dollar sign ($) can also be used to indicate the beginning and end of a group of data records.
- Spaces and tabs can also be used to separate namelist value assignments and to separate constants in a constant list. Any number of consecutive spaces is equivalent to a single space.

- A comma preceded or followed by spaces is equivalent to a single comma.
- When a list of values is assigned to an array element, the assignment begins with the specified array element, rather than with the first element of the array.
- "END" is an optional part of the last delimiter (&END).
- A T or F representing a logical constant can be preceded by a period and followed by other characters.

Namelist-Directed WRITE and PRINT Statements

- Namelist-directed input/output with internal files is not allowed.

Chapter 11. Format Specifications

This chapter describes formatted input/output and the FORMAT statements available in FORTRAN. To understand these concepts, you need to be familiar with the FORTRAN file system, units, records, access methods, and input/output statements as described in Chapter 10, "Input and Output."

"Format specifications" are used in conjunction with READ, WRITE, and PRINT format-specified input/output statements to direct the editing or conversion between the internal representations of the computer and the representations of character strings in a file or character data item.

Overview of FORMAT Statements ✦

READ, WRITE, and PRINT statements that specify formats are formatted input/output statements. A format can be specified by:

- the label of a FORMAT statement
- the name of an integer variable that is assigned the label of a FORMAT statement by an ASSIGN statement
- a character variable, character array element, character array name, or character expression that has the form of a format specifier
- an array name with a data type of integer, real, double-precision, logical, or complex. The data must be a valid format specification. The length of the format specification may exceed the length of the first element of the array; it is considered the concatenation of all the elements of the array in the order given by array element ordering.
- an asterisk (*), which indicates list-directed input/output.

Examples:

```
      WRITE(*, 990) LAKE, STREAM, OCEAN
 990  FORMAT(I5, 2F10.2)
```

This code specifies a format by referencing a FORMAT statement.

```
      ASSIGN 880 TO INTVAR
 880  FORMAT(I5, F10.2, I5)
      WRITE(*,INTVAR) MARK, GLEN, MAUREEN
```

This code specifies a format by assigning the label of a FORMAT statement to an integer variable.

```
      CHARACTER*9 CHARVAR
      CHARVAR = '(F10.2, I5, F10.2)'
      WRITE(*,CHARVAR) SOLID, LIQUID, GAS
```

This code specifies a format by using a character variable that has the form of a format specifier.

```
      CHARACTER*7 CHAREXP
      DATA CHAREXP /'2I5, I3'/
      WRITE(*, '(' // CHAREXP // ')') IRENE, JANET, KAREN
```

This code specifies a format by using a character expression that has the form of a format specifier.

```
      CHARACTER*4 FMT
      IWIDE = INT(LOG10(REAL(IABS(I))))+1
      IF (I.LT.0) IWIDE = IWIDE + 1
      FMT = '(I' // CHAR(ICHAR('0')+IWIDE) // ')'
      WRITE(*,FMT) I
```

This code computes a format specification from the size of the integer to be printed out. The common log of the integer + 1 determines how many columns wide the integer is.

```
WRITE(*,*) CAR, BUS, TRAIN
```

This code specifies a format by using an asterisk (*), and is a list-directed write.

A FORMAT statement must have a label. Like all nonexecutable statements, FORMAT statements cannot be the targets of branching statements.

A "format list" contains a list of format specifications that are separated by commas. Format lists are enclosed in parentheses and can include format specifications that are:

- repeatable edit-descriptors
- nonrepeatable edit-descriptors
- nested format specifications.

In general, repeatable edit-descriptors are used for the conversion of individual input/output list items between internal storage and character strings. Nonrepeatable edit-descriptors interact directly with a record to control the format and are not associated with input/output list items.

Repeatable and nonrepeatable edit-descriptors can be grouped in a nested format specification that consists of up to 10 levels of nested parentheses within the outermost level.

A nested format specification is repeated when a nonzero unsigned integer constant, known as a "repeat factor", precedes its left parenthesis. A repeatable edit-descriptor is repeated when a repeat factor appears directly before or directly after its left parenthesis. Characters following matching right parenthesis are ignored.

The forms of the repeatable edit-descriptors are:

I*w* and I*w.m*	integer editing
F*w.d*	real editing
E*w.d* and E*w.d*E*e*	real editing
D*w.d*	real editing
G*w.d* and G*w.d*E*e*	real editing
L*w*	logical editing
A and A*w*	character editing
Z*w*	hexadecimal editing

The *d* and *m* are unsigned integer constants. The *w* and *e* are unsigned nonzero integer constants.

The forms of the nonrepeatable edit-descriptors are:

'*xxxxxxx*'	apostrophe editing
*n*H*xxxxxxxxxxxx*	Hollerith editing
T*c*, TL*c*, and TR*c*	tabbing to columns
*n*X	inserting spaces
/	starting a new record
:	conditionally terminating a format list
*k*P	scale-factor editing
BN and BZ	blank interpretation
S, SS, and SP	sign control

The *x* represents any printable character. The *n* and *c* are unsigned, nonzero integer constants. The *k* is an optionally signed integer constant.

Interactions Between Format Lists and Input/Output Lists

To understand how the edit-descriptors control editing, you first need to understand how format lists interact with input/output lists of READ, WRITE, and PRINT statements.

An empty format list () can only be used if there are no items in the input/output list, in which case the only action the input/output statement performs is the implicit record-skipping action associated with formatted

input/output. If an input/output list is not empty, at least one repeatable edit-descriptor must appear in the format list. Each item in the input/output list is associated in turn with a repeatable edit-descriptor during the execution of the input/output statement. The nonrepeatable edit-descriptors interact directly with the record to control the format and are not associated with input/output list items.

Items in a format list are interpreted from left to right. A repeatable edit-descriptor is treated as if the edit-descriptor is present *r* times, where *r* is its repeat factor. If *r* is omitted, the repeatable edit-descriptor is present only one time. Similarly, a nested format specification is treated as if its list of items appears *r* times, and is only present one time if *r* is omitted.

Format specifications are interpreted when an input/output statement is executed. The term "format controller" is used to describe the entity that interprets the format specifications. The process formatted input/output follows is:

The format controller scans the format specifications in the format list. When a repeatable edit-descriptor is found, a corresponding item must appear in the input/output list; otherwise, the format controller ends the input/output process. If an item does appear in the input/output list, the item and the edit-descriptor are associated and input or output of that item proceeds under the format control of that edit-descriptor. If the format controller encounters a colon (:) edit-descriptor in the format list and there are no more items in the input/output list, the input/output process is ended.

If the format specification is nested, the format controller begins associating input/output items as the outermost specification level first. That is, the first input/output item is associated with the outermost specification, the second input/output item is associated with the next outermost specification, and so on.

If the format controller encounters the last right parenthesis of the format list and there are no more items in the input/output list, the input/output process is ended. If there are more input/output list items, the file is positioned at the beginning-of-record of the next record and the format controller re-scans the format, starting at the beginning of the current format specification and ending at the last right parenthesis preceding it. If there is no preceding right paren-

thesis, the format controller re-scans the format from the beginning of the format list.

Within the portion of the format list that is to be re-scanned, there must be at least one repeatable edit-descriptor. If the re-scan of the format list begins with a nested format specification that is repeated, the repeat factor indicates the number of times to repeat that nested format specification. The re-scan does not change the previously set scale-factor, the BN or BZ blank interpretation, or the S, SP, or SS sign control.

Usually, when the input/output process ends, the unread characters of an input record are skipped or an end-of-record is written on output.

Edit-Descriptors

Repeatable Edit-Descriptors ✧

Repeatable edit-descriptors are associated with items from an input/output list and are used for editing numerical, logical, and character data items.

Numeric Editing ✦ ✧: The I, F, E, D, and G edit-descriptors are used for formatting integer, real, and complex data. The rules that each of these edit-descriptors follow are:

- On input, leading spaces are not significant. The interpretation of other spaces depends on whether the BN (blank null) or BZ (blank zero) edit-descriptor is in effect, but all blank fields are always treated as the value 0. Plus signs are optional.
- On input with E and F editing, a decimal point appearing in the input field overrides the decimal point specified in the edit-descriptor.
- On output, characters generated are right-justified in the field with leading spaces if required.

- On output, the entire field is filled with asterisks (*) if the number of characters produced exceeds the field width or if the exponent exceeds its specified width.
- On output, the maximum positive or negative number is substituted for the floating-point exceptions (+infinity, -infinity, and NaN).
- Editing of complex numbers is controlled by two successive D, E, F, or G edit-descriptors; one controls the editing of the complex number's real part, and one controls the editing of its imaginary part. The two edit-descriptors for a complex number do not have to be the same.

I — Integer Editing ✦

The I edit-descriptor must be associated with an input/output list item that has an integer data type. One form of the I edit-descriptor is:

I*w*

w
is the character width of the field. On input, an optional sign can appear in the field.

The other form of an I edit-descriptor is:

I*w.m*

w
is the character width of the field.

m
specifies a minimum field width for the integer value. The *m* must not be greater than the *w*.

On input, the *m* specification has no effect. On output, if the converted integer number is shorter than *m* characters, leading zeros are placed in the field. If *m* is 0 and the integer value to be formatted is also 0, the output field consists of *w* spaces regardless of any sign control in effect.

Examples:

```
      INTEGER I,J,K
      DATA I,J,K /12,345,6789/
      .
      .
      WRITE(*,100) I,J,K
      WRITE(*,200) I,J,K
      WRITE(*,300) I,J,K
      WRITE(*,400) I,J,K
 100  FORMAT(3I3)
 200  FORMAT(3I4)
 300  FORMAT(3I5)
 400  FORMAT(3I5.4)
```

The output is:

```
12345***
 12 3456789
  12  345 6789
0012 0345 6789
```

FORMAT statement 100 specifies three characters of output for each integer. Since the integer 6789 is four characters, the rest of the field is filled with asterisks (*). FORMAT statement 200 specifies an output field of four characters and FORMAT statements 300 and 400 specify output fields of five characters. FORMAT statement 400 specifies a minimum field of four characters.

```
      INTEGER IVAL(3)
      .
      .
      READ(*,100) IVAL
      READ(*,200) IVAL
      READ(*,300) IVAL
 100  FORMAT(3I3)
 200  FORMAT(3I4)
 300  FORMAT(3I5)
```

Input data:

```
123451234512345
123451234512345
123451234512345
```

After the READ statement using FORMAT statement 100 is executed, IVAL(1) has the value 123, IVAL(2) has the value 451, and IVAL(3) has the value 234.

After the READ statement using FORMAT statement 200 is executed, IVAL(1) has the value 1234, IVAL(2) has the value 5123, and IVAL(3) has the value 4512.

Finally, after the READ statement using FORMAT statement 300 is executed, IVAL(1), IVAL(2), and IVAL(3) all have the value 12345.

F — Real Editing

The F edit-descriptor must be associated with an input/output list item that has a real or double-precision data type, or it can be associated with either part of a complex list item. The form of the F edit-descriptor is:

F*w.d*

w
is the character width of the field.

d
is the length of the number's fractional part.

The input field begins with an optional sign followed by a string of digits optionally containing a decimal point. If the decimal point is present, it overrides the *d* specified in the edit-descriptor. If the decimal point is not present, the rightmost *d* digits of the string are interpreted as following the decimal point and leading spaces are converted to zeros if necessary.

The number can be followed by an exponent. The form of the exponent is either a plus sign (+) or minus sign (-) followed by an integer, or an E or D followed by zero or more spaces followed by an optional sign and an integer. E and D are treated identically.

Input examples:

```
Input      Format      Value       Comments

-100         F6.2      -1.0          (1)
2.9          F6.2      2.9
4.E+2        F6.2      400.0         (2)

(1) The decimal point is assumed to be two
    digits in from the right.
(2) The F format can be used for exponents.
```

The output field occupies *w* digits, *d* of which follow the decimal point. The output value is controlled both by the input/output list item and by the current scale-factor described in "P — Scale-Factor Editing" on page 11-27. The output value is rounded, not truncated.

Output examples:

Note: A ƀ indicates a blank.

```
Value      Format      Output      Comments

+ 1.2      F4.3        ****          (1)
+ 1.2      F8.4        ƀƀ1.2000
.12345     F8.3        ƀƀƀƀ.123      (2)

(1) The format field can not handle any
    number > 1, so asterisks are printed.
(2) The fractional part is rounded to
    three digits.
```

E and D — Real Editing

An E or D edit-descriptor must be associated with an input/output list item that has a real or double-precision data type, or can be associated with either part of a complex list item. The forms of the E and D edit-descriptors are:

E*w.d*

D*w.d*

E*w.d***E***e*

w
is the character width of the field.

d
is the length of the number's fractional part.

e
specifies the field width of an exponent and has no effect on input.

The forms E*w.d* and D*w.d* have identical editing effects. The input field for

an E edit-descriptor is identical to that of an F edit-descriptor containing the same *w* and *d* fields.

Input examples:

```
Input       Format      Value        Comments

12.34       E8.4        12.34          (1)
.1234E2     E8.4        12.34
.1234E99    E8.4        +++++          (2)
E+20        D10.6       * error *      (3)
2.E10       E12.6E1     2.E10          (4)

(1) No E is necessary in the input field.
(2) Single-precision reals have an upper
    limit of about 3.4E+38.
(3) A leading digit is required.
(4) The two digits in the exponent field
    override the one digit specified by
    the format.
```

The form of the output field depends on the current scale-factor described in "P — Scale-Factor Editing" on page 11-27. For a scale-factor of 0, the output field contains a minus sign (-) if needed, followed by a decimal point, followed by a string of digits, followed by an exponent. The form of the exponent depends upon its size and can be any one of these:

- If "E*w.d*" is the form of the edit-descriptor and if -99 ≤ exponent ≤ 99, the output for an exponent is an E followed by a plus (+) or minus (-) sign followed by the two-digit exponent.
- If "E*w.d*" is the form of the edit-descriptor and if -999 ≤ exponent ≤ 999, the output for an exponent is a plus (+) or minus (-) sign followed by the three-digit exponent.
- If "E*w.d*E*e*" is the form of the edit-descriptor and if -(10***e*)+1 ≤ exponent ≤ (10***e*)-1, the output for an exponent is an E followed by a plus (+) or minus (-) sign followed by *e* digits of exponent with possible leading zeros.

The form E*w.d* cannot be used if the absolute value of the exponent to be printed exceeds 999.

The scale-factor controls decimal normalization of the printed E field. If the scale-factor k is in the range $-d < k \leq 0$, the output field contains exactly $d - k$ leading zeros after the decimal point and $d + k$ significant digits after that. If $0 < k < d + 2$, the output field contains exactly k significant digits before the decimal point and $d - k - 1$ places after the decimal point. Other values of k are errors.

Output examples:

Note: A ƀ indicates a blank.

```
Value      Format      Output          Comments

1234.56    E10.3       ƀƀ.123E+04
1234.56    D10.3       ƀƀ.123D+04        (1)
88.D106    E12.6       ƀ.880000+108      (2)
88.D106    E12.4E3     ƀƀ.8800E+108

(1) Ew.d and Dw.d are the same.
(2) There is no room for the E.
```

G — Real Editing ✦ ✧

The G edit-descriptor is like the E and F edit-descriptors, except that it allows the output format to adapt to the magnitude of the number being converted. Thus the G edit-descriptor gives you a choice of output formats and you do not have to check the size of the numbers ahead of time.

The G edit-descriptor must be associated with an input/output list item that has a real or double-precision data type, or it can be associated with either part of a complex list item. The forms of the G edit-descriptor are:

G*w.d*

G*w.d***E***e*

w
: is the character width of the field.

d
: is the length of the number's fractional part unless a scale-factor greater than 1 is in effect.

e
: specifies the field width of an exponent and has no effect on input.

On input, the G edit-descriptor is the same as the F edit-descriptor.

On output, the format of the converted number is dependent on its magnitude. The following table describes the action of the G edit-descriptor, in which *N* is the number to be converted, *b* indicates a blank, $n = 4$ in G*w.d* editing, and $n = e + 2$ in G*w.d*E*e* editing.

Magnitude of N	Equivalent Conversion
N < 0.1 or N > 10**d	Gw.d is the same as kPEw.d Gw.dEe is the same as kPEw.dEe
0.1 ≤ N < 1	F(w–n).d, n(b)
1 ≤ N < 10	F(w–n).(d–1), n(b)
.	.
.	.
.	.
10**(d–2) ≤ N < 10**(d–1)	F(w–n).1, n(b)
10**(d–1) ≤ N < 10**d	F(w–n).0, n(b)

Figure 11-1. G Edit-Descriptor

The G edit-descriptor can be used to transmit integer or logical data according to the type specification of the corresponding variable in the input/output list.

Output examples:

Note: A ƀ indicates a blank.

```
Value      Format   Output         Comments

1234.56    G12.5    ƀƀ1234.6ƀƀƀƀ     (1)
12345.6    G12.5    ƀƀ12346.ƀƀƀƀ     (2)
123456.    G12.5    ƀƀ.12346E+06     (3)

(1) This is in F8.1 format.
(2) This is in F8.0 format.
(3) This is in E12.5 format.
```

L — Logical Editing

The L edit-descriptor must be associated with an input/output list item that has a logical data type. The form of the L edit-descriptor is:

L*w*

w
is the character width of the field.

On input, the field consists of optional spaces followed by an optional decimal point followed by a T for .TRUE. or an F for .FALSE.. Any characters following the T or F are accepted on input but are not acted upon; therefore, the strings .TRUE. and .FALSE. are valid inputs.

Input examples:

```
Input      Format      Value        Comments

T            L4        .TRUE.
.F           L4        .FALSE.        (1)
  T          L6        .TRUE.         (2)
.FALSE.      L7        .FALSE.        (3)
.XX          L3        * error *      (4)

(1) The decimal point is optional.
(2) The leading blanks are optional.
(3) The "ALSE." is ignored.
(4) Other characters are not allowed.
```

On output, the T or F is preceded by w - 1 spaces.

Output examples:

Note: A ƀ indicates a blank.

```
Value        Format        Output

.TRUE.         L4          ƀƀƀT
.FALSE.        L1          F
.TRUE.         L6          ƀƀƀƀƀT
```

A — Character Editing ✦

The A edit-descriptor can be associated with an input/output list item that has any data type. The forms of the A edit-descriptor are:

A
A*w*

w
is the field width.

The straight A format acquires an implied field width from the number of characters in its associated input/output list item.

On input, if *w* exceeds the number of characters in the input/output list item, the input characters are left-justified and trailing blanks are added. If *w* is less that the number of characters in the input/output list item, the rightmost characters are truncated.

Input examples:

```
Input      Format      Value       Comments

ABCD         A         'ABCD'
ABCD         A4        'ABCD'
ABCDEF       A2        'AB'          (1)
ABCD         A6        'ABCD  '      (2)

(1) The rightmost characters are truncated.
(2) Trailing blanks are added.
```

On output, if *w* exceeds the number of characters produced by the input/output list item, leading blanks are provided; otherwise, the leftmost *w* characters of the input/output list item are output.

Output examples:

Note: A ƀ indicates a blank.

```
Value       Format      Output      Comments

'ABCDEF'      A         ABCDEF
'ABCDEF'      A6        ABCDEF
'ABCDEF'      A3        ABC           (1)
'ABCD'        A6        ƀƀABCD        (2)

(1) The leftmost characters are output.
(2) Two leading blanks are added.
```

Z — Hexadecimal Editing ✧ ✧

The Z edit-descriptor is used for hexadecimal representations of internal numeric, logical, and character data items. The form of the Z edit-descriptor is:

Z*w*

w
is the field width.

On input of numeric and logical input/output list items, hex digits are accepted with the most significant digits appearing first. Two hex digits are placed in each byte of the input/output list items. Blanks are treated as zeros or nulls depending whether the BZ or BN edit-descriptor is in effect. A warning message is issued if overflow occurs.

On input of character input/output list items, initial input data is ignored if the specified field width exceeds the default width. Hex digits are placed into the character input/output list item in address order, with blanks treated as zeros. Character input/output list items are filled with zero bytes through their highest byte if the input field width is less than the default field width.

Input examples:

```
            External      Internal
Format       Field        Hex Value

 Z3          E02           E02
 Z5          FF215A        FF215
 Z5          12C           0012C
 Z4          -FFF          * error *
 Z4          12.A          * error *
```

On output, if a specified field width exceeds the default width, the output field is filled with initial padding blanks (or zeros if the internal represen-

tation is zero). If a specified field width is less than the default width, the leftmost digits are truncated and the rest of the number is printed. A warning message is issued when truncation occurs.

Output examples:

```
            Internal
            (Decimal)       External
 Format      Value          Representation

  Z4          28643           6FE3
  Z8         -28643           FFF901D
  Z2          16              10
  Z4         -1               FFFF
```

This is a sample program using hexadecimal editing.

```
      PROGRAM HEXIO
      INTEGER I,J,K
      CHARACTER*5 STRING
      READ(*,100) I
      READ(*,100) J
      READ(*,100) K
      READ(*,100) STRING
100   FORMAT(Z)
      WRITE(*,300) I,J,K,STRING
300   FORMAT(3I8,A8)
      STOP
      END
```

Input data:

```
FFFFFFFF
000000A6
00006FE3
48454C4C4F
```

The output is:

```
-1      166      28643      HELLO
```

Formatting Extreme Values ✧

The "extreme values" used in the RT PC VS FORTRAN floating-point system include +infinity, -infinity, and Not-a-Number (NaN). Infinity is the result of floating-point overflow. NaN is produced by certain invalid operations, such as division by 0. For a further description of extreme values, see the *RT PC VS FORTRAN User's Guide*.

When extreme values are printed in D, E, F, or G format, the maximum positive or negative number is substituted for +infinity, -infinity, and NaN.

On the output of non-extreme values, the entire field is filled with asterisks (*) if the field width as specified by the associated format specifier in the FORMAT statement is inadequate to display the value.

Example:

```
C      (+infinity)
  1    A=1.E99
C      (-infinity)
  2    B=-1.E99
C     (Not-a-Number)
  3    C=0./0
       WRITE(*,5) A,B,C
  5    FORMAT(E15.10, E10.1, D15.10)
       END
```

Statements 1, 2, and 3 produce compile-time warnings for +infinity, -infinity, and NaN, respectively. The maximum positive number is substituted for A and C; the maximum negative number is substituted for B.

Nonrepeatable Edit-Descriptors ✦ ✧

Nonrepeatable edit-descriptors are format list items that are not associated with any input/output list items.

'xxx' — Apostrophe Editing

An apostrophe edit-descriptor has the form of a character constant and causes the characters (including the spaces) contained within the apostrophes to be written. An apostrophe that is part of the character string is indicated by two consecutive apostrophes (''). Apostrophe edit-descriptors cannot be used for input.

Examples:

```
      IVAL = 15
      WRITE(*,50) IVAL
  50  FORMAT('The value is -- ',I2)
```

The output is:

```
The value is -- 15
```

```
      READ(*,100) AMOUNT
C     An error occurs in the next statement.
 100  FORMAT('Input the amount:  ',F6.2)
```

An error occurs because apostrophe editing cannot be used for input.

H — Hollerith Editing

The form of the H edit-descriptor is:

n
is the number of characters after the H that are to be output. Spaces are included in the character count.

Hollerith editing cannot be used for input.

Examples:

```
      IVAL = 15
      WRITE(*,50) IVAL
  50  FORMAT(16HThe value is -- ,I2)
```

The output is:

```
The value is -- 15
```

```
      READ(*,100) AMOUNT
C     An error occurs in the next statement.
 100  FORMAT(18HInput the amount: ,F6.2)
```

An error occurs because Hollerith editing cannot be used for input.

X (Tab) and T (Skip) — Positional Editing

The X and T edit-descriptors position the format controller within a record and do not transmit any characters to or from a record.

When a formatted record is read on input, it is treated as if it has an infinite length, with as many trailing spaces as needed being used to to satisfy input requests. The X and T edit-descriptors determine the position of the next character to be read from the record. These edit-descriptors can therefore be used to skip portions of the input record or to read the same positions in the record more than once.

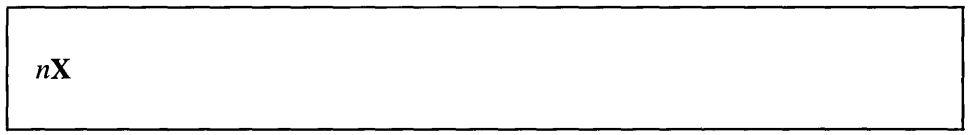
On output, the input/output system initially appears to create a record of infinite length that is filled with spaces. As formatted output transmits characters to the record, the final length of the record is determined by the rightmost position to which a character is transmitted. Changing the position with the X or T edit-descriptor does not directly affect the length of the record but does affect the position to which the next character is transmitted in the output record.

Using the X or T edit-descriptors, some positions in the record may not have any characters transmitted to them (they are skipped), which means that those character positions retain their original blank values. Characters

need to be transmitted after the skipped positions in order for those character positions to eventually be included in the output record.

It is possible to position to and overwrite a formatted output record position by using the X and T edit-descriptors. In this case, only the last value written to a given character position becomes part of the final formatted record.

The form of the X edit-descriptor is:



n
is the number of spaces the record position is advanced.

The T edit-descriptor positions the record to a specified column. One form of the T edit-descriptor is:

c
is an absolute column position.

Another form of the T edit-descriptor is:

c
is the number of characters to the left (backward) that the column position is moved relative to the current position.

A third form of the T edit-descriptor is:

TR*c*

c
is the number of characters to the right (forward) that the column position is moved relative to the current position.

On input, the T edit-descriptors are used for skipping or re-reading portions of the input record. If the TL*c* edit-descriptor moves the character position to where input fields were previously transmitted, those items can be re-transmitted.

On output, if a character is transmitted to a position where another character has already been transmitted, the earlier transmission is replaced.

Examples:

```
      WRITE(*,50)
 50   FORMAT('Column 1',5X,'Column 14',
     +       TR2,'Column 25')
```

The output is:

```
  Column 1      Column 14  Column 25
```

```
      WRITE(*,100)
100   FORMAT('aaaaa',TL2,'bbbbb',5X,'ccccc',
     +       T10,'ddddd')
```

The output is:

```
aaabbbbb dddddcccc
```

```
      READ(5,150) K,L
 150  FORMAT(I4,T30,I4)
```

Input data:

```
1000                          2000
```

K is assigned the value 1000 and L is assigned the value 2000.

```
      INTEGER ARR(5)
      REAL X
      .
      .
      READ(5,200) X,ARR
 200  FORMAT(F6.2,5X,I4,TL4,I4,TL4,I4,
     +       TL4,I4,TL4,I4)
```

Input data:

```
314.52     9876
```

X is assigned the value 314.52 and each of the five elements in the array ARR is assigned the value 9876.

/ — Starting a New Record

A slash (/) indicates the end of transfer on the current record.

On input, the file is positioned to the beginning of the next record.

On output, an end-of-record is written and the file is positioned to write at the start of the next record.

Examples:

```
      INTEGER X,Y,Z
      DATA X,Y,Z /12,13,14/
      .
      .
      WRITE(*,100) X Y,Z
 100  FORMAT(I4 / I4 / I4)
```

The output is:

```
12
13
14
```

```
      REAL X(3)
      .
      .
      READ(5,500) X
 500  FORMAT(F6.2 / 2F6.2)
```

Input data:

```
13.4
16.5   12.4
```

X(1), X(2), and X(3) are assigned the values 13.4, 16.5, and 12.4, respectively.

: — Conditional Termination

A colon (:) appearing in a format list ends the processing of the format list if there are no more items in the input/output list. If there are more items in the input/output list when the colon is encountered, the colon is ignored. More than one colon can appear in a format list.

The colon edit-descriptor is useful in preventing the transfer of extraneous textual data that might otherwise be printed after all appropriate numeric items have been transferred. It is also useful in preventing further slash editing on input.

Example:

```
      WRITE(*,10) (A(I),I=1,N)
10    FORMAT(3(:'Array Value' F10.5)/)
```

This code causes N values of the array A to be printed to the console. Because of the colon, the text is not printed if N is 0.

P — Scale-Factor Editing

The P edit-descriptor sets the scale-factor for subsequent E, F, and G edit-descriptors until another P edit-descriptor is encountered. The form of the P edit-descriptor is:

*k***P**

k
is an optionally signed integer constant.

At the start of each input/output statement, the scale-factor is 0.

On input with E, F, or G editing, providing that an exponent does not appear in the external field, the externally represented number is multiplied by 10**-*k* before being assigned to the corresponding input/output list item. The scale-factor has no effect if there is an exponent in the input field.

Input examples:

```
Input        Format      Value       Comments

98.765       3PF8.6      0.098765      (1)
98.765       -3PF8.6     98765.        (2)
.98765E+2    3PF10.5     .98765E+2     (3)

(1) Value=Input * 10**-3
(2) Value=Input * 10**3
(3) The scale-factor has no effect.
```

On output with E editing, the real part of the input/output list element is multiplied by 10**k and the exponent is reduced by k, thereby altering the column position of the decimal point but not the actual output value.

On output with F and G editing, the input/output list element is multiplied by 10**k before transfer to the record.

Output examples:

Note: A ƀ indicates a blank.

```
Value     Format     Output      Comments

12.34     2PF7.2     1234.00       (1)
12.34     -2PF6.4    0.1234        (2)
12.34     2PE10.3    ƀ12.34E+00    (3)

(1) Output=Value * 10**2
(2) Output=Value * 10**-2
(3) Real Part = Real Part * 10**2
    Exponent = Exponent - 2
```

BN and BZ — Blank Interpretation

BN and BZ edit-descriptors specify how blanks (spaces) are to be interpreted in numeric input fields. The initial setting for blank interpretation in a file is set with the BLANK= parameter in its OPEN statement.

If BZ editing is in effect, leading blanks are ignored and embedded blanks are treated as zeros. This edit-descriptor stays in effect until a BN edit-descriptor is encountered in the format list.

If BN editing is in effect, blanks in input fields are ignored. When blanks are ignored, all the non-blank characters in the input field are treated as if they are right-justified, with the number of leading blanks equal to the number of ignored blanks.

Examples:

```
C     The READ statement treats the characters
C     between the vertical bars as the value 123.
C     <Enter> indicates pressing the ENTER key.
C
      READ (*,100) I
 100  FORMAT(BN,I6)

      |123     <Enter>|
      |123   456<Enter>|
      |123<Enter>|
      |  123<Enter>|
```

Using the BN edit-descriptor in conjunction with the infinite blank padding at the end of formatted records makes interactive input very convenient.

```
      INTEGER AMOUNT
      .
      .
      READ(*,100) AMOUNT
 100  FORMAT(BZ,I3)
```

This code prompts the user for an amount that cannot be greater than 999 because the I3 edit-descriptor is used. Since BZ editing is in effect, the user must type in leading blanks in the input field if the number less than 100 because any trailing blanks are treated as zeros, making the input incorrect. If the BN edit-descriptor is used instead, as illustrated in the next example, the procedure is made less cumbersome.

```
      INTEGER AMOUNT
      .
      .
      READ(*,100) AMOUNT
 100  FORMAT(BN,I3)
```

In this code, all trailing blanks are ignored. The input field is right-justified and the trailing blanks are moved to the beginning of the field. For amounts less than 100, initial blanks do not have to be inserted.

S, SS and SP — Sign Control

An output field generated by I, D, E, F, or G editing includes an optional sign immediately preceding the digits of the value. The sign always appears if the number is negative; however, FORTRAN omits it from positive numbers.

The S, SS, and SP edit-descriptors can be used to control the printing of the optional plus signs. Any edit-descriptor chosen remains in effect until another one is encountered in the format list.

An SP edit-descriptor specifies that the optional plus signs be printed. An SS edit-descriptor specifies that they not be printed. An S edit-descriptor restores the option of omitting plus signs to FORTRAN.

On input, these edit-descriptors have no effect and are ignored.

R1 Mode Specifics

This section describes the instances in which R1 mode differs from IBM mode (the default mode).

Overview of FORMAT Statements

- A format cannot be specified by an array name.

Numeric Editing

- In a formatted read, a comma can be used to separate values in an input record. Commas override the field lengths given in the FORMAT statement. For example, the format `(I10, F20.10, I4)` reads the record `-345, .05E-3, 12` correctly.

I — Integer Editing

- The I*w.m* form of the I edit-descriptor is not allowed.

G — Real Editing

- The G edit-descriptor cannot be used to transmit integer or logical data.

A — Character Editing

- The A edit-descriptor can only be associated with an input/output list item that has a character data type.

Z — Hexadecimal Editing

- The Z edit-descriptor is not allowed.

Formatting Extreme Values

- An error message is issued upon output of +infinity, -infinity, and NaN.

Nonrepeatable Edit-Descriptors

- The dollar sign ($) edit-descriptor is also allowed, and is used to inhibit an end-of-record.

  Usually when the format controller ends a format list, data transmission to the current record ceases and the file is positioned so that a new WRITE starts a new record. But if the format controller encounters a dollar sign while scanning the format list, this automatic end-of-record action is inhibited. This means that subsequent input/output statements can continue reading from or writing to the same record.

  A common use for dollar sign editing is to prompt to the console and read a response from the same line.

  Example:

```
WRITE(*,'($A)') 'Enter your weight -> '
READ(*, '(BN, I6)') LIGHT
```

  The dollar sign edit-descriptor does not inhibit the automatic end-of-record generated when reading from the * unit. Input made to the console must be terminated by a carriage return, which permits the proper functioning of the backspace and line-delete keys.

VX Mode Specifics

This section describes the instances in which VX mode differs from IBM mode (the default mode).

Repeatable Edit-Descriptors

- The O edit-descriptor is also allowed, and is used for octal editing.

  The O edit-descriptor transfers octal (base 8) values, and can be used with any data type. The forms of the O edit-descriptor are:

O*w*
O*w.m*

w
: is the character width of the field.

m
: specifies a minimum field width for the integer value. The *m* must be greater than *w*.

On input, the O edit-descriptor transfers *w* characters from the external field and assigns them as an octal value to the corresponding input/output list element. The external field can contain only the numbers 0–7; it cannot contain a sign, a decimal point, or an exponent field. An all-blank field is treated as a value of 0. If the value of the external field exceeds the range of the corresponding list element, an error occurs. On input, the *m* specification has no effect.

Input examples:

Format	External Field	Internal Octal Value
O5	32767	32767
O4	16234	1623
O3	97Δ	* error *

On output, the O edit-descriptor transfers the octal value of the corresponding input/output list element, right-justified, to an external field *w* characters long. No signs are output; a negative value is transmitted in internal form. If the value does not fill the field, leading spaces are inserted; if the value exceeds the field width, the entire field is filled with asterisks.

If *m* is present, the external field consists of at least *m* digits, and is zero-filled on the left if necessary. Note that if *m* is 0 and the external representation is 0, then the external field is blank-filled.

Output examples:

Format	Internal (Decimal) Value	External Representation
O6	32767	Δ77777
O6	-32767	100001
O2	14261	**
O4	27	ΔΔ33
O5	10.5	41050
O4.2	7	ΔΔ07
O4.4	7	0007

Numeric Editing

- In a formatted read, a comma can be used to separate values in an input record. Commas override the field lengths given in the FORMAT statement. For example, the format `(I10, F20.10, I4)` reads the record `-345, .05E-3, 12` correctly.

G — Real Editing

- The G edit-descriptor cannot be used to transmit integer or logical data.

Z — Hexadecimal Editing

- The Z edit-descriptor can also have the form:

Z*w.m*

m
specifies a minimum field width.

If *m* is present, the external field consists of at least *m* digits, and is zero-filled on the left if necessary. Note that if *m* is 0 and the external representation is 0, then the external field is blank-filled.

Output examples:

Note: A ␣ indicates a blank.

Format	Internal (Decimal) Value	External Representation
Z3.3	1500	5DC
Z6.4	1500	␣␣5DC

Nonrepeatable Edit-Descriptors

- The dollar sign ($) edit-descriptor is also allowed, and is described on page 11-32.

- The Q edit-descriptor is also allowed, and is used for character count editing.

  The Q edit-descriptor determines how many characters are remaining in the input field and must be associated with an input/output list item that has an integer or logical data type.

  On input, the number of characters in the input record remaining to be transferred is assigned to the associated input/output list item. This can be useful if you wish to read a string of characters into an array and the length of the string is not known.

  Example:

```
      INTEGER COUNT
      CHARACTER*1 NAME(80)
      READ (*,10) I,J,COUNT,(NAME(K),K=1, COUNT)
10    FORMAT(2I5,Q,80A1)
```

  On output, the Q edit-descriptor is ignored and its associated input/output list item is skipped.

Appendix A. Intrinsic Functions

In the following figures, a parenthesized number following an intrinsic function refers to one of the notes that follow this chart. The notes contain additional information that you need to know to properly use the intrinsic functions.

"Dbl Cmplx" indicates COMPLEX*16 in IBM mode and indicates COMPLEX*16 or DOUBLE COMPLEX in R1 and VX modes.

IBM Mode Intrinsic Functions

Intrinsic Function	Number of Arguments	Generic Name	Specific Name	Data Type of Argument	Data Type of Function
Conversion to Integer (1)	1	INT	INT	Real	Integer
			IFIX	Real	Integer
			IDINT	Double	Integer
Conversion to Real (2)	1	REAL	REAL	Integer	Real
			SNGL	Double	Real
			DREAL	Complex	Double

Figure A-1 (Part 1 of 7). IBM Mode Intrinsic Functions

Intrinsic Function	Number of Arguments	Generic Name	Specific Name	Data Type of Argument	Data Type of Function
Conversion to Double-Precision (2)	1	DBLE	DFLOAT	Integer	Double
				Real	Double
				Double	Double
				Complex	Double
Conversion to Complex (3)	1 or 2	CMPLX		Integer	Complex
				Real	Complex
				Double	Complex
				Complex	Complex
			DCMPLX	Double	Dbl Cmplx
Conversion to Integer (4)	1		ICHAR	Character	Integer
Conversion to Character (4)	1		CHAR	Integer	Character
Truncation (5)	1	AINT	AINT	Real	Real
			DINT	Double	Double
Nearest Whole (6)	1	ANINT	ANINT	Real	Integer
			DNINT	Double	Double
Nearest Integer (7)	1	NINT	NINT	Real	Integer
			IDNINT	Double	Integer
Absolute Value (8)	1	ABS	IABS	Integer	Integer
			ABS	Real	Real
			DABS	Double	Double
			CABS	Complex	Real
			CDABS	Dbl Cmplx	Double

Figure A-1 (Part 2 of 7). IBM Mode Intrinsic Functions

Intrinsic Function	Number of Arguments	Generic Name	Specific Name	Data Type of Argument	Function
Remaindering (9)	2	MOD	MOD AMOD DMOD	Integer Real Double	Integer Real Double
Transfer of sign (10)	2	SIGN	ISIGN SIGN DSIGN	Integer Real Double	Integer Real Double
Positive Difference (11)	2	DIM	IDIM DIM DDIM	Integer Real Double	Integer Real Double
Double-Precision Product	2		DPROD	Real	Double
Choosing Largest Value	2 or more	MAX	MAX0 AMAX1 DMAX1 AMAX0 MAX1	Integer Real Double Integer Real	Integer Real Double Real Integer
Choosing Smallest Value	2 or more	MIN	MIN0 AMIN1 DMIN1 AMIN0 MIN1	Integer Real Double Integer Real	Integer Real Double Real Integer
Length (12)	1		LEN	Character	Integer
Index of Substring (13)	2		INDEX	Character	Integer

Figure A-1 (Part 3 of 7). IBM Mode Intrinsic Functions

Intrinsic Function	Number of Arguments	Generic Name	Specific Name	Data Type of Argument	Data Type of Function
Imaginary Part of Complex Argument	1		AIMAG DIMAG	Complex Dbl Cmplx	Real Double
Complex Conjugate (14)	1		CONJG DCONJG	Complex Dbl Cmplx	Complex Dbl Cmplx
Square Root (15)	1	SQRT	SQRT DSQRT CSQRT CDSQRT	Real Double Complex Dbl Cmplx	Real Double Complex Dbl Cmplx
Exponential	1	EXP	EXP DEXP CEXP CDEXP	Real Double Complex Dbl Cmplx	Real Double Complex Dbl Cmplx
Natural Logarithm (16)	1	LOG	ALOG DLOG CLOG CDLOG	Real Double Complex Dbl Cmplx	Real Double Complex Dbl Cmplx
Common Logarithm (16)	1	LOG10	ALOG10 DLOG10	Real Double	Real Double
Sine (17)	1	SIN	SIN DSIN CSIN CDSIN	Real Double Complex Dbl Cmplx	Real Double Complex Dbl Cmplx

Figure A-1 (Part 4 of 7). IBM Mode Intrinsic Functions

Intrinsic Function	Number of Arguments	Generic Name	Specific Name	Data Type of Argument	Data Type of Function
Cosine (17)	1	COS	COS DCOS CCOS CDCOS	Real Double Complex Dbl Cmplx	Real Double Complex Dbl Cmplx
Tangent (17)	1	TAN	TAN DTAN	Real Double	Real Double
Arcsine (18)	1	ASIN	ASIN DASIN	Real Double	Real Double
Arccosine (19)	1	ACOS	ACOS DACOS	Real Double	Real Double
Arctangent (20)	1	ATAN	ATAN DATAN	Real Double	Real Double
	2	ATAN2	ATAN2 DATAN2	Real Double	Real Double
Hyperbolic Sine	1	SINH	SINH DSINH	Real Double	Real Double
Hyperbolic Cosine	1	COSH	COSH DCOSH	Real Double	Real Double
Hyperbolic Tangent	1	TANH	TANH DTANH	Real Double	Real Double
Lexically Greater Than or Equal To (21)	2		LGE	Character	Logical
Lexically Greater Than (21)	2		LGT	Character	Logical

Figure A-1 (Part 5 of 7). IBM Mode Intrinsic Functions

Intrinsic Function	Number of Arguments	Generic Name	Specific Name	Data Type of Argument	Function
Lexically Less Than or Equal To (21)	2		LLE	Character	Logical
Lexically Less Than (21)	2		LLT	Character	Logical
Cotangent	1	COTAN	COTAN DCOTAN	Real Double	Real Double
Error Function	1 1 1 1		ERF ERFC DERF DERFC	Real Real Double Double	Real Real Double Double
Gamma and Log-gamma	1 1 1 1		GAMMA ALGAMA DGAMMA DLGAMA	Real Real Double Double	Real Real Double Double
Bitwise Logical OR	2		IOR	Integer	Integer
Bitwise Logical AND	2		IAND	Integer	Integer
Bitwise Logical XOR (exclusive OR)	2		IEOR	Integer	Integer
Bitwise Logical NOT (ones complement)	1		NOT	Integer	Integer

Figure A-1 (Part 6 of 7). IBM Mode Intrinsic Functions

Intrinsic Function	Number of Arguments	Generic Name	Specific Name	Data Type of Argument	Function
Bitwise Logical Shift	2		ISHFT	Integer	Integer
Clear a Bit (set to 0)	2		IBCLR	Integer	Integer
Set a Bit (set to 1)	2		IBSET	Integer	Integer
Test a Bit (return .TRUE. if bit is 1)	2		BTEST	Integer	Logical

Figure A-1 (Part 7 of 7). IBM Mode Intrinsic Functions

R1 Mode Intrinsic Functions

Intrinsic Function	Number of Arguments	Generic Name	Specific Name	Data Type of Argument	Function
Conversion to Integer (1)	1	INT	INT	Real	Integer
			IFIX	Real	Integer
			IDINT	Double	Integer
Conversion to Real (2)	1	REAL	REAL	Integer	Real
			FLOAT	Integer	Real
			SNGL	Double	Real
Conversion to Double-Precision (2)	1	DBLE		Integer	Double
				Real	Double
				Double	Double
				Complex	Double
Conversion to Complex (3)	1 or 2	CMPLX		Integer	Complex
				Real	Complex
				Double	Complex
				Complex	Complex
			DCMPLX	Double	Dbl Cmplx
Conversion to Integer (4)	1		ICHAR	Character	Integer
Conversion to Character (4)	1		CHAR	Integer	Character
Truncation (5)	1	AINT	AINT	Real	Real
			DINT	Double	Double
Nearest Whole (6)	1	ANINT	ANINT	Real	Integer
			DNINT	Double	Double

Figure A-2 (Part 1 of 5). R1 Mode Intrinsic Functions

Intrinsic Function	Number of Arguments	Generic Name	Specific Name	Data Type of Argument	Data Type of Function
Nearest Integer (7)	1	NINT	NINT	Real	Integer
			IDNINT	Double	Integer
Absolute Value (8)	1	ABS	IABS	Integer	Integer
			ABS	Real	Real
			DABS	Double	Double
			CABS	Complex	Real
			ZABS	Dbl Cmplx	Double
Remaindering (9)	2	MOD	MOD	Integer	Integer
			AMOD	Real	Real
			DMOD	Double	Double
Transfer of sign (10)	2	SIGN	ISIGN	Integer	Integer
			SIGN	Real	Real
			DSIGN	Double	Double
Choosing Largest Value	2 or more	MAX	MAX0	Integer	Integer
			AMAX1	Real	Real
			DMAX1	Double	Double
			AMAX0	Integer	Real
			MAX1	Real	Integer
Choosing Smallest Value	2 or more	MIN	MIN0	Integer	Integer
			AMIN1	Real	Real
			DMIN1	Double	Double
			AMIN0	Integer	Real
			MIN1	Real	Integer
Length (12)	1		LEN	Character	Integer

Figure A-2 (Part 2 of 5). R1 Mode Intrinsic Functions

Intrinsic Function	Number of Arguments	Generic Name	Specific Name	Data Type of Argument	Data Type of Function
Index of Substring (13)	2		INDEX	Character	Integer
Imaginary Part of Complex Argument	1		AIMAG DIMAG	Complex Dbl Cmplx	Real Double
Complex Conjugate (14)	1		CONJG DCONJG	Complex Dbl Cmplx	Complex Dbl Cmplx
Square Root (15)	1	SQRT	SQRT DSQRT CSQRT	Real Double Complex	Real Double Complex
Exponential	1	EXP	EXP DEXP CEXP	Real Double Complex	Real Double Complex
Natural Logarithm (16)	1	LOG	ALOG DLOG CLOG	Real Double Complex	Real Double Complex
Common Logarithm (16)	1	LOG10	ALOG10 DLOG10	Real Double	Real Double
Sine (17)	1	SIN	SIN DSIN CSIN	Real Double Complex	Real Double Complex

Figure A-2 (Part 3 of 5). R1 Mode Intrinsic Functions

Intrinsic Function	Number of Arguments	Generic Name	Specific Name	Data Type of Argument	Data Type of Function
Cosine (17)	1	COS	COS DCOS CCOS	Real Double Complex	Real Double Complex
Tangent (17)	1	TAN	TAN DTAN	Real Double	Real Double
Arcsine (18)	1	ASIN	ASIN DASIN	Real Double	Real Double
Arccosine (19)	1	ACOS	ACOS DACOS	Real Double	Real Double
Arctangent (20)	1	ATAN	ATAN DATAN	Real Double	Real Double
	2	ATAN2	ATAN2 DATAN2	Real Double	Real Double
Hyperbolic Sine	1	SINH	SINH DSINH	Real Double	Real Double
Hyperbolic Cosine	1	COSH	COSH DCOSH	Real Double	Real Double
Hyperbolic Tangent	1	TANH	TANH DTANH	Real Double	Real Double
Bitwise Logical OR	2		IOR	Integer	Integer

Figure A-2 (Part 4 of 5). R1 Mode Intrinsic Functions

Intrinsic Function	Number of Arguments	Generic Name	Specific Name	Data Type of Argument	Function
Bitwise Logical AND	2		IAND	Integer	Integer
Bitwise Logical XOR (exclusive OR)	2		IEOR	Integer	Integer
Bitwise Logical NOT (ones complement)	1		NOT	Integer	Integer
Bitwise Logical Shift	2		ISHFT	Integer	Integer
Bitwise Circular Shift	2		ISHFTC	Integer	Integer
Clear a Bit (set to 0)	2		IBCLR	Integer	Integer
Set a Bit (set to 1)	2		IBSET	Integer	Integer
Test a Bit (return .TRUE. if bit is 1)	2		BTEST	Integer	Logical
Retrieve Bits	3		IBITS	Integer	Integer

Figure A-2 (Part 5 of 5). R1 Mode Intrinsic Functions

VX Mode Intrinsic Functions

Intrinsic Function	Number of Arguments	Generic Name	Specific Name	Data Type of Argument	Data Type of Function
Conversion to Integer	1	IFIX		Real	Integer
Conversion to Real (2)	1	REAL	REAL	Integer	Real
			REAL	Complex	Real
			REAL	Dbl Cmplx	Real
			SNGL	Double	Real
		FLOAT		Integer	Real
Conversion to Double-Precision (2)	1	DBLE		Integer	Double
				Real	Double
				Double	Double
				Complex	Double
				Dbl Cmplx	Double
		DFLOAT		Integer	Double
Conversion to Complex (3)	1 or 2	CMPLX		Integer	Complex
	1 or 2			Real	Complex
	1 or 2			Double	Complex
	1			Complex	Complex
	1			Dbl Cmplx	Complex

Figure A-3 (Part 1 of 9). VX Mode Intrinsic Functions

Intrinsic Function	Number of Arguments	Generic Name	Specific Name	Data Type of Argument	Data Type of Function
Conversion to Double-Complex	1 or 2	DCMPLX		Integer	Dbl Cmplx
	1 or 2			Real	Dbl Cmplx
	1 or 2			Double	Dbl Cmplx
	1			Complex	Dbl Cmplx
	1			Dbl Cmplx	Dbl Cmplx
Conversion to Integer (4)	1		ICHAR	Character	Integer
Conversion to Character (4)	1		CHAR	Logical	Character
				Integer	Character
Truncation (1, 5)	1	INT		Real	Integer
				Double	Integer
				Complex	Integer
				Dbl Cmplx	Integer
		IDINT		Double	Integer
		AINT	AINT	Real	Real
			DINT	Double	Double
Nearest Whole (6)	1	ANINT	ANINT	Real	Real
			DNINT	Double	Double
Nearest Integer (7)	1	NINT		Real	Integer
				Double	Integer
		IDNINT		Double	Integer

Figure A-3 (Part 2 of 9). VX Mode Intrinsic Functions

Intrinsic Function	Number of Arguments	Generic Name	Specific Name	Data Type of Argument	Data Type of Function
Absolute Value (8)	1	ABS	ABS	Real	Real
			DABS	Double	Double
			CABS	Complex	Real
			CDABS	Dbl Cmplx	Double
		IABS		Integer	Integer
Remaindering (9)	2	MOD	MOD	Integer	Integer
			AMOD	Real	Real
			DMOD	Double	Double
Transfer of Sign (10)	2	SIGN	SIGN	Real	Real
			DSIGN	Double	Double
		ISIGN		Integer	Integer
Positive Difference (11)	2	IDIM		Integer	Integer
		DIM	DIM	Real	Real
			DDIM	Double	Double
Double-Precision Product	2		DPROD	Real	Double

Figure A-3 (Part 3 of 9). VX Mode Intrinsic Functions

Intrinsic Function	Number of Arguments	Generic Name	Specific Name	Data Type of Argument	Data Type of Function
Choosing Largest Value	2 or more	MAX	AMAX1	Real	Real
			DMAX1	Double	Double
		MAX0		Integer	Integer
		MAX1		Real	Integer
		AMAX0		Integer	Real
Choosing Smallest Value	2 or more	MIN	AMIN1	Real	Real
			DMIN1	Double	Double
		MIN0		Integer	Integer
		MIN1		Real	Integer
		AMIN0		Integer	Real
Length (12)	1		LEN	Character	Integer
Index of Substring (13)	2		INDEX	Character	Integer
Real Part of Complex Argument	1		REAL	Complex	Real
			DREAL	Dbl Cmplx	Double

Figure A-3 (Part 4 of 9). VX Mode Intrinsic Functions

Intrinsic Function	Number of Arguments	Generic Name	Specific Name	Data Type of Argument	Data Type of Function
Imaginary Part of Complex Argument	1		AIMAG DIMAG	Complex Dbl Cmplx	Real Double
Complex Conjugate (14)	1	CONJG	CONJG DCONJG	Complex Dbl Cmplx	Complex Dbl Cmplx
Square Root (15)	1	SQRT	SQRT DSQRT CSQRT CDSQRT	Real Double Complex Dbl Cmplx	Real Double Complex Dbl Cmplx
Exponential	1	EXP	EXP DEXP CEXP CDEXP	Real Double Complex Dbl Cmplx	Real Double Complex Dbl Cmplx
Natural Logarithm (16)	1	LOG	ALOG DLOG CLOG CDLOG	Real Double Complex Dbl Cmplx	Real Double Complex Dbl Cmplx
Common Logarithm (16)	1	LOG10	ALOG10 DLOG10	Real Double	Real Double
Sine (17)	1	SIN	SIN DSIN CSIN CDSIN	Real Double Complex Dbl Cmplx	Real Double Complex Dbl Cmplx

Figure A-3 (Part 5 of 9). VX Mode Intrinsic Functions

Intrinsic Function	Number of Arguments	Generic Name	Specific Name	Data Type of Argument	Data Type of Function
Sine (degree)	1		SIND DSIND	Real Double	Real Double
Cosine (17)	1	COS	COS DCOS CCOS CDCOS	Real Double Complex Dbl Cmplx	Real Double Complex Dbl Cmplx
Cosine (degree)	1		COSD DCOSD	Real Double	Real Double
Tangent (17)	1	TAN	TAN DTAN	Real Double	Real Double
Tangent (degree)	1	TAND	TAND DTAND	Real Double	Real Double
Arcsine (18)	1	ASIN	ASIN DASIN	Real Double	Real Double
Arcsine (degree)	1	ASIND	ASIND DASIND	Real Double	Real Double
Arccosine (19)	1	ACOS	ACOS DACOS	Real Double	Real Double
Arccosine (degree)	1	ACOSD	ACOSD DACOSD	Real Double	Real Double

Figure A-3 (Part 6 of 9). VX Mode Intrinsic Functions

Intrinsic Function	Number of Arguments	Generic Name	Specific Name	Data Type of Argument	Data Type of Function
Arctangent (20)	1	ATAN	ATAN DATAN	Real Double	Real Double
	2	ATAN2	ATAN2 DATAN2	Real Double	Real Double
Arctangent (degree)	1	ATAND	ATAND DATAND	Real Double	Real Double
	2	ATAN2D	ATAN2D DATAN2D	Real Double	Real Double
Hyperbolic Sine	1	SINH	SINH DSINH	Real Double	Real Double
Hyperbolic Cosine	1	COSH	COSH DCOSH	Real Double	Real Double
Hyperbolic Tangent	1	TANH	TANH DTANH	Real Double	Real Double
Lexically Greater Than or Equal To (21)	2		LGE	Character	Logical
Lexically Greater Than (21)	2		LGT	Character	Logical

Figure A-3 (Part 7 of 9). VX Mode Intrinsic Functions

Intrinsic Function	Number of Arguments	Generic Name	Specific Name	Data Type of Argument	Function
Lexically Less Than or Equal To (21)	2		LLE	Character	Logical
Lexically Less Than (21)	2		LLT	Character	Logical
Zero-Extend Functions	1	ZEXT	IZEXT	LOGICAL*1	INTEGER*2
				LOGICAL*2	INTEGER*2
				INTEGER*2	INTEGER*2
			JZEXT	LOGICAL*1	INTEGER*4
				LOGICAL*2	INTEGER*4
				LOGICAL*4	INTEGER*4
				INTEGER*2	INTEGER*4
				INTEGER*4	INTEGER*4
Bitwise Logical OR	2	IOR	IIOR	INTEGER*2	INTEGER*2
			JIOR	INTEGER*4	INTEGER*4
Bitwise Logical AND	2	IAND	IIAND	INTEGER*2	INTEGER*2
			JIAND	INTEGER*4	INTEGER*4
Bitwise Logical XOR (exclusive OR)	2	IEOR	IIEOR	INTEGER*2	INTEGER*2
			JIEOR	INTEGER*4	INTEGER*4
Bitwise Logical NOT (ones complement)	1	NOT	INOT	INTEGER*2	INTEGER*2
			JNOT	INTEGER*4	INTEGER*4

Figure A-3 (Part 8 of 9). VX Mode Intrinsic Functions

Intrinsic Function	Number of Arguments	Generic Name	Specific Name	Data Type of Argument	Data Type of Function
Bitwise Logical Shift	2	ISHFT	IISHFT	INTEGER*2	INTEGER*2
			JISHFT	INTEGER*4	INTEGER*4
Bitwise Circular Shift	2	ISHFTC	IISHFTC	INTEGER*2	INTEGER*2
			JISHFTC	INTEGER*4	INTEGER*4
Clear a Bit (set to 0)	2	IBCLR	IIBCLR	INTEGER*2	INTEGER*2
			JIBCLR	INTEGER*4	INTEGER*4
Set a Bit (set to 1)	2	IBSET	IIBSET	INTEGER*2	INTEGER*2
			JIBSET	INTEGER*4	INTEGER*4
Test a Bit (return .TRUE. if bit is 1)	2	BTEST	BITEST	INTEGER*2	LOGICAL*4
			BJTEST	INTEGER*4	LOGICAL*2
Retrieve Bits	3	IBITS	IIBITS	INTEGER*2	INTEGER*2
			JIBITS	INTEGER*4	INTEGER*4

Figure A-3 (Part 9 of 9). VX Mode Intrinsic Functions

Notes

In general, if a generic name of an intrinsic function exists, it can be used in place of a specific name to permit greater flexibility. Except for the data type conversion functions, the data type of the argument to a generic intrinsic function determines the data type of the result.

For example, the generic intrinsic function LOG computes the natural logarithm of its argument, which can have a real, double-precision, or complex data type. The data type of the result is the same as the data type of its argument. The specific intrinsic functions ALOG, DLOG, and CLOG also compute natural logarithms. ALOG computes the logarithm of a real argument and returns a real result. Likewise, DLOG and CLOG accept double-precision and complex arguments and return double-precision and complex results, respectively.

All arguments in an intrinsic function reference must have the same data type.

The restrictions on the ranges of arguments and the ranges of results apply to the intrinsic functions when referenced in both their generic forms and specific forms.

The result of a complex function is the principal value of the function.

---

(1) The INT function truncates a real or double-precision argument toward 0. If the argument to INT is a complex number or variable, the function is applied to its real part. IFIX is the same as INT for real arguments.

(2) If the argument to a REAL or DBLE function is complex, the function is applied to its real part.

(3) A single argument to a CMPLX function can have an integer, real, double-precision, or complex data type. If a CMPLX function has two arguments, both arguments must have the same data type, which can be integer, real, or double-precision.

If *a* is a complex variable, CMPLX(*a*) is *a*. If *a* is an integer, real, or double-precision variable, CMPLX(*a*) is the complex value whose real part is REAL(*a*) and whose imaginary part is 0. CMPLX(*a,b*) is a complex value whose real part is REAL(*a*) and whose imaginary part is REAL(*b*).

(4) ICHAR converts a character argument to an integer argument based on the position of the character argument in the character collating sequence used by the processor. If the length of the argument to ICHAR is greater than one character, ICHAR returns the collating sequence index of the first character of the argument. Similarly, CHAR(*i*) returns the *i*'th character in the collating sequence. CHAR has a character data type and a length of 1.

RT PC VS FORTRAN uses the ASCII character set to represent characters; therefore, the complementary functions CHAR and ICHAR convert characters to their ASCII representations. For example, ICHAR('A') is 65.

(5) The truncation functions AINT and DINT are like the INT function except that a real or double-precision result is returned instead of an integer result.

(6) The nearest whole number function returns INT(*a* + .5) if *a* is greater than or equal to 0 or INT(*a* - .5) if *a* is less than 0. If the generic form of the function is used, the data type of the result depends on the data type of *a* and can be either real or double-precision.

(7) The nearest integer function is like the nearest whole number function except that it returns an integer result.

(8) When used with a complex argument, ABS returns the square root of the sum of the squares of the real and imaginary components of the complex value. The result in this case has a real data type.

(9) MOD(a,b) is defined as a-b*INT(a/b). The result for MOD, AMOD, and DMOD is undefined when the value of the second argument is 0.

(10) SIGN(a,b) is ABS(a) if b is greater than or equal to 0 and -ABS(a) if b is less than 0. If the value of a is 0, the result is 0.

(11) DIM(a,b) is a-b if a is greater than b; otherwise, it is 0.

(12) When applied to a character variable or array element, LEN returns the length that the variable or element had when it was declared in the CHARACTER type statement. LEN is useful in a subprogram where a character variable appears as a dummy argument and is declared as an assumed-size variable. The argument of LEN does not have to be defined at the time the function is referenced. The LEN of a character constant is the number of characters between the apostrophes.

(13) INDEX(a,b) returns an integer value indicating the starting position within the character string a of a substring identical to string b. If b occurs more than once in a, the starting position of the first occurrence is returned. If a does not contain b or if b is longer than a, the INDEX function returns a value of 0.

(14) A complex value is expressed as an ordered pair of reals (a,b). The a is the real part and b is the imaginary part. CONJG(a,b) returns ($a,-b$); in other words, CONJG returns a complex result whose imaginary part is the negative of the imaginary part of the argument.

(15) SQRT and DSQRT require an argument that is not less than 0. The result of CSQRT is the principal value with the real part greater than or equal to 0. When the real part of the result is 0, the imaginary part is greater than or equal to 0.

(16) ALOG, DLOG, ALOG10, or DLOG10 require an argument that is greater than 0. The value of the argument of CLOG must not be (0.,0.). The range of the imaginary part of the result of CLOG is $-\pi <$ imaginary part $\leq \pi$. The imaginary part of the result is π only when the real part of the argument is less than 0 and the imaginary part of the argument is 0.

(17) All angles are expressed in radians. The absolute value of the argument of SIN, DSIN, COS, DCOS, TAN, and DTAN can be greater than $2*\pi$.

(18) The absolute value of the argument of ASIN and DASIN must be less than or equal to 1. The range of the result in radians is $-\pi/2 \leq \text{result} \leq \pi/2$.

(19) The absolute value of the argument of ACOS and DACOS must be less than or equal to 1. The range of the result in radians is $0 \leq \text{result} \leq \pi$.

(20) The range of the result in radians for ATAN and DATAN is $-\pi/2 \leq \text{result} \leq \pi/2$. ATAN2(*a,b*) and DATAN2(*a,b*) return the arctangent of a/b. If the value of the first argument of ATAN2 or DATAN2 is positive, the result is positive. If the value of the first argument is 0, the result is 0 if the second argument is positive; the result is π if the second argument is negative. If the value of the first argument is negative, the result is negative. If the value of the second argument is 0, the absolute value of the result is $\pi/2$. The arguments cannot both have the value 0. The range in radians of the result for ATAN2 and DATAN2 is $-\pi < \text{result} \leq \pi$.

(21) The logical functions LGE(*a,b*), LGT(*a,b*), LLE(*a,b*) and LLT(*a,b*) return true or false values depending on the collating sequence of the characters in variables *a* and *b*. LGE(*a,b*) returns a true value if $a=b$ or if *a* follows *b* in the collating sequence; otherwise, a false value is returned. LGT(*a,b*) returns a true value if *a* follows *b* in the collating sequence; otherwise, a false value is returned. LLE(*a,b*) returns a true value if $a=b$ or if *a* comes before *b* in the collating sequence; otherwise, a false value is returned. LLT(*a,b*) returns a true value if *a* comes before *b* in the collating sequence; otherwise, a false value is returned.

If the operands for any of these functions do not have equal lengths, the shorter operand is extended to the right with blanks to the length of the longer operand. For example, LGT('Todd','Wayne') returns a false value.

Appendix B. Information for the FORTRAN 66 Programmer

Note: Information in this appendix applies only to FORTRAN programs compiled under IBM mode, which is the default mode.

The purpose of this appendix is to assist the FORTRAN 66 programmer in using RT PC VS FORTRAN, which is based upon FORTRAN 77 standards. This appendix includes a description of:

- FORTRAN 77 character data
- migrating FORTRAN 66 programs to RT PC VS FORTRAN.

Using FORTRAN 77 Character Data

The following sections provide a description of how character data is used in FORTRAN 77 programs. For the exact syntax of statements, as well as any restrictions, see the appropriate sections in Chapters 1 - 11 of this manual.

The ANSI standard for FORTRAN 77 does not include how character data is to be stored in the computer's memory. The standard does refer to a collating sequence (or order) of the characters but does not assign numerical values to them. In RT PC VS FORTRAN, ASCII codes are used to internally represent character data; therefore, programs written in this code that assume character data is represented by the ASCII codes may not be portable to other computers that use a different display mode such as EBCDIC.

Character Variables

In FORTRAN 66, there are no character variables and all characters must be stored in numeric variables, which can become quite cumbersome.

FORTRAN 77 features the character data type, which defines variables for character assignment, storage, and manipulation. Numeric data types are not allowed to do this. Character data objects are declared with CHARACTER type statements.

Examples:

```
C     A 20-byte variable is declared.
      CHARACTER*20 NAME

C     An 8-byte variable and a 15-byte
C     variable are declared.
      CHARACTER FNAME*8,LNAME*15

C     An array of 50 elements having
C     20 bytes each is declared.
      CHARACTER*20 NAMES(50)
```

Character Constants

In standard FORTRAN 66, character constants are represented only by Hollerith constants. In FORTRAN 77, a character constant can be represented by a Hollerith constant or by a string of characters enclosed in apostrophes ('xxx').

This second type of character constant has to be between 1 and 255 characters in length. Spaces within the apostrophes are part of the character constant and are included in the character count. An apostrophe that is part of the character constant is indicated by two consecutive apostrophes.

This is a comparison of hexadecimal and string character constants:

Hexadecimal	String
4HABCD	'ABCD'
15HEnter File Name	'Enter File Name'
10HWhat's Up?	'What''s Up?'
1H'	''''

Character constants can be assigned to character variables, passed as parameters in subroutine subprogram calls and function references, and outputted using the standard input/output routines.

Examples:

```
      CHARACTER*10 NAME,POLICY(15)

      NAME = 'James'
C       The string 'James     ' is assigned to NAME.

      NAME = 'Doctor Proctor'
C       The string 'Doctor Pro' is assigned to NAME.

      DO 10 I=1,15
         POLICY(I) = 'Life Insur'
  10  CONTINUE
C       All 15 elements of the array POLICY
C       are initialized to 'Life Insur'.
```

Character constants can be given symbolic names by using the PARAMETER statement.

Example:

```
      CHARACTER*22 TITLE
      PARAMETER (TITLE = 'Encyclopedia Americana')
```

This code allows the programmer to refer to the character constant "Encyclopedia Americana" by the name TITLE. TITLE is a symbolic name and

not a variable; therefore, it cannot be assigned to a different character constant.

Character Substrings

A character substring is a portion of a character string and is identified by a substring name that can be referenced and have values assigned to it. The form of a substring name is:

charvar ([*start*] : [*finish*])

charvar
is a character variable.

start
finish
are integer expressions within the bounds of the declared size of *charvar*. The default size of *start* is 1 and the default size of *finish* is the size of *charvar*.

Examples:

```
      CHARACTER*20 STR1,STR2,STR3(10)

      STR1(1:5) = 'Stuff'
C       'Stuff' is assigned to the
C       first five positions of STR1.

      STR1(6:10) = STR2(1:5)
C       The first five characters of STR2 are
C       assigned to positions 6-10 of STR1.

      STR1(:10) = STR2(11:)
C       The last 10 characters of STR2 are
C       assigned to the first 10 positions of STR1.

      STR1(:) = STR2(:)
C       All of STR2 is assigned to STR1.
```

```
      STR3(4)(1:10) = STR(1:10)
C       The first 10 characters of STR1 are
C       assigned to the first 10 positions of
C       the fourth element of array STR3.

      CHARACTER*21 STRING
      DATA STRING /'All Things Considered'/

      STRING(:)
C       STRING = 'All Things Considered'
      STRING(:10)
C       STRING = 'All Things'
      STRING(11:)
C       STRING = ' Considered'
      STRING(5:15)
C       STRING = 'Things Cons'
      STRING(10:30)
C       An error occurs because 30 is
C       greater than the size of STRING.
```

Initializing Character Variables

A DATA statement can be used to initially assign character constants to character variables.

Examples:

```
      CHARACTER*10 NAME(10)
      DATA NAME(1),NAME(2) /'Mr.','President'/
      DATA NAME /10*'unknown'/
```

Both DATA statements initialize elements of the character array NAME.

A DATA statement can also assign nonprintable characters such as line-feed and form-feed to character variables. The character variable must be 1 byte in length and assigned to the ASCII representation of the nonprintable character.

Example:

```
      CHARACTER*1 CR,FF,CC(10)
      DATA CR,FF,CC /13,$12,(I,I=0,9)/
```

This code initializes CR to a carriage-return, FF to a form-feed, and array CC to the first 10 ASCII characters.

The Concatenation Operator

FORTRAN's only character operator is // (concatenation). Concatenation joins two strings in the order specified, thus forming one string whose length is equal to the sum of the lengths of the two strings.

Examples:

```
      CHARACTER*9 FNAME,LNAME
      DATA FNAME,LNAME /'President','Roosevelt'/

      PRINT *,FNAME // LNAME
C      'PresidentRoosevelt' is printed.

      PRINT *,FNAME // ' ' // LNAME
C       'President Roosevelt' is printed.

      PRINT *,FNAME(1:4) //'. '// LNAME
C       'Pres. Roosevelt' is printed.
```

Character Intrinsic Functions

FORTRAN 77 provides intrinsic functions that either input or output character data. These include ICHAR, CHAR, INDEX, LEN, LGE, LGT, LLE, and LLT, which are listed in Appendix A, "Intrinsic Functions."

ICHAR

The ICHAR function converts a character argument to an integer argument based on the position of the character argument in the character collating sequence used by the processor. The form of the ICHAR function is:

ICHAR (*char*)

char
is a character argument. If *char* is longer than one character, the collating sequence index of the first character is returned and the remaining characters are ignored.

The ICHAR function in RT PC VS FORTRAN converts a character to its ASCII representation; for example, ICHAR('A') is 65.

CHAR

The CHAR function converts an integer argument to the character argument that has that integer as its collating sequence index. The form of the CHAR function is:

CHAR (*int*)

int
is an integer argument.

The CHAR function in RT PC VS FORTRAN converts an integer to a character based on the ASCII collating sequence; for example, CHAR(65) is A.

INDEX

The INDEX function requires two character expressions as arguments and returns an integer value indicating the starting position within the first character string of a substring identical to the second string. The form of the INDEX function is:

INDEX (*a* , *b*)

a
is the character string to be scanned for the substring *b*.

If *b* occurs more than once in *a*, the starting position of the first occurrence is returned. If *a* does not contain *b*, or if *b* is longer than *a*, the INDEX function returns a value of 0.

LEN

The LEN function requires a character expression as an argument and returns its length. The form of the LEN function is:

LEN (*char*)

char
is a character expression.

The LEN of a character variable or array element is the length it had when it was declared in the CHARACTER type statement. The LEN of a character constant is the number of characters between the apostrophes.

Examples:

```
      CHARACTER*10 STR
      STR = 'ABC'
      L = LEN(STR)
C       L = 10
      L = LEN('XYZ')
C       L = 3

C     In this code, CHARACTER*(*) causes STR to
C     take on the length of the argument passed.
C     On the first call to SUB, LEN(STR) evaluates to 80;
C     on the second call to SUB, LEN(STR) evaluates to 20.
C
      PROGRAM MAIN
      CHARACTER STR1*80, STR2*20
      READ(*,'(A)')STR1
      STR2 = 'Title'
      CALL SUB(STR1)
      CALL SUB(STR2)
      .
      .
      END

      SUBROUTINE SUB(STR)
      CHARACTER*(*) STR
      DO 10 I=1, LEN(STR)
      .
      .
  10  CONTINUE
      RETURN
      END
```

LGE, LGT, LLE, and LLT

The LGE, LGT, LLE, and LLT functions return true or false values depending on the collating sequences of two character arguments. The forms of these logical intrinsic functions are:

```
LGE ( a , b )

LGT ( a , b )

LLE ( a , b )

LLT ( a , b )
```

a
b
are character arguments.

LGE(*a,b*) returns a true value if *a=b* or if *a* follows *b* in the collating sequence; otherwise, a false value is returned. LGT(*a,b*) returns a true value if *a* follows *b* in the collating sequence; otherwise, a false value is returned. LLE(*a,b*) returns a true value if *a=b* or if *a* comes before *b* in the collating sequence; otherwise, a false value is returned. LLT(*a,b*) returns a true value if *a* comes before *b* in the collating sequence; otherwise, a false value is returned.

If the operands for any of these functions do not have equal lengths, the shorter operand is extended to the right with blanks to the length of the longer operand. For example, LGT('Todd','Wayne') returns a false value.

Sample Program Using Character Data

This program prompts the user for a list of up to 10 names. The names are then separated into two arrays — one for first names and one for last names. Each array is then sorted alphabetically. Finally, the first name in the first name array is concatenated with the first name in the last name array and printed, the second name in the first name array is concatenated with the second name in the last name array and printed, and so on.

```
      PROGRAM CHARS
      CHARACTER*20 NAMES(10), FNAMES(10), LNAMES(10)
      INTEGER COUNT
      DATA NAMES,FNAMES,LNAMES /30*' '/
      CALL GETNAM(NAMES,COUNT)
      CALL SPLIT(NAMES,FNAMES,LNAMES,COUNT)
      CALL SORT(FNAMES,COUNT)
      CALL SORT(LNAMES,COUNT)
      CALL PRINT(FNAMES,LNAMES,COUNT)
      STOP
      END

C      GETNAM prompts the user for the list of input names.
C      The input list is ended by entering a blank name.
C
      SUBROUTINE GETNAM(NAMES,COUNT)
      INTEGER I,COUNT
      CHARACTER*20 NAMES(10)
      WRITE(*,*)
      DO 10 I=1,10
         WRITE(*,'(A,I2,A,$)') 'Enter name #',I,' -- '
         READ(*,'(A)') NAMES(I)
         IF (NAMES(I)(1:1).EQ.' ') THEN
            COUNT = I-1
            RETURN
         ENDIF
   10 CONTINUE
      COUNT = 10
      RETURN
      END
```

```
C     SPLIT searches the array of names for blanks
C     and assumes that a blank separates a first
C     name from a last name.
C     It then places first names and last names in
C     the FNAMES and LNAMES arrays, respectively.
C
      SUBROUTINE SPLIT(NAMES,FNAMES,LNAMES,COUNT)
      INTEGER COUNT,PLACE,I
      CHARACTER*20 NAMES(COUNT), FNAMES(COUNT), LNAMES(COUNT)
      DO 10 I=1,COUNT
         PLACE = INDEX(NAMES(I),' ')
         IF (PLACE.EQ.0) PLACE = 20
         FNAMES(I) = NAMES(I)(:PLACE)
         IF (PLACE.NE.20) LNAMES(I) = NAMES(I)(PLACE+1:)
   10 CONTINUE
      RETURN
      END

C     SORT sorts the arrays of names
C     using a basic binary sort.
C
      SUBROUTINE SORT(NAMES,COUNT)
      INTEGER COUNT,I,J
      CHARACTER*20 NAMES(COUNT),TEMP
      DO 20 I=1, COUNT-1
         DO 10 J=I+1,COUNT
            IF (LGT(NAMES(I),NAMES(J))) THEN
               TEMP = NAMES(I)
               NAMES(I) = NAMES(J)
               NAMES(J) = TEMP
            ENDIF
   10    CONTINUE
   20 CONTINUE
      RETURN
      END
```

```
C     PRINT concatenates the first and last
C     names and removes any excess blanks.
C
      SUBROUTINE PRINT(FNAMES,LNAMES,COUNT)
      INTEGER COUNT,I,PLACE1,PLACE2
      CHARACTER*20 FNAMES(COUNT), LNAMES(COUNT)
      WRITE(*,'(/,A,/)') 'After manipulation, the names are:'
      DO 10 I=1,COUNT
        PLACE1 = INDEX(FNAMES(I),' ')
        IF (PLACE1.EQ.0) PLACE1 = 20
        PLACE2 = INDEX(LNAMES(I),' ')
        IF (PLACE2.EQ.0) PLACE2 = 20
        WRITE(*,'(A,I2,A,$)') 'Name #',I,' = '
        WRITE(*,'(A)') FNAMES(I)(:PLACE1) // LNAMES(I)(:PLACE2)
  10  CONTINUE
      RETURN
      END
```

A sample run of this program is:

```
Enter name # 1 - Mickey Mouse
Enter name # 2 - Donald Duck
Enter name # 3 - Pluto
Enter name # 4 - Bugs Bunny
Enter name # 5 - Porky Pig
Enter name # 6 - Tweety Bird
Enter name # 7 -

After manipulation, the names are:

Name # 1 - Bugs
Name # 2 - Donald Bird
Name # 3 - Mickey Bunny
Name # 4 - Pluto Duck
Name # 5 - Porky Mouse
Name # 6 - Tweety Pig
```

Migrating FORTRAN 66 Programs to RT PC VS FORTRAN

Migrating FORTRAN 66 programs to RT PC VS FORTRAN, which is a FORTRAN 77 compiler, is made easier with the following FORTRAN 66 compatibility features, which are activated by issuing the y+ command-line option to the compiler.

DO Loops: Unlike in FORTRAN 77, DO loops in FORTRAN 66 are executed at least once no matter what parameters are initially set. DO loops are described in "DO Statements — Loop Control ✧" on page 8-10.

Common Blocks: Unlike in FORTRAN 77, character and numeric data in FORTRAN 66 can be assigned to the same common block. Common blocks are described in "COMMON Statements — Declaring Common Blocks ✦ ✧" on page 7-7.

Default Sizes of Variables: In FORTRAN 77, the default size for an integer or logical variable is 4 bytes. In FORTRAN 66, the default size is 2 bytes.

Character Assignment: Since FORTRAN 66 does not have a character data type, all character processing is accomplished with numeric variables and Hollerith constants. Hollerith constants can be assigned to numeric variables in assignment statements and DATA statements.

A Hollerith constant must be between 1-255 characters in length, and spaces are significant. A Hollerith constant can be written in either of two forms. One form is:

*n*H*cc* ...

n
is the number of characters in the constant.

c
is one of the *n* characters.

The other form of a Hollerith constant is:

'ccc...'

c
is a character in the constant.

Examples:

```
4HABCD

'ABCD'

6H   ABC

'   ABC'

These are all valid Hollerith constants.


4HABCDEF   - This has too many characters.

4HAB       - This does not have enough characters.

''         - The string length is 0.

'ABCD      - The trailing apostrophe is missing.

These are all invalid Hollerith constants.
```

When a Hollerith constant is part of an assignment or DATA statement and the variable has a length that is less than the length of the Hollerith constant, the rightmost characters are truncated. If the length of the variable is greater than the length of the Hollerith constant, trailing spaces are added.

Examples:

```
      INTEGER*4 I
      INTEGER*2 J
      REAL*4 X
      REAL*8 Y

      DATA I /'ABCD'/
C       I = 'ABCD'
      DATA J /1HA/
C       J = 'A '
      DATA K /'ABCD'/
C       K = 'A'

      X = 8HABCD  GH
C       X = 'ABCD'
      Y = 8HABCD  GH
C       Y = 'ABCD  GH'
```

Reading or writing numeric variables that take on character data can be done by using the A edit-descriptor in a FORMAT statement. If no length is specified, an implied length of the variable size is used. The A edit-descriptor is described in "A — Character Editing ✧" on page 11-16.

If a Hollerith constant is passed as a parameter to a subroutine or function subprogram, the actual value passed depends upon the form of the Hollerith constant. If a call to the subroutine subprogram X is made as CALL X(4HABCD), the address of a 4-byte numeric constant with the value "ABCD" is passed. If the call is made as CALL X('ABCD'), the Hollerith constant is treated as a FORTRAN 77 character string and both the address and the length of the string are passed. The CALL X('ABCD') must be received by a subroutine subprogram that expects a parameter with a character data type.

Hollerith constants cannot be used in logical expressions. However, a Hollerith constant can be assigned to a numeric variable which can be used in logical expressions.

Examples:

```
C     This is not permitted.
      IF(I .NE. 4HABCD) GOTO 100

J=4ABCD
C     This is permitted and accomplishes
C     what the previous code tries to do.
      IF(I .NE. J) GOTO 100
```

Index

Special Characters

A

B

C

D

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

Reader's Comment Form

IBM RT PC VS FORTRAN
Reference Manual

SH23-0130-0

Your comments assist us in improving our products. IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

For prompt resolution to questions regarding setup, operation, program support, and new program literature, contact the authorized IBM RT PC dealer in your area.

Comments:

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE:

International Business Machines Corporation
Department 79L, Building 4
Commerce Park & Eagle Road
Danbury, Connecticut 06810

Fold and tape

Fold and tape

Cut or Fold Along Line

International Business Machines Corporation
Department 79L, Building 4
Commerce Park and Eagle Road
Danbury, CT 06810

Printed in the
United States of America

SH23-0130