Data General

FORTRAN 5 Reference Manual

093-000085-04

FORTRAN 5 Reference Manual

093-000085-04

For the latest enhancements, cautions, documentation changes, and other information on this product, please see the Release Notice (085-series) supplied with the software.

NOTICE

FORTRAN 5
Reference Manual
093-000085

Revision History:

Original Release	- December 1972
First Revision	- August 1973
Second Revision	- November 1973
Third Revision	- April 1974
Fourth Revision	- October 1978

This document has been extensively revised from revision 03; therefore, change indicators have not been used.

Preface

Data General's FORTRAN 5 is a superset of Data General's FORTRAN IV and ANSI FORTRAN, and includes elements of IBM FORTRAN IV and Univac FORTRAN 5.

FORTRAN 5 is compatible with many of the language facilities described in the ANSI FORTRAN Standard (X3.9 - 1966) and offers several language extensions, some of which are:

- Full mixed-mode arithmetic
- Simple (free-formatted) I/O
- Generic library functions
- Library functions that allow full bit manipulation
- Assembly language routines that you call using FORTRAN 5 statements
- Data-initialization of blank and named COMMON in any program unit
- Alternate returns from a subroutine or function via a dummy variable

FORTRAN 5 both globally and locally optimizes the object code it produces, minimizing its execution time and the storage needs.

The FORTRAN 5 compiler performs the following global optimizations:

- It removes loop-invariant computations from DO-loops. These are expressions (and parts of expressions) which are not dependent upon the value of the index variable of the DO-loop. In this way, FORTRAN 5 computes the expression only once before entering the loop.
- It eliminates common subexpressions. Once it evaluates an expression, FORTRAN 5 reuses that value for each occurrence of the expression until your program redefines part of the expression, forcing re-evaluation.
- It optimally allocates registers. It preferentially maintains the values of frequently used expressions in the accumulators to minimize redundant load/store operations.

The FORTRAN 5 compiler performs the following local optimizations:

- It computes expressions involving only integer constants at compile time.
- It optimizes array subscripts by calculating array references having identical subscripts only once. FORTRAN 5 maintains such references in the accumulators wherever possible. It also absorbs constant elements in subscripts into the base address of the array at compile time to reduce the number of calculations.

The following features optimize the total amount of runtime main-memory space that your program occupies, and enables several users to use the same sequence(s) of instructions:

- Re-entrant Code -- all compiled and library FORTRAN 5 programs are fully re-entrant, letting several users share a single procedural task.
- Multitasking -- provides multiple execution paths that perform functions nonsequentially. You can coordinate many independent operations by letting each task share subroutines, data buffers, and disk files.
- Overlays -- allow you to divide large programs into disk-resident segments.

We have written this manual as a reference for the programmer who is familiar with FORTRAN programming, and who wants to become familiar with Data General's FORTRAN 5. The *FORTRAN 5 Reference Manual* describes all aspects of the language, its variations and extensions to ANSI FORTRAN, and instructs you in coding programs for both singletask and multitask environments.

DGC's FORTRAN 5 runs under Data General's Real-time Disk Operating System (RDOS) and Advanced Operating System (AOS). We present the information in this manual in a system independent manner.

We have organized the manual as follows:

Chapter 1	Lists the variations and extensions of FORTRAN 5 to ANSI FORTRAN, gives the character set and basic design of a FORTRAN 5 program, and includes a sample FORTRAN 5 program.
Chapter 2	Discusses the components of a FORTRAN 5 program (data types, user-defined names, expressions, etc.).
Chapter 3	Defines the structure and use of FORTRAN 5 assignment statements.
Chapter 4	Discusses the statements that allow you to change the flow of control in a FORTRAN 5 program.
Chapter 5	Discusses the concepts of file access and file organization; describes the structure and use of FORTRAN 5 input/output statements.
Chapter 6	Describes the FORMAT statement and its various forms.
Chapter 7	Gives a detailed description of all FORTRAN 5 specification statements and includes discussions of array dimensioning, storage allocation, and FORTRAN 5 element definition.
Chapter 8	Discusses FORTRAN 5 subprograms including details on function subprograms, subroutine subprograms, block data subprograms, library functions, and statement functions.
Chapter 9	Discusses FORTRAN 5 programming in a multitask environment and gives a sample multitask program.
Appendix A	Contains an alphabetized list of all FORTRAN 5 statements and their functions.
Appendix B	Lists compiler error and warning messages.
Appendix C	Contains a list of runtime math errors.
Appendix D	Contains a table of the ASCII character set.

Required Manuals

You should supplement your reading of this manual with the following: *Real-time Disk Operating System (RDOS) Reference Manual* (093-000075), *FORTRAN 5 Programmer's Guide (RDOS)* (093-000227), *Advanced Operating System (AOS) Programmer's Manual* (093-000120), and *FORTRAN 5 Programmer's Guide (AOS)* (093-000154).

Reader, Please Note:

We use these conventions for command formats in this manual:

COMMAND required *[optional]* ...

Where	**Means**
COMMAND	You must enter the command (or its accepted abbreviation) as shown.
required	You must enter some argument (such as a filename). Sometimes, we use: $\left\{\begin{matrix}\text{required}_1\\ \text{required}_2\end{matrix}\right\}$ which means you must enter *one* of the arguments. Don't enter the braces; they only set off the choice.
[optional]	You have the option of entering some argument. Don't enter the brackets; they only set off what's optional.
...	You may repeat the preceding entry or entries. The explanation will tell you exactly what you may repeat.

Additionally, we use certain symbols in special ways:

Symbol	**Means**
)	Press the NEW-LINE or RETURN key on your terminal's keyboard.
□	Be sure to put a space here. (We use this only when we must; normally, you can see where to put spaces.)

All numbers are decimal unless we indicate otherwise; e.g., 35_8.

Finally, we usually show all examples of entries and system responses in THIS TYPEFACE. But, where we *must* clearly differentiate your entries from system responses in a dialog, we will use

THIS TYPEFACE TO SHOW YOUR ENTRY)
THIS TYPEFACE FOR THE SYSTEM RESPONSE

End of Preface

Contents

Chapter 1 - The FORTRAN Program

Variations and Extensions 1-1
Program Units and Procedures 1-3
Elements of a FORTRAN 5 Program Unit 1-3
  Statements and Lines 1-3
  Comment Lines (C) 1-4
  Comments on a Statement Line 1-4
  Optionally Compiled Line (X) 1-4
  Blank Lines 1-4
  Statement Labels 1-4
  FORTRAN 5 Character Set 1-4
The INCLUDE Statement 1-5
FORTRAN 5 Sample Program 1-5

Chapter 2 - FORTRAN 5 Components

Data Types 2-1
Constants 2-2
  Integer Constants 2-2
  Octal Constants 2-2
  Real Constants 2-2
  Double Precision Constants 2-3
  Complex Constants 2-4
  Double Precision Complex Constants 2-5
  Logical Constants 2-6
  Hollerith and String Constants 2-6
    Using Angle Brackets 2-6
    String Storage 2-7
    Using Hollerith and String Constants 2-8
Symbolic Names 2-8
Variables 2-9
  Data Types of Variables 2-9
Arrays 2-9
  Subscripts 2-9
  Array Storage 2-9
Expressions 2-10
  Arithmetic Expressions 2-10
  Rules for Evaluating Arithmetic Expressions 2-11
    Data Types of Operands 2-11
    Operator Precedence 2-13
    Use of Parentheses 2-13
Relational and Logical Expressions 2-14
  Relational Expressions 2-14
  Logical Expressions 2-14
  Rules for Evaluating Logical and Relational Expressions 2-14
Function References 2-15

Chapter 3 - Assignment Statements

Arithmetic Assignment Statement 3-1
Logical Assignment Statement 3-3
ASSIGN Statement 3-3

Chapter 4 - Control Statements

GO TO Statements 4-1
Unconditional GO TO 4-1
Computed GO TO 4-2
Assigned GO TO 4-2
IF Statements 4-3
Arithmetic IF 4-3
Logical IF 4-3
DO 4-4
Execution of the DO 4-4
Nested DO Loops 4-4
Extended Range of the DO Loop 4-5
DO Loop Restrictions 4-5
CONTINUE 4-6
PAUSE 4-6
STOP 4-7
END 4-7

Chapter 5 - Input/Output Statements

Input/Output Unit Numbers 5-1
Format Identifiers 5-1
Input/Output Records 5-2
File Access 5-2
Sequential Access 5-2
Line Access 5-2
Stream Access 5-2
Direct Access 5-2
Data Transfers 5-2
Types of File Organization 5-3
Input/Output Lists 5-3
Simple Lists 5-3
DO-implied Lists 5-4
Options List 5-4
FORTRAN 5 Input/Output Statements 5-5
I. Formatted Input/Output 5-5
READ 5-5
WRITE 5-6
ENCODE 5-6
DECODE 5-7
Simple Formatted I/O 5-7
READ, PRINT, PUNCH 5-7
II. Free-Formatted Input/Output 5-8
READ FREE 5-8
WRITE FREE 5-8
ACCEPT 5-9
TYPE 5-9

III. Unformatted Input/Output 5-10
READ BINARY 5-10
WRITE BINARY 5-10
IV. Auxiliary Input/Output Statements 5-11
OPEN 5-11
CLOSE, ENDFILE 5-14
DELETE 5-14
RENAME 5-15
REWIND, BACKSPACE 5-15

Chapter 6 - FORMAT Statement

Numeric Field Descriptors 6-3
I Format Code 6-3
O Format Code 6-4
Y Format Code 6-5
B Format Code 6-6
F Format Code 6-7
E Format Code 6-8
D Format Code 6-9
G Format Code 6-10
Logical Field Descriptor 6-11
L Format Code 6-11
String and Editing Descriptors 6-12
A Format Code 6-12
R Format Code 6-13
S Format Code 6-14
Hollerith and String Constants 6-15
X Format Code 6-16
T Format Code 6-16
Scale Factor 6-17
Group Repeat Specifications 6-18
Carriage Control 6-19
Suppressing a New-line Character (Z) 6-19
Field Separators 6-19
Format and I/O Statement Association 6-21
Complex Data in I/O Lists 6-21
Multiple-Record Formatting 6-21
Summary of Rules for Formatted I/O 6-22
Input 6-22
Output 6-22

Chapter 7 - Specification Statements

DIMENSION 7-1
Type-Statements 7-3
IMPLICIT 7-3
COMMON 7-4
STATIC 7-6
EQUIVALENCE 7-6
PARAMETER 7-8
EXTERNAL 7-9
DATA 7-10
COMPILER 7-12
OVERLAY 7-13

Chapter 8 - Subprograms

Functions. 8-1
Statement Functions. 8-1
Function Subprograms. 8-2
FUNCTION. 8-3
Arguments of Function and Subroutine Subprograms. 8-3
Subroutine Subprograms 8-4
SUBROUTINE. 8-4
CALL 8-5
RETURN 8-5
Library Functions. 8-6
Logical Word Operations. 8-11
Single Bit Operations. 8-12
FLD Function 8-13
BYTE Function 8-14
Block Data Subprogram 8-14

Chapter 9 - Multitask Programming

Introduction 9-1
Tasks. 9-1
Task Priorities 9-1
Task Identification Numbers 9-2
The Multitask Environment. 9-2
Task Control Blocks. 9-2
FORTRAN 5 Multitasking Statements 9-2
Events 9-3
TASK. 9-3
KILL 9-5
WAIT. 9-5
SUSPEND 9-6
WAKEUP 9-6
ANTICIPATE. 9-7
FORTRAN 5 Multitasking Example 9-8

Appendix A - Alphabetized List of FORTRAN 5 Statements

Appendix B - Compiler Error and Warning Messages

Appendix C - Runtime Math Errors

Appendix D - ASCII Character Set

Tables

Table	Caption	
1-1	ANSI Versus DGC FORTRAN Data Storage	1-1
1-2	Special Symbols in FORTRAN 5	1-4
2-1	FORTRAN 5 Data Types	2-1
2-2	Special FORTRAN 5 Symbols	2-6
2-3	FORTRAN 5 Operators	2-10
2-4	Evaluating Mixed Data Types for Addition, Subtraction, Multiplication, and Division	2-11
2-5	Evaluating Mixed Data Types for Exponentation	2-12
2-6	Evaluating Mixed Data Types for Replacement Operations	2-12
2-7	Relational Operators	2-14
2-8	Logical Operators	2-14
2-9	Logical Truth Table	2-14
2-10	Logical and Relational Truth Tables	2-15
3-1	Assignment Statement Conversion Rules	3-2
5-1	Data Transfers	5-3
5-2	I/O Statements with No I/O Lists	5-4
5-3	OPEN Statement Options	5-12
6-1	Print File Examples	6-20
6-2	Format Control Examples	6-23
8-1	Trigonometric Functions	8-7
8-2	Arithmetic and Conversion Functions	8-8
8-3	Logical Word Operation	8-11
8-4	Single Bit Operations	8-12

Illustrations

Figure	Caption	
1-1	Sample FORTRAN 5 Program	1-6
2-1	Internal Representation of an Integer	2-2
2-2	Internal Representation of a Real Number	2-2
2-3	Internal Representation of a Double Precision Number	2-3
2-4	Internal Representation of a Complex Constant	2-4
2-5	Internal Representation of a Double Precision Complex Number	2-5
2-6	Left-justified String Storage	2-7
2-7	Right-justified String Storage	2-8
2-8	Ranking Data Types	2-11
2-9	Arithmetic Operator Precedence	2-13
2-10	Operator Precedence Example	2-13
2-11	FORTRAN 5 Operator Precedence	2-15
6-1	Nested Parentheses Example	6-23
7-1	Storage in Common Area BLK1	7-5
9-1	Multitasking Example	9-9

Chapter 1
The FORTRAN Program

Variations and Extensions

DGC's FORTRAN 5 is compatible with the FORTRAN language standards specified by the American National Standards Institute (ANSI Standard, X3.9 - 1966) except for the following variations and extensions, listed by chapter number:

Chapter 1 -- The FORTRAN Program

1. You may place a comment on the same line as a statement by terminating the statement with a semicolon. Comments may occupy as many character positions as exist between the semicolon and the end of the line.

2. FORTRAN 5 compiles statements that contain an X in column 1 if you specify the /X global switch in the FORTRAN 5 compilation command line; otherwise, they are treated as comments. (See the *FORTRAN 5 Programmer's Guide* for specific operating instructions.)

3. You may include blank lines within a program unit.

4. An INCLUDE statement lets you insert additional program source files for FORTRAN 5 to compile as part of the current program. Compilation of the current file then resumes at the statement following the INCLUDE statement.

Chapter 2 -- FORTRAN Components

1. FORTRAN 5 supports double precision complex data.

2. ANSI specifies the number of storage units that each FORTRAN 5 datum should occupy according to its data type, where *storage unit* is implementation-defined. DGC's basic storage unit is one 16-bit word. Storage of DGC's FORTRAN 5 data compared to ANSI storage is shown in Table 1-1.

Table 1-1. ANSI Versus DGC FORTRAN Data Storage

Data Type	DGC Words	ANSI Storage Units
integer	1	1
real	2	1
double precision	4	2
complex	4	2
double precision complex	8	-
logical	1	1

3. You can use a PARAMETER statement to assign symbolic names to constants or expressions.

4. You can specify octal constants in your source program in the form nK.

5. An array can have any number of dimensions.

6. A subscript expression can be any valid arithmetic expression; FORTRAN 5 will convert it to type integer if necessary.

7. When you dimension an array, the lower bound of a subscript does not have to be 1; it may be zero or negative. You separate the lower and upper bounds of the subscript by a colon.

8. You may combine all types of arithmetic data in an arithmetic expression.

9. In general, FORTRAN 5 converts expressions to the data type required by the context in which they appear; e.g.,

```
GO TO (10,20), X
```

where X is real is interpreted as:

```
GO TO (10,20), IFIX(X)
```

10. Unary plus and minus may directly follow another arithmetic operator; no parentheses are necessary. For example, A*-B is valid and is equivalent to A*(-B).

11. You may use the boolean operators .AND., .OR., .XOR., and .NOT. with integer data. They are equivalent to the functions IAND, IOR, IXOR, and NOT respectively.

Chapter 3 -- Assignment Statements

A Hollerith or string constant may appear on the right-hand side of the equal sign in any assignment statement (except an ASSIGN statement). You may assign the constant to an entity of any data type. FORTRAN 5 assigns the ASCII values of the first *n* characters of the constant, where *n* is the number of bytes in the entity to the left of the equal sign (2 for integer or logical, 4 for real, 8 for double precision or complex, and 16 for double precision complex).

Chapter 4 -- Control Statements

1. FORTRAN 5 treats an assigned GO TO statement as an unconditional GO TO statement; i.e., the statement label list is not checked for validity.
2. If the index value in a computed GO TO statement is out of range, control passes to the next executable statement.
3. The control variable of a DO statement may be of type integer, real, or double precision.
4. The parameters of a DO statement may be any expression of type integer, real, or double precision.
5. You may terminate the range of a DO with any statement other than another DO statement.

Chapter 5 -- Input/Output Statements

We have implemented the following additional I/O statements:

READ FREE WRITE FREE	free-form ASCII transfer (formatting done by I/O processor)
READ BINARY WRITE BINARY	unformatted I/O
READ TAPE WRITE TAPE	unformatted I/O
READ INPUT TAPE WRITE OUTPUT TAPE	ASCII formatted I/O
READ PRINT PUNCH	card reader input line printer output high speed punch output
ENCODE DECODE	memory-to-memory transfer
ACCEPT TYPE	conversational I/O
OPEN CLOSE DELETE RENAME	auxiliary I/O

Chapter 6 -- The FORMAT Statement

1. The FORMAT statement permits the following nonstandard field descriptors:

O	octal transfer
Y	hexadecimal transfer
B	binary transfer
T	tabulation to a specified carriage position
S	transmission of string data
R	transmission of ASCII data, right-justified
Z	suppression of new-line character at end of record

2. The control variable of a DO-implied list may be of type integer, real, or double precision.
3. The parameters of a DO-implied list may be any expression of type integer, real, or double precision.

Chapter 7 -- Specification Statements

1. You may DATA-initialize elements in blank COMMON.
2. You may DATA-initialize elements in blank and named COMMON in any program unit.
3. You may use an IMPLICIT statement to reconfigure the name rule for the symbolic names of variables, arrays, and functions.
4. You may use a STATIC statement to put specified variables in static storage. FORTRAN 5 places static storage variables in a fixed area in memory (rather than on the runtime stack), where they may retain their values between successive executions of the program unit.

5. You may use a PARAMETER statement to give a symbolic name to a constant or to an expression.

6. A COMPILER statement lets you specify one or more of the following options:

   DOUBLE PRECISION -- FORTRAN 5 will store all real program variables and constants in double precision form, and performs all real calculations in double precision.

   FREE -- READ/WRITE statements that do not reference a FORMAT statement will transfer free-formatted ASCII data (rather than binary data, which is the default option).

   STATIC -- FORTRAN 5 will place in static storage all variables and arrays which are not in COMMON or are not arguments to a subprogram (rather than on the runtime stack).

7. An OVERLAY statement specifies a name you can use for referencing an overlay.

Chapter 8 -- Subprograms

1. FORTRAN 5 permits alternate returns from functions and subroutines.

2. We have added a number of library functions to the standard FORTRAN 5 library. The bit manipulation functions IAND, IOR, IXOR, IEOR, ITEST, and FLD, together with two subroutine calls, ISET and ICLR, allow full bit manipulation. The byte manipulation function is called BYTE.

3. A statement function reference can appear to the left of an equal sign if the function reference expands to a variable name, an array element name, a FLD reference, or a BYTE reference.

4. FORTRAN 5 does not associate a data type with the name of a statement function.

5. The actual arguments of a statement function reference and the dummy arguments of a statement function definition have to agree only in number and order; they do not have to agree in type. FORTRAN 5 does not perform type conversions on mismatched arguments.

Chapter 9 -- Multitask Programming

The following statements provide multitasking capabilities:

ANTICIPATE
KILL
SUSPEND
TASK
WAIT
WAKEUP

Program Units and Procedures

A FORTRAN 5 program contains one or more program units. A *program unit* consists of a sequence of statements, with optional comment lines; it is either a main program or a subprogram. You compile each unit separately, then bind them together at a later time.

A *main program unit* does not contain a FUNCTION, SUBROUTINE, or BLOCK DATA statement. A FORTRAN 5 program may have only one main program unit.

A *subprogram unit* contains a FUNCTION, SUBROUTINE, or BLOCK DATA statement. These statements define a function subprogram, a subroutine subprogram, or a block data subprogram respectively.

A *procedure* is a subroutine, an external function, a statement function, or an intrinsic function.

Elements of a FORTRAN 5 Program Unit

A FORTRAN 5 program unit contains statements (with optional comment lines) terminated by an END statement. An END statement is a single line that consists solely of the letters E N D in that order. FORTRAN 5 ignores any statements following this line.

Statements and Lines

A FORTRAN 5 statement is either executable or nonexecutable. Executable statements specify actions. Nonexecutable statements describe the characteristics, arrangement, and the initial values of variables; contain editing information; specify statement functions; and classify program units.

A *statement* must start at character position 7, or beyond, and terminate at or before character position 72. You can press the tab key once to tabulate to position 9, or you can press the space bar 6 or more times. After a statement label, pressing the tab key also positions you at position 9.

You write statements in *lines*. The first line of a statement is the initial line, and you must put either a 0 or a blank in character position 6 to indicate an initial line. (Using the tab key inserts blanks in the character positions skipped.) If a statement is too long to fit on one line, follow the initial line with continuation lines, for which you must put a character other than 0 or blank in character position 6. You may not intersperse any comment or blank lines between an initial line and any of its continuation lines.

You may identify each executable statement by a statement label, so that other statements can refer to it.

FORTRAN 5 eliminates blank characters when translating a program.

Comment Lines (C)

A line that has a C in character position 1 is a *comment line*. You may write the comment anywhere in the line following the C.

A comment line is not a statement and has no effect on the execution of the program. It provides documentation to make a program easier to read and understand.

Comments on a Statement Line

You may terminate a statement by following the statement with a semicolon. A semicolon in column 7, or any character position thereafter, reserves the remainder of the line for an optional comment. (A semicolon appearing within a Hollerith or string constant is considered part of the constant. A semicolon in column 6 indicates a continuation line. At compile time, FORTRAN 5 flags a semicolon in any of columns 1 through 5 as an illegal statement label.)

Optionally Compiled Line (X)

If you wish to optionally compile certain lines of your program, enter the letter X in character position 1. You indicate whether or not to compile these lines at compile time by specifying the /X global switch in the FORTRAN 5 compilation command line. Otherwise, the compiler treats them as comment lines (see the *FORTRAN 5 Programmer's Guide* for operating instructions).

Blank Lines

DGC's FORTRAN 5 allows blank lines within a program unit. Like comment lines, they do not affect program execution.

Statement Labels

If character position 1 does not contain a C or an X, FORTRAN 5 reserves character positions 1 through 5 for a statement label. If character position 1 contains an X, character positions 2 through 5 are reserved for a statement label. The label must be an unsigned integer of 1 to 5 digits (or 1 to 4 digits if character position 1 contains an X). You may place the label anywhere in character positions 1 through 5 (or 2 through 5). Leading zeros are not significant; e.g., 12 and 00012 are treated as the same label. FORTRAN 5 ignores statement labels on continuation lines, specification statements, and on statements that you do not reference.

FORTRAN 5 Character Set

The ASCII characters that make up the FORTRAN 5 character set are the letters A - Z (ASCII octal equivalents 101 - 136), the digits 0 - 9 (ASCII octal equivalents 60 -71), and the special symbols shown in Table 1-2.

Table 1-2. Special Symbols in FORTRAN 5

Symbol	Meaning	ASCII 7-Bit Octal Code
␣	Blank or Space	040
"	Quotation Mark	042
$	Dollar Sign	044
'	Apostrophe	047
(	Left Parenthesis	050
)	Right Parenthesis	051
*	Asterisk	052
+	Plus	053
,	Comma	054
-	Minus	055
.	Period	056
/	Slash	057
:	Colon	072
<	Left Angle Bracket	074
=	Equal Sign	075
>	Right Angle Bracket	076

Any printable character may appear in a comment line or follow a semicolon. Hollerith and string constants may contain any printable or nonprintable character; you specify a nonprintable character by enclosing its ASCII octal equivalent in angle brackets.

The INCLUDE Statement

An INCLUDE statement allows you to insert a FORTRAN 5 source file in the current FORTRAN 5 program.

Format

INCLUDE "filename"

where:

filename is the name of a FORTRAN 5 source file.

Statement Execution

When the compiler encounters an INCLUDE statement, it compiles the statements of the included file, then resumes compilation with the statement following the INCLUDE statement.

An INCLUDE statement may appear anywhere in your program and anywhere in an included file. It must occupy only one line of program text; continuation lines are not permitted.

You must enclose the filename in quotation marks, and it must be the complete name of the file, including any directory specifiers and extensions. The included file may contain any legal FORTRAN 5 statements.

You may suppress output of included files to the listing file by specifying the /I global switch in the FORTRAN 5 compilation command. The numbering of lines following the included file's text is not affected by this switch; i.e., the number of lines in the included file are reflected in the line number count whether or not you opt to output the included file to the program listing.

Examples

```
INCLUDE "DECLRTN"

INCLUDE "PARAM.FR"

INCLUDE "DP1:DECLR.FR"
```

FORTRAN 5 Sample Program

Figure 1-1 is a sample program that illustrates the use of and interaction between some basic FORTRAN 5 statements. If you want to run this program, simply enter the program using a Data General editor, compile it, bind it, and then execute it (see the *FORTRAN 5 Programmer's Guide* for the instructions). The messages displayed on the console will be:

HOW MANY VALUES DO YOU WANT TO AVERAGE?

TYPE 1 TO CHECK DATA, 0 TO CONTINUE

TYPE ANY NON-BLANK CHARACTER AFTER '?' TO INDICATE THAT YOU WISH TO CHANGE A VALUE

After each colon, the system will await your response. When you have answered all questions and entered all values, FORTRAN 5 will calculate the mean and standard deviation of the values and display them on the console.

```
C       THIS PROGRAM CALCULATES THE MEAN AND STANDARD DEVIATION
C       OF UP TO 99 SINGLE-PRECISION REAL VALUES.  THE PROGRAM
C       READS DATA FROM THE CONSOLE AND THEN ALLOWS VERIFICATION
C       AND CORRECTION OF THE ORIGINAL VALUES.

        REAL A(99)
        INTEGER COUNT


C  READ ALL THE VALUES, FREE-FORMATTED

        ACCEPT "<NL>HOW MANY VALUES DO YOU WANT TO AVERAGE?  ", COUNT, "<NL>"
        IF (COUNT .GT. 99) STOP "TOO MANY VALUES TO AVERAGE"

        DO 1 I = 1, COUNT
        WRITE (10,100) I
100     FORMAT(Z,"ENTER A(",I2,"):  ")
1       READ FREE (11) A(I)


C  IF VERIFICATION IS DESIRED, TYPE OUT EACH VALUE,
C  FOLLOWED BY "?", AND WAIT FOR A RESPONSE.
C  IF THE RESPONSE IS NOT A BLANK OR "<NL>", REREAD THE VALUE.

        ACCEPT "<NL>TYPE 1 TO CHECK DATA, 0 TO CONTINUE  ", I
        IF (I.EQ.0) GOTO 4

        TYPE "<NL>TYPE ANY NON-BLANK CHARACTER AFTER '?' TO INDICATE"
        TYPE "THAT YOU WISH TO CHANGE A VALUE<NL>"
        DO 3 I=1, COUNT
2       WRITE (10,101) I, A(I)
101     FORMAT(Z,"A(",I2,") =",G16.6,"    ?  ")
        READ (11,102) J
102     FORMAT (A2)
        IF (J.EQ."  ") GOTO 3
        WRITE (10,103) I
103     FORMAT(Z,"NEW VALUE FOR A(",I2,") = ")
        READ FREE(11) A(I)
        GOTO 2
3       CONTINUE


C  CALCULATE AND OUTPUT THE AVERAGE AND STANDARD DEVIATION

4       DO 5 I= 1, COUNT
        SUM = SUM + A(I)                ;SUM THE VALUES
5       SUMSQ = SUMSQ + A(I)**2         ;SUM THEIR SQUARES

        AVG = SUM / COUNT
        SUMSQ = SUMSQ - COUNT * AVG**2
        STANDARD DEVIATION = SQRT(SUMSQ/COUNT)

        TYPE "<NL><NL>AVERAGE VALUE IS", AVG
        TYPE "STANDARD DEVIATION IS", STANDARD DEVIATION

        END
```

Figure 1-1. Sample FORTRAN 5 Program

End of Chapter

Chapter 2
FORTRAN 5 Components

Five types of components appear in FORTRAN 5 statements:

- Constants
- Variables
- Arrays and array elements
- Expressions
- Function references

Many FORTRAN 5 components have a data type associated with them (those that do not are called *typeless*). FORTRAN 5 distinguishes six types of data:

- Integer
- Real
- Double precision
- Complex
- Double precision complex
- Logical

Data Types

Many components of a FORTRAN 5 statement have one of several data types. The type may be implied by the name rule (see the section "Symbolic Names"), or be inherent in the makeup of the component, or you may reconfigure the name rule by using an IMPLICIT or Type-statement. Table 2-1 shows the FORTRAN 5 data types, their meanings, and the amount of storage required by each in words.

Table 2-1. FORTRAN 5 Data Types.

Data Type	Meaning	Amount of Storage Required (Words)
Integer	A whole number.	1
Real	A decimal number which can be a whole number, a decimal fraction, or a combination of the two.	2
Double Precision	Similar to a real number, but has more than twice the degree of accuracy in its representation.	4
Complex	An ordered pair of real values; the first value is the real part of the number, and the second value is the imaginary part.	4
Double Precision Complex	An ordered pair of double precision values; the first value is the real part of the number, and the second value is the imaginary part.	8
Logical	The logical value *true* or *false.*	1

The data type of an entity not only determines the amount of space FORTRAN 5 allocates to that entity, but also its range and accuracy.

Hollerith and string constants do not have an associated data type, so they are *typeless.* They assume the data type of the context in which they appear (see the section "Hollerith and String Constants").

Constants

A *constant* is a fixed value that does not change during program execution. The value of a constant can be arithmetic or logical, or the constant can be a character string.

See the PARAMETER statement in Chapter 7 for information on assigning a symbolic name to a constant.

Integer Constants

An *integer constant* is an optionally-signed whole number. It may have a positive, negative, or zero value. If you do not specify an explicit sign, the constant is positive.

An integer constant may contain only the digits 0 through 9 and an optional sign. Leading zeros and interspersed blanks have no effect on the value of the constant.

FORTRAN 5 stores an integer in two's complement form, using one full 16-bit word. The allowable range is -2**15 to +2**15-1 (-32,768 to 32,767 decimal). (If the integer is outside this range, FORTRAN 5 stores it as a real constant.) An integer's internal representation is shown in Figure 2-1.

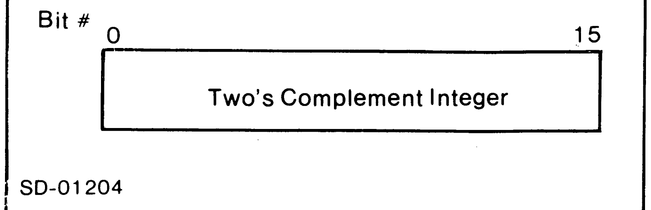


Figure 2-1. Internal Representation of an Integer

Examples:

Valid	Invalid
-125	12.5
0	66,391
+4525	+33333
91	

Octal Constants

An *octal constant* (an alternative way of representing an integer) consists of a string of digits followed by the letter K. It must be in the range 0 to 177777 inclusive.

You may not sign an octal constant. It may contain only the digits 0 through 7 and the letter K.

Examples:

Valid	Invalid
10K	-10K
777K	77.7K
1K	89K

Real Constants

A *basic real constant* is an optionally-signed string of decimal digits with a decimal point. You may place the decimal point anywhere in the decimal string. A basic real constant may have a positive, negative, or zero value.

A *real constant* can be a basic real constant, a basic real constant followed by a real exponent, or an integer constant followed by a real exponent.

A *real exponent* is the letter E, followed by an optionally-signed integer constant. You must specify the constant preceding the letter E, even if it is 1 (for example, 1E-04). The integer constant following the E may be explicitly 0, but may not be blank.

A *real constant* may contain only the numbers 0 through 9, the letter E, and the symbols -- period (.), plus (+), and minus (-).

FORTRAN 5 stores a single precision real number in two memory words, with the high-order word preceding the low-order word. It must be in the range 5.4 x 10**-79 to 7.2 x 10**-75. It has the significance of 6 hexadecimal digits (approximately 7.2 decimal digits). Its internal representation is shown in Figure 2-2.

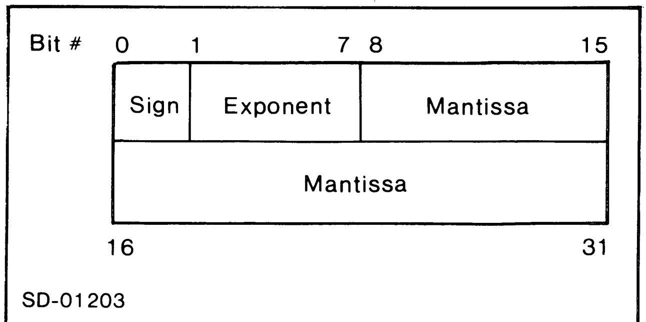


Figure 2-2. Internal Representation of a Real Number

Bits 1 through 7 in Figure 2-2 contain the exponent. This is the power to which 16 must be raised in order to give the correct value to the number. FORTRAN 5 uses *Excess 64* representation so that the exponent field may include negative and positive exponents. This means that the value in the exponent field is 64 greater than the true value of the exponent. If the exponent field is zero, the true value of the exponent is -64. If the exponent field is 64, the true value of the exponent is zero. If the exponent field is 127, the true value of the exponent is 63.

Bits 8 through 31 in Figure 2-2 contain the mantissa which is in true form. It is a binary fraction with a hexadecimal point implied immediately to the left of bit 8. The mantissa is always hexadecimally normalized (left-justified).

Examples:

Valid	**Invalid**
+.005678	$45.67
15.E-04	9,876.53
-005E2	-65.4E

Double Precision Constants

A *double precision constant* is a basic real constant or an integer constant followed by a double precision exponent. It may have a positive, negative, or zero value.

A *double precision exponent* is the letter D, followed by an optionally-signed integer constant. You must specify the constant preceding the letter D, even if it is 1 (for example, 1D+03). The integer constant following the D may be explicitly 0, but may not be blank.

FORTRAN 5 stores a double precision number in four words, and stores the sign and exponent in the same manner as real numbers. A double precision constant has the significance of 14 hexadecimal digits (approximately 16.8 decimal digits). Its internal representation is shown in Figure 2-3.

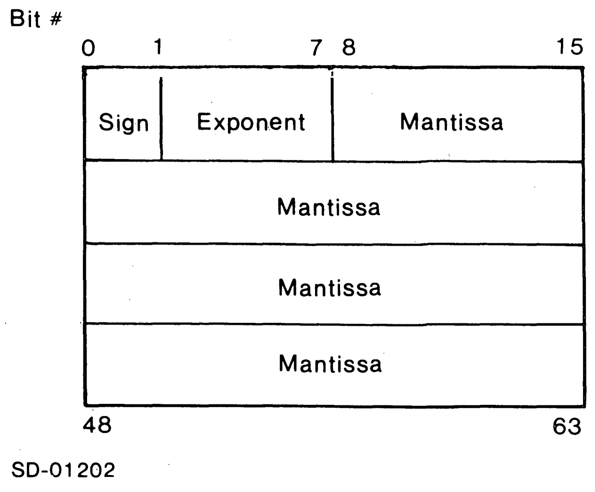


Figure 2-3. Internal Representation of a Double Precision Number

Examples:

```
-21987D0
5.0D-3
+.203D+15
```

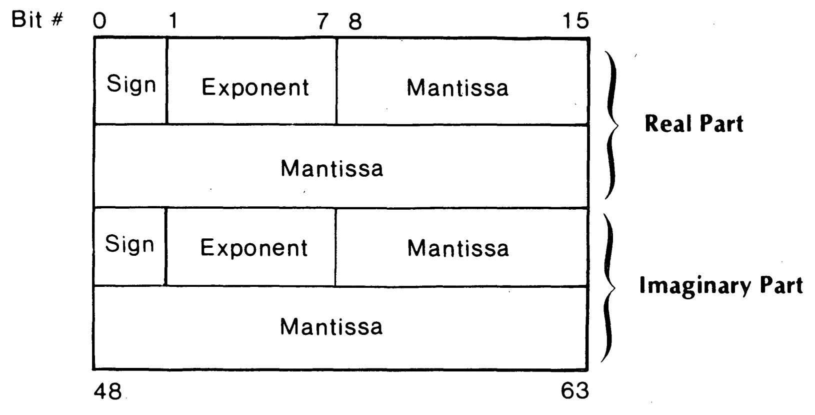
Complex Constants

A *complex constant* consists of a left parenthesis followed by an ordered pair of real constants separated by a comma, followed by a right parenthesis. The first real constant is the real part of the complex constant and the second is the imaginary part.

FORTRAN 5 stores a complex constant as two real constants, thus requiring four words. It stores the real part in the first two words and the imaginary part in the second two words. The internal representation is shown in Figure 2-4.

Examples:

```
(3.2,1.86)
(2.1,0.0)
(5.0E3,-2.12)
```



SD-01201

Figure 2-4. Internal Representation of a Complex Constant

Double Precision Complex Constants

A *double precision complex constant* consists of a left parenthesis followed by an ordered pair of optionally-signed double precision constants, separated by a comma, and followed by a right parenthesis. The first double precision constant is the real part of the double precision complex constant and the second is the imaginary part.

FORTRAN 5 stores a double precision complex constant as two double precision constants, thus requiring eight words. It stores the real part in the first four words and the imaginary part in the second four words. The internal representation is shown in Figure 2-5.

Examples:

```
(346.012D-5,.0045D+3)
(31D0,45D03)
(-9.3D-1,+.006D9)
```

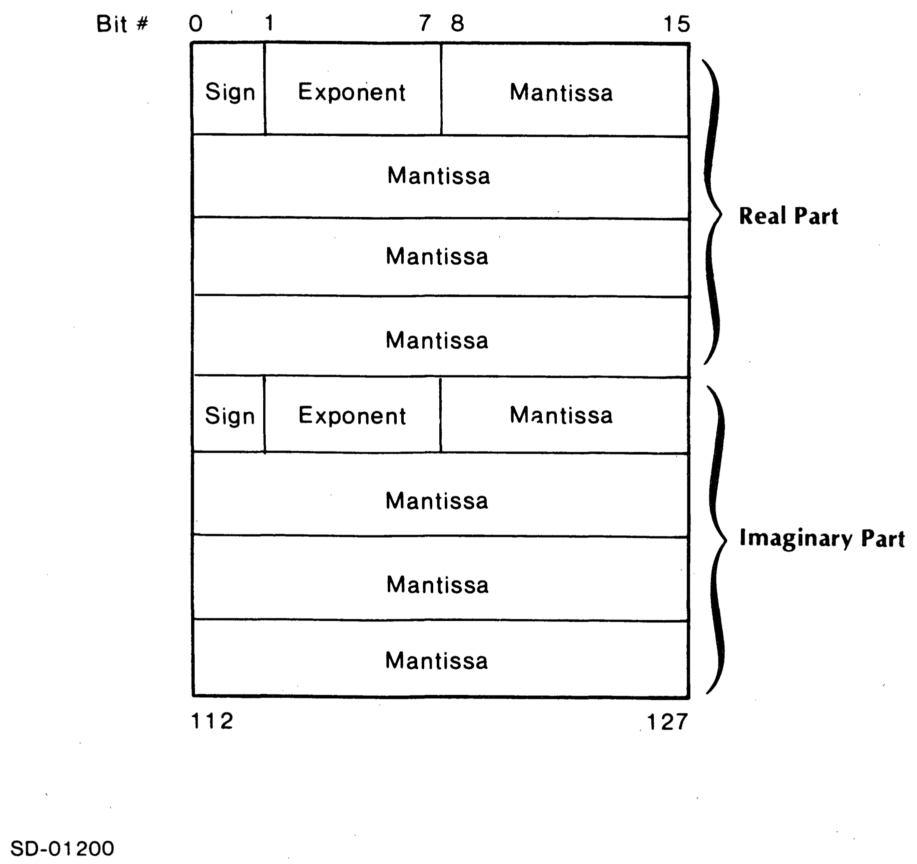


Figure 2-5. Internal Representation of a Double Precision Complex Number

Logical Constants

A *logical constant* has either a true value or a false value and may take only one of two forms: .TRUE. or .FALSE..

You must include the delimiting periods as part of the constant.

FORTRAN 5 stores one word of all zeros for a false value and a nonzero word for a true value.

Hollerith and String Constants

Hollerith and string constants are strings of ASCII characters; FORTRAN 5 handles each string character by character without interpretation.

A *string constant* may take one of two forms:

```
"string"
'string'
```

A string constant enclosed in quotation marks may not contain a quotation mark within the string. A string constant enclosed in apostrophes may not contain an apostrophe within the string.

Examples:

```
"PRINTER"
'NOT□AGAIN'
"CAN'T"
```

A *Hollerith constant* is a string of characters that is preceded by a character count and the letter H or R. It may take one of two forms:

```
nHstring
nRstring
```

Each n represents the number of characters in the string (not including the H or R). It must be an unsigned, nonzero integer constant. The H specifies that you want the string of characters left-justified in storage. The R specifies that you want the string of characters right-justified in storage.

Examples:

```
5HSTART
6REND□OF
```

Note that blanks within a Hollerith or string constant are significant.

Using Angle Brackets

Any printable ASCII character may appear in a Hollerith or string constant. To include an ASCII control character in a Hollerith or string constant, you must enclose the octal code of the character in angle brackets. Only a single code may appear in each set of angle brackets. For example, you may wish to include the nonprinting bell character in a Hollerith or string constant:

```
"DATA□FOLLOWS:<7>"
'DATA□FOLLOWS:<7>'
14HDATA□FOLLOWS:<7>
```

The above three examples are equivalent strings. Note that the character count of the Hollerith constant increases by only 1 to include the angle brackets and the enclosed octal code.

In addition to ASCII octal codes, Table 2-2 shows the special symbols that FORTRAN 5 recognizes when enclosed in angle brackets.

Table 2-2. Special FORTRAN 5 Symbols

Symbol	Meaning	ASCII 7-Bit Octal Code
NUL	Null	000
BEL	Bell	007
HT	Horizontal Tab	011
LF	Line Feed	012
FF	Form Feed	014
CR	Carriage Return	015
NL	New Line	012 or 015

Note that you may not use the PARAMETER statement to define additional symbolic names for ASCII octal codes. The names listed above are the only legal names that you may use in this context.

If you enclose a number in angle brackets that is not octal, is larger than 377_8 or is a parameter name, FORTRAN 5 will treat the angle brackets and the enclosed number or name in the same manner as the other characters in the string. For example, if you input the string "ABCDEF<378>", it will be output as ABCDEF<378>. If you specify the statement, PARAMETER TAB = 11k, and then later use the string constant "$8.41<TAB>", FORTRAN 5 will output the constant as $8.41<TAB>.

As previously described, a Hollerith constant treats angle brackets and the enclosed symbol or ASCII octal code as one character. However, if you enclose a number in angle brackets that is not octal, is larger than 377_8, or is a parameter name, you must reflect this in the character count. Each angle bracket counts as one character as does each digit or letter within the angle brackets. The following are valid Hollerith constants:

```
12HEND□OF□LINE<15>
12HEND□OF□PAGE<FF>
15HNEW□CODE□IS<38>
```

String Storage

In a 16-bit machine word, FORTRAN 5 stores one character per byte, two characters per word. It left justifies strings of the form

3HEND or "END" or 'END'

A null byte terminates the string if there are an odd number of characters; two null bytes terminate the string if there are an even number of characters (see Figure 2-6).

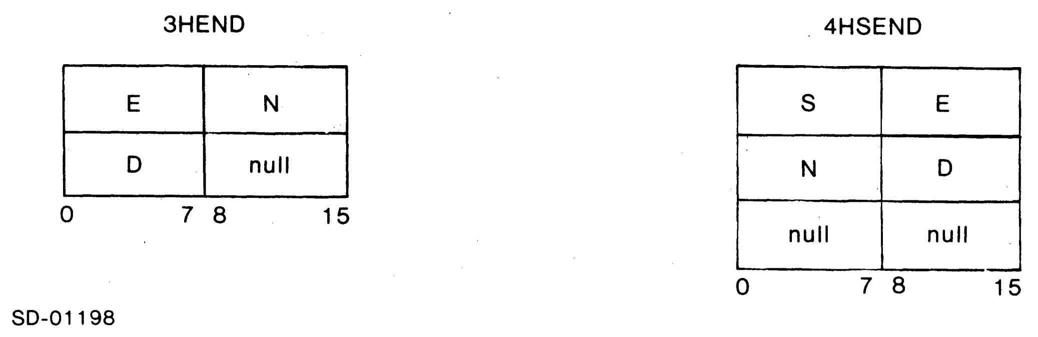


Figure 2-6. Left-justified String Storage

In the case of a right-justified string constant, the compiler starts the storage of the string with a null byte. It terminates a right-justified string containing an even number of characters by a null byte. A right-justified string containing an odd number of characters is followed by two null bytes (see Figure 2-7).

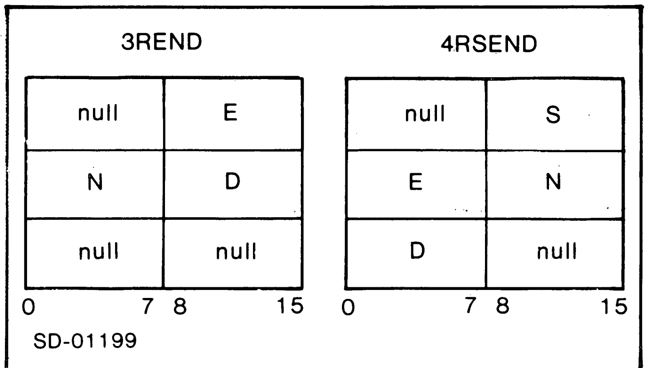


Figure 2-7. Right-justified String Storage

Using Hollerith and String Constants

Hollerith and string constants do not have a data type associated with them; they are called *typeless*. They assume the data type of the context in which they appear. Because of this, you may only use a Hollerith or string constant where you would use other constants if the Hollerith or string constant appears with at least one operand that has a data type and one operator. The two exceptions to this rule are:

1. A typeless constant may be an actual argument in a CALL statement, and

2. A typeless constant may not be a base or an exponent in an exponentiation operation.

If a typeless constant appears as the only entity to the right of an equal sign, it takes on the type of the entity to the left of the equal sign. If it appears with an operator other than an equal sign (not an exponent operator), it takes on the data type of the entity you combine it with. You cannot combine two typeless constants.

FORTRAN 5 does not blank pad a typeless constant when you assign it to a variable that is longer than it is. If the variable's length is shorter than the typeless constant, the compiler assigns the first *n* characters of the constant, where *n* is the number of bytes in the variable.

The following examples show valid and invalid uses of Hollerith and string constants.

VALID	**INVALID**
X = "AB"	X = "AB" + "CD"
A = (C = B + 5HMONTH) + 9	GO TO (10,20), "X"
CALL FUNC1 (X,Y,"ZED")	DO 10 J = "A",1HB,2

Symbolic Names

You identify many FORTRAN 5 components in a program by *symbolic names*. A symbolic name may contain from 1 to 31 letters and digits, and must begin with a letter. FORTRAN 5 ignores any blanks.

The symbolic name that identifies a datum or function may also identify its data type. According to the name rule, the first letter of the name implies its type: the letters I, J, K, L, M, and N imply type integer and any other letter implies type real. You may reconfigure the name rule by using the IMPLICIT statement (see Chapter 7).

You may specify a symbolic name that identifies a variable, array, function subprogram, or a statement function argument in a Type-statement. This statement identifies the symbolic name as either INTEGER, REAL, DOUBLE PRECISION, COMPLEX, DOUBLE PRECISION COMPLEX, or LOGICAL. It either overrides or confirms the symbolic name's implied type (see the section "Type-Statements" in Chapter 7).

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

The name of a function subprogram implies its data type. You may override or confirm the implied data type in a FUNCTION statement, a Type-statement, or an IMPLICIT statement (see Chapter 7).

The type of symbolic name identifying a FORTRAN 5 intrinsic function name is discussed in Chapter 8. A generic function name does not have a predetermined data type; it depends on the type of its argument (see Chapter 8).

The PARAMETER statement allows you to assign a symbolic name to a constant or expression. The form of the constant or expression implies the name's data type.

A symbolic name that identifies a subroutine, common block, task, or overlay has no data type.

Variables

A *variable* is a dynamic entity represented by a symbolic name that has a data type associated with it. You may define and reference a variable. Its value is the value that is currently stored in the storage location it occupies; you may change this value by assigning a new value to the variable.

You may associate two or more variables with each other by associating them with the same storage location. You accomplish this by using a COMMON statement, an EQUIVALENCE statement, and actual and dummy arguments in subprogram references.

Use a DATA statement to assign a value to a variable prior to execution. Use an input or assignment statement to assign a value to a variable during execution.

Data Types of Variables

A variable can have one of several data types: integer, real, double precision, complex, double precision complex, or logical.

The data type of a variable is the type specified for its symbolic name, either implicitly by the name rule, explicitly by a Type-statement, or you may reconfigure the name rule by using an IMPLICIT statement.

Arrays

An *array* is a sequence of data of one or more dimensions that you identify by a symbolic name which has a data type associated with it. Its name may be implicitly typed according to the name rule, explicitly typed by a Type-statement, or you may reconfigure the name rule by using an IMPLICIT statement. You reference each member of the sequence of data, called an *array element,* by appending a qualifier to the array name, called a *subscript.*

An array may have an arbitrary number of dimensions. A column of figures is an example of a one-dimensional array. If you have two or more columns of figures, you have a two-dimensional array; to refer to a specific element of this array you must specify both its row number and its column number. If these columns continue over several pages, you have a three-dimensional array and must specify the row number, the column number, and the page number to refer to a specific element.

You may use the DIMENSION statement, the COMMON statement, or any Type-statement to establish an array. Using one of these statements, you define the name of the array, the number of dimensions in the array, and the number of elements in each dimension. Some examples are:

```
DIMENSION A(10,10)
```

where A is a two-dimensional array of 100 elements.

```
COMMON B(2,5,5)
```

where B is a three-dimensional array of 50 elements.

```
INTEGER C(5,2,2,2)
```

where C is a four-dimensional array of 40 elements.

Subscripts

A *subscript* is a qualifier that you append to an array name. It determines which element of the array you are referencing. The form of a subscript is:

array name (s *[,s]* . . .)

where s is a subscript expression. To refer to an array element, you must specify one subscript expression for each dimension defined for that array. For example, if array XYZ is a three-dimensional array, you must specify all three dimensions in the subscript. A valid subscripted array reference might look like the following:

```
XYZ (2,4,3)
```

A subscript expression can be any valid arithmetic expression. If it is not of type integer, FORTRAN 5 converts the value to type integer (before using it) according to the rules for arithmetic conversion.

During execution, the subscript of an array must not assume a value that is less than the lower bound or greater than the upper bound for that dimension of the array. Chapter 1 in Part 1 of the *FORTRAN 5 Programmer's Guide* discusses use of the /S switch in the FORTRAN 5 command line to check subscript references at runtime for validity. If you do not use the /S switch, and give an array subscript that is out of range, results are undefined and may be fatal.

Array Storage

FORTRAN 5 stores elements of an array in linear sequence. It stores an array with only one dimension so that the first array element occupies the array's first storage location and the last array element occupies the last storage location. Given a multi-dimensional array, FORTRAN 5 stores the array elements such that the leftmost subscript(s) vary most rapidly. We call this the "order of subscript progression." When a nonsubscripted array name appears in a statement that assigns values to that array, FORTRAN 5 assigns the values in the order of subscript progression. Array storage sequence is shown in the following three examples:

1. Elements of an array dimensioned C(15) are stored as:

   ```
   C(1),C(2),...,C(15)
   ```

2. Elements of an array dimensioned A(10,10) are stored as:

   ```
   A(1,1),A(2,1),...,A(9,1),A(10,1),A(1,2),A(2,2),...,
     A(9,10),A(10,10)
   ```

3. Elements of an array dimensioned B(2,3,4) are stored as:

   ```
   B(1,1,1),B(2,1,1),B(1,2,1),B(2,2,1),B(1,3,1),
     B(2,3,1)B(1,1,2),...,B(1,3,4),B(2,3,4)
   ```

Expressions

An *expression* is a combination of operators and operands that represents a single value. The operators specify the computations to perform.

FORTRAN 5 has three classes of expressions: arithmetic, relational, and logical. An arithmetic expression yields a numeric value; both relational and logical expressions yield logical values (.TRUE. or .FALSE.).

Arithmetic Expressions

Arithmetic operators and arithmetic operands form an *arithmetic expression.* An arithmetic operand may be a variable, an array element, a statement function, a function subprogram, or a constant. It specifies the computation to perform using the given operands and produces a numeric value as a result. The FORTRAN 5 operators are shown in Table 2-3.

Table 2-3. FORTRAN 5 Operators.

Operator	Function
**	Exponentiation
*	Multiplication
/	Division
+	Addition and Unary Plus
-	Subtraction and Unary Minus
.AND.	Boolean Conjunction
.OR.	Boolean Disjunction
.XOR.	Boolean Exclusive OR
.NOT.	Boolean Negation
=	Replacement

The operators in Table 2-3 are called binary operators because each one operates on two elements. The + and - symbols are also unary operators when you use them to confirm or change the sign of an arithmetic operand.

The symbol ** means *raising to a power.* For example, A ** B represents A raised to the B power.

You may only use the boolean operators listed in Table 2-3 with integer data; i.e., you may not use them with real, double precision, complex, or double precision complex data. FORTRAN 5 performs boolean operations on a bit-by-bit basis. The operators .AND., .OR., .XOR., and .NOT. are equivalent to the functions IAND, IOR, IXOR, and NOT respectively (see Chapter 8 for information on these functions).

The symbol = means *is replaced by.* For example, A = B represents the value A after FORTRAN 5 has assigned the (possibly converted) value of B to it. Note that the item to the left of the equal sign must be a reference (a variable, array element, FLD function, BYTE function, or a statement function reference that evaluates to one of the above).

An arithmetic operand can have one of the five data types: integer, real, double precision, complex, or double precision complex.

Rules for Evaluating Arithmetic Expressions

You must assign a value to a variable or array element before you can use the variable in an expression.

When you use either plus (+) or minus (-) as a unary operator, the data type of the result is the same as that of the operand.

Data Types of Operands

If an expression contains two operands of the same data type and you specify either +, -, *, or / as an operator, the data type of the result is the same as that of the operands.

An expression may contain operands of differing data types. In an expression using +, -, *, or / as its operator, FORTRAN 5 converts the operand with the lower ranking data type to the higher ranking data type. The result of the evaluated expression has the higher ranking data type. For example, if you combine a noncomplex operand with a complex or double precision complex operand, the noncomplex operand temporarily becomes a complex number with an imaginary part of zero. The exception to the rule is if you combine a double precision operand and a complex operand: the double precision operand becomes a double precision complex number and the resulting data type of the evaluated expression is double precision complex. Figure 2-8 shows the order in which FORTRAN 5 ranks data types.

Type	Rank of Data
Double Precision Complex	Highest
Complex	↓
Double Precision	↓
Real	↓
Integer	Lowest

Figure 2-8. Ranking Data Types

In an expression, base and exponent operands (** operator) with the same data type yield a value of the same data type. Base and exponent operands with differing data types yield a value of the higher ranking data type. There is one exception: raising a complex base to a double precision exponent yields a value that is double precision complex.

You may have operands of differing data types in an expression that contains the replacement operator (=). Upon evaluation, the result has the data type of the operand to the left of the replacement operator. See Chapter 3 for further information.

Tables 2-4, 2-5, and 2-6 show the rules for evaluating mixed data types. The resulting data types are in the shaded areas.

Table 2-4. Evaluating Mixed Data Types for Addition, Subtraction, Multiplication, and Division

(Operators + - * /)		Operand A				
		Integer	Real	Double Precision	Complex	Double Complex
Operand B	Integer	Integer	Real	Double Precision	Complex	Double Complex
	Real	Real	Real	Double Precision	Complex	Double Complex
	Double Precision	Double Precision	Double Precision	Double Precision	Double Complex	Double Complex
	Complex	Complex	Complex	Double Complex	Complex	Double Complex
	Double Complex	Double Complex	Double Complex	Double Complex	Double Complex	Double Complex

Table 2-5. Evaluating Mixed Data Types for Exponentation

(Operator **)		Exponent Operand				
		Integer	Real	Double Precision	Complex	Double Complex
Base Operand	Integer	Integer	Real	Double Precision	Complex	Double Complex
	Real	Real	Real	Double Precision	Complex	Double Complex
	Double Precision	Double Precision	Double Precision	Double Precision	Double Complex	Double Complex
	Complex	Complex	Complex	Double Complex	Complex	Double Complex
	Double Complex	Double Complex	Double Complex	Double Complex	Double Complex	Double Complex

Table 2-6. Evaluating Mixed Data Types for Replacement Operations

(Operator =)		Left-hand Operand				
		Integer	Real	Double Precision	Complex	Double Complex
Right-hand Operand	Integer	Integer	Real	Double Precision	Complex	Double Complex
	Real	Integer	Real	Double Precision	Complex	Double Complex
	Double Precision	Integer	Real	Double Precision	Complex	Double Complex
	Complex	Integer	Real	Double Precision	Complex	Double Complex
	Double Complex	Integer	Real	Double Precision	Complex	Double Complex

Operator Precedence

FORTRAN 5 evaluates arithmetic expressions in an order determined by the precedence of each operator, as shown in Figure 2-9.

Operator	Precedence
**	Highest (evaluated first)
+ - (Unary)	
/ *	
+ - (Binary)	
.NOT.	
.AND.	
.OR. .XOR.	
=	Lowest (evaluated last)

Figure 2-9. Arithmetic Operator Precedence

When operators of equal precedence appear in an expression, the result is as if FORTRAN 5 performs the operations from left to right for +, -, *, and /, and right to left for ** and =.

Because of its optimization capabilities, the actual order in which the FORTRAN 5 compiler evaluates operands and performs operations in an expression is unspecified. The only restrictions on the compiler are:

- It must preserve the order established by explicit parentheses;
- It must ultimately obtain results as if operator performance was as described in the previous paragraph.

In other words, the resulting value of an expression is predictable (subject to range and precision limitations), but the means the compiler will employ to get that result is not.

Another facet to compiler optimization is that FORTRAN 5 need not evaluate any element in an expression which is not necessary to establish the value of that expression. For example, given the expression

```
.TRUE. .OR. (FUNC(X) .EQ. 0)
```

FORTRAN 5 need not evaluate the element (FUNC(X) .EQ. 0) because the ultimate value of this expression is *true* regardless of what (FUNC(X) .EQ. 0) is equal to.

Where appropriate, FORTRAN 5 may revise the order of evaluating certain combinations of elements by applying the associative and commutative laws (subject to explicit parentheses, of course). For example, given the expression A + B, the compiler may evaluate it as A + B or B + A.

You are restricted when specifying a function in an expression in that the function may not alter the value of any other element within the expression. If you should specify the expression X + F(X), where the function F modifies its argument, the value of the expression is undefined. FORTRAN 5 may add the functional value to either the original or modified value of X.

Use of Parentheses

If you want the compiler to treat an expression as an entity, enclose that expression in parentheses. In this manner, you establish the desired interpretation because FORTRAN 5 preserves the integrity of parentheses when evaluating operands and performing operations in an expression. If you nest parenthesized expressions, FORTRAN 5 evaluates the innermost first. Figure 2-10 shows an example of how parentheses affect the result of an evaluated expression.

Expression	Result
8 + 4 * 9 - 6 / 2	41
(8 + 4) * 9 - 6 / 2	105
8 + 4 * (9 - 6 / 2)	32
8 + 4 * ((9 - 6) / 2)	14

Figure 2-10. Operator Precedence Example

If you have more than one expression within a parenthesized group, evaluation takes place according to the normal order of operator precedence.

You should always enclose replacement expressions in parentheses to insure correct evaluation.

Relational and Logical Expressions

Relational Expressions

A relational expression consists of two arithmetic expressions separated by a relational operator. The value of a relational expression is either true or false, depending on whether or not a stated relationship exists. The relational operators are shown in Table 2-7. You must include the delimiting periods as part of each relational operator.

Table 2-7. Relational Operators

Operator	Relationship
.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

Logical Expressions

A logical expression is a combination of logical operators and logical operands. Logical operands are entities that are data typed LOGICAL. A logical expression yields a single logical value, true or false. Table 2-8 lists the logical operators.

Table 2-8. Logical Operators

Operator	Meaning
.AND.	Logical Conjunction. The expression is true if both operands are true. It is false if one or both of the operands is false.
.OR.	Logical Disjunction. The expression is true if either operand, or both, is true. It is false if both operands are false.
.XOR.	Logical Exclusive OR. The expression is true if one operand is true and the other is false. It is false if both operands have the same value.
.NOT.	Logical Negation. The expression is true if the operand is false.

You must include the delimiting periods as part of each logical operator. The .AND, .OR., and .XOR. operators are binary operators; .NOT. is a unary operator. You may use these operators with integer data (see the section "Arithmetic Expressions"). Table 2-9 shows a logical truth table, given the operands Y and Z.

Rules for Evaluating Logical and Relational Expressions

You may combine logical and relational operators within an expression. For example:

```
A .LE. B .AND. D .GT. F
```

The general rules of operator precedence in evaluating relational and logical expressions are the same as for arithmetic expressions: First, FORTRAN 5 evaluates parenthesized expressions as entities, and then proceeds from left to right (given operators of equal precedence).

Table 2-9. Logical Truth Table

Y Operand	Z Operand	.NOT. Y	Y .AND. Z	Y .OR. Z	Y .XOR. Z
.FALSE.	.FALSE.	.TRUE.	.FALSE.	.FALSE.	.FALSE.
.FALSE.	.TRUE.	.TRUE.	.FALSE.	.TRUE.	.TRUE.
.TRUE.	.FALSE.	.FALSE.	.FALSE.	.TRUE.	.TRUE.
.TRUE.	.TRUE.	.FALSE.	.TRUE.	.TRUE.	.FALSE.

FORTRAN 5 evaluates arithmetic expressions (except =) first in accordance with the rules of arithmetic operator precedence, then relational operations, then logical operations and then replacements (=). The precedence of all operators in the FORTRAN 5 system is shown in Figure 2-11.

Operator	Precedence
**	Highest
+ - (Unary)	
* /	
+ - (Binary)	
.GE. .GT. .EQ. .NE. .LT. .LE.	
.NOT.	
.AND.	
.OR. .XOR.	
=	Lowest

Figure 2-11. FORTRAN 5 Operator Precedence

When any two of the operators in Figure 2-11 (except = and **) appear in the same expression and are of equal precedence, FORTRAN 5 evaluates the operands from left to right. Given the symbols = and **, it evaluates the operands from right to left.

Table 2-10 shows examples of logical and relational expression evaluation.

Assume that: A = .TRUE, W =2, X = 4, Y = 6

Table 2-10. Logical and Relational Truth Tables

Expression	Interpretation of Operation(s)	Resulting Value
W .LE. X	true	.TRUE.
W .LT. X .AND. W .LT. Y	true and true	.TRUE.
W .NE. X .AND. .NOT. A	true and false	.FALSE.
.NOT. A .OR. W .EQ. X	false and false	.FALSE.

Function References

A *function reference* is the name of a function followed by a list of actual arguments. After FORTRAN 5 invokes and executes the function, it returns a value to the point of invocation. We discuss function references in detail in Chapter 8.

End of Chapter

Chapter 3
Assignment Statements

An *assignment statement* is a replacement expression; that is, it assigns the resulting value of an expression to a variable, array element, FLD reference, BYTE reference, or a statement function reference that expands to one of the above.

A Hollerith or string constant may appear to the right of the equal sign in any assignment statement (except the ASSIGN statement) and you may assign it to an entity of any data type. FORTRAN 5 assigns the ASCII value of the first *n* characters of such a constant, where *n* is the number of bytes in the entity to the left of the equal sign (2 for integer and logical, 4 for real, 8 for double precision or complex, and 16 for double precision complex).

There are three types of assignment statements:

1. Arithmetic assignment statements
2. Logical assignment statements
3. ASSIGN statements

Arithmetic Assignment Statement

The arithmetic assignment statement assigns the value of the expression on the right side of the equal sign to the entity on the left side of the equal sign. The new value replaces the previous value of the entity, if any.

Format

```
v = e
```

Where:

v is a variable, array element, FLD reference, BYTE reference, or a statement function reference that expands to one of the above.

e is an expression.

Statement Execution

The = symbol means *is replaced by*. For example,

```
A = A + B
```

means to replace the current value of A with the sum of that value and the value of B.

The entity to the left of the equal sign does not need a preassigned value, but you must assign values to each item in the expression (e) prior to execution of the assignment statement.

If the data types of the entity on the left and the expression on the right are the same, then the statement assigns the value directly. If they are different, FORTRAN 5 converts the value of the expression to the data type of the entity on the left before the assignment takes place. Table 3-1 gives a summary of data type conversions for the arithmetic assignment statement.

Unlike other function references, a statement function reference can appear to the left of an equal sign (even if you have already given the statement function definition and/or referenced the function to the right of an equal sign) if the function expands to one of the following:

1. A variable name
2. An array element name
3. A FLD reference
4. A BYTE reference

Examples

```
AVG = (A + B + C) / 3

B = C(I) + (SIN(Y) + 6)

X = "ABCD" + 3 * A - (B / 6)

FLD(I,1,5) = 7

BYTE(X,4) = 43

B(I,J) = A ((I-1) * 10 + J)

ROOT(A,B,C) = (-B + SQRT
  (B ** 2 - 4. * A * C)) / 2. * A)
```

Table 3-1. Assignment Statement Conversion Rules

	EXPRESSION (e)				
VARIABLE (v)	**INTEGER**	**REAL**	**DOUBLE PRECISION**	**COMPLEX**	**DOUBLE PRECISION COMPLEX**
INTEGER	Assign e to v.	Fix e to integer and assign to v.	Fix e to integer and assign to v.	Fix the value of the real part of e and assign to v; imaginary part is not used.	Fix the value of the real part of e and assign to v; imaginary part is not used.
REAL	Float e and assign to v.	Assign e to v.	Assign MS portion of e to v; LS portion of e is truncated.	Assign the real part of e to v; imaginary part is not used.	Assign MS portion of the real part of e to v; LS portion of the real part of e is truncated; imaginary part is not used.
DOUBLE PRECISION	Float e and assign to MS portion of v; LS portion of v is zero.	Assign e to MS portion of v; LS portion of v is zero.	Assign e to v.	Assign real part of e to MS portion of v; LS portion of v is zero; imaginary part is not used.	Assign the real part of e to v.
COMPLEX	Float e and assign to MS portion of v; imaginary part of v is zero.	Append an imaginary part of zero to e and assign to v.	Assign MS portion of e to real part of v; LS portion of e is truncated; imaginary part of v is zero.	Assign e to v.	Assign MS portion of both real and imaginary parts of e to v; LS portions of e are truncated.
DOUBLE PRECISION COMPLEX	Float e and assign to MS portion of v; LS portion and imaginary part of v are zero.	Assign e to MS portion of v; LS portion and imaginary part of v are zero.	Assign e to v; imaginary part of v is zero.	Assign e to MS portions of real and imaginary parts of v; LS portions of real & imaginary parts of v are zero.	Assign e to v.

MS = Most Significant (high-order)
LS = Least Significant (low-order)

Logical Assignment Statement

The logical assignment statement is similar to the arithmetic assignment statement, except that it operates with logical data. The expression to the right of the equal sign evaluates to a logical value (true or false). FORTRAN 5 then assigns this value to the logical entity to the left of the equal sign.

Format

v = e

Where:

v is a logical variable, an array element, or a statement function reference that expands to either one.

e is a logical or relational expression.

Statement Execution

The entity to the left of the equal sign must be of type logical.

All entities of the logical expression must have previously assigned values having one of the numeric data types, a logical data type, or no data type (a Hollerith or string constant). The expression must yield a logical value (true or false).

Examples

```
LOGIC(4) = X .GT. 5. .OR. Y .LT. Z

Y = .TRUE.

L(2) = NAME(2) .EQ. "BILL"
```

ASSIGN Statement

The ASSIGN statement associates a statement label with an integer variable. You may then use the variable in subsequent assigned GO TO statements.

Format

ASSIGN n TO v

Where:

n is the label of an executable statement.

v is an integer variable.

Statement Execution

This statement assigns statement label n to the variable v. The statement label must refer to an executable statement within the same program unit. If the label refers to a nonexecutable statement, FORTRAN 5 signals an error at compile time.

The statement ASSIGN 20 TO J (J gets the statement label 20) is not equivalent to the statement J = 20 (J gets the value 20).

Examples

```
ASSIGN 25 TO J

ASSIGN 100 TO IERR
```

End of Chapter

Chapter 4
Control Statements

FORTRAN 5 executes statements sequentially unless you use a control statement. To change the flow of program logic, use the GO TO, IF, DO, or CONTINUE statements. Use the END, PAUSE, and STOP statements respectively to mark the end of a program, temporarily suspend program execution, or terminate program execution.

GO TO Statements

A GO TO statement transfers control either conditionally or unconditionally to a labeled statement. The three types of GO TO statements are:

1. Unconditional GO TO statement
2. Computed GO TO statement
3. Assigned GO TO statement

Unconditional GO TO

Transfers control unconditionally to a specified statement.

Format

GO TO n

Where:

n is the label of an executable statement.

Statement Execution

Control transfers to the statement indicated by n when FORTRAN 5 executes the GO TO statement. The statement label must refer to an executable statement within the same program unit. If the label refers to a nonexecutable statement, the compiler signals an error.

You must label the next executable statement following an unconditional GO TO statement, or the compiler will signal an error.

Example

```
   GO TO 25
10 R = 2*A
   .
   .
   .
25 I = I+1
```

When FORTRAN 5 executes the above example, the GO TO statement transfers control to the statement labeled 25.

Computed GO TO

Transfers control to one of several specified statements depending on the value of a specified variable.

Format

GO TO (n_1, n_2, ..., n_m) *[,]* v

Where:

n is the label of an executable statement.

v is an integer variable, an array element, or an expression evaluated to an integer.

Statement Execution

Control transfers to n_1, n_2, ..., or n_m depending on whether the current value of v is 1, 2, ..., or m respectively. If the value of v is less than 1 or greater than m, control passes to the next executable statement.

A statement label must refer to an executable statement within the same program unit, or the compiler will signal an error.

The comma separating the right parenthesis from v is optional.

Example

```
GO TO (10,100,50,10,9) K
```

The variable K must be 1, 2, 3, 4, or 5 for a branch to occur. If K is 1, control passes to the statement labeled 10; if K is 2, control passes to the statement labeled 100, etc.

Assigned GO TO

Transfers control to a previously ASSIGNed statement label.

Format

GO TO v *[,] [(n_1, n , [(n_1, n_2, ..., n_m)]*

Where:

v is an integer variable.

n is the label of an executable statement.

Statement Execution

This statement transfers control to the statement whose label you most recently ASSIGNed to the integer variable v (see Chapter 3, the ASSIGN statement).

The comma separating the v and the left parenthesis is optional.

The list of statement labels is also optional. FORTRAN 5 lets you include this list to conform with ANSI FORTRAN standards; the compiler does not check the labels for legality and treats the statement as an unconditional GO TO.

You must label the next executable statement following an assigned GO TO statement, or the compiler will signal an error.

Examples

```
   ASSIGN 5 TO J
   .
   .
   GO TO J
15 CALL SUBR1
   .
5  FLAG=3.

   ASSIGN 5 TO J
   .
   .
   GO TO J(25,16,5,35)
15 CALL SUBR1
   .
5  FLAG=3.
```

In both examples, control transfers to the statement labeled 5. (If we did not include "5" in the statement label list following the statement GO TO J in example 2, FORTRAN 5 would not signal an error and the result would be the same.)

IF Statements

An IF statement performs a conditional transfer based on the value of an expression and on specifications given within the statement itself. There are two types of IF statements:

1. Arithmetic IF Statement
2. Logical IF Statement

Arithmetic IF

Transfers control conditionally to one of three statements based on the value of an arithmetic expression.

Format

IF (e) n_1, n_2, n_3

Where:

e	is an arithmetic expression.
n	is the label of an executable statement.

Statement Execution

Upon execution, FORTRAN 5 evaluates the arithmetic expression. The result determines where to pass control: if the result is less than zero, control passes to n_1; if the result is equal to zero, control passes to n_2; if the result is greater than zero, control passes to n_3.

You must specify all three statement labels. They do not have to refer to three different statements, but they must refer to executable statements within the same program unit. If not, the compiler signals an error.

Because an arithmetic IF statement tests for exact zero, do not use it to test for real zero (as opposed to integer zero). Real zero is often approximate.

Examples

```
IF (K(I,J) - L) 10,4,30

IF (IQ * IR) 5,5,2

IF (ITEST1**4-ITEST2/2)10,15,15
```

Logical IF

Conditionally executes and transfers control to a statement based on the value of a logical expression.

Format

IF (le) s

Where:

le	is a logical expression.
s	is any executable statement except a DO statement.

Statement Execution

Upon execution, FORTRAN 5 evaluates the logical expression. If the value of the expression is .TRUE., FORTRAN 5 executes statement s. Control then passes to the first executable statement following statement s unless statement s transfers control elsewhere. If the value of the expression is .FALSE., statement s is bypassed and control passes to the next executable statement.

Examples

```
IF (A .GT. 3. .AND. B .EQ. .1) F = SIN (R)

IF (1 .GT. 0) GO TO 25
```

DO

Executes a group of statements one or more times.

Format

DO n v = p_1, p_2 *[,p_3]*

Where:

n is the label of an executable statement.

v is a numeric control variable.

p_1 is the initial parameter.

p_2 is the terminal parameter.

p_3 is the incrementation parameter.

Execution of the DO

The DO statement sets up a loop which begins at the DO statement and ends at the statement labeled n. We refer to this set of statements as "the range of the DO". FORTRAN 5 always executes the range of the DO at least once.

The action taken by the execution of a DO statement is as follows:

- FORTRAN 5 assigns the value of the initial parameter to the control variable (this value must be less than or equal to the terminal parameter).
- The range of the DO is executed.
- Unless a statement transfers control out of the loop, FORTRAN 5 executes the terminal statement and increments the control variable by the value of the incrementation parameter.
- FORTRAN 5 then tests the value of the control variable. If it is less than or equal to the terminal parameter, the cycle is repeated. If the value of the control variable is greater than the value of the associated terminal parameter, control passes to the first executable statement following the DO loop's terminal statement.

If the final value of v is greater than or equal to 32767, and the control variable is of type integer, the DO loop executes forever.

The control variable may be of type integer, real, or double precision. The parameters (p_1, p_2, and p_3) may be any expressions of type integer, real, or double precision.

The increment value may be positive or negative. An increment value of zero causes the loop to execute forever. If you do not specify the increment value, it is equal to 1.

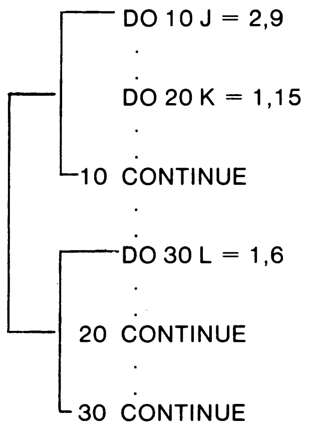
The DO loop terminates for any of the following reasons:

1. The value of the control variable is greater than the value of its final parameter if the increment value is positive, or the value of the control variable is less than the final parameter if the increment value is negative.
2. If you specify an initial value for v that is larger than the final value of v and give a positive increment value (FORTRAN 5 executes the loop once because it tests the control variable at the end of the loop).
3. You transfer control out of the loop.

Upon termination of a DO loop, control passes to the first executable statement following the terminating statement of the loop.

Nested DO Loops

You can nest DO loops to any depth. The range of an inner DO loop cannot extend beyond the range of the outer DO loop, but both may share the same terminal statement. For example,



```
     DO 10 J = 2,9
     .
     DO 20 K = 1,15
     .
     .
  10 CONTINUE
     .
     DO 30 L = 1,6
     .
     .
  20 CONTINUE
     .
     .
  30 CONTINUE
```

is incorrectly nested.

One way you can correctly nest these statements is:

```
      DO 10 J = 2,9
      .
      DO 20 K = 1,15
      .
20    CONTINUE
      .
      DO 30 L = 1,6
      .
30    CONTINUE
      .
10    CONTINUE
```

Two loops may use the same statement as their terminal statement if both loops execute in normal sequence. If, however, a statement transfers control out of one loop into another without completing the first loop, the results will be undefined. For example,

```
      DO 10 K=1,20
      I=J+L
      IF (I.LT.0)GO TO 10
      DO 10 M=1,10
      I=ABS(I)+1
      J=SQRT(A+B)-I**4
10    CONTINUE
```

In this example, the compiler essentially creates two CONTINUE statements, lets say 10a and 10b, where 10a is the terminal statement for the inner loop and 10b is the terminal statement for the outer loop. If the statement IF (I.LT.0) GO TO 10 transfers control to statement 10 on the first pass, FORTRAN 5 will attempt to execute the *part* of the CONTINUE statement that terminates the inner loop (10a). The results will be undefined.

In a nested DO, the control variable of the innermost loop varies most rapidly and the control variable of the outermost loop varies least rapidly. In an array, the first subscript bound varies most rapidly and the last subscript bound varies least rapidly. By matching the innermost control variable to the first subscript bound of the array, etc., you significantly optimize the code and reduce execution time of the loop. Because of the way FORTRAN 5 stores array elements, this way of coding the loop permits linear access to the array elements. The savings may be as much as 500 percent.

For example:

Most Efficient Code	**Least Efficient Code**
DO 5 K = 1,10 DO 5 J = 1,10 DO 5 I = 1,10 5 A(I,J,K) = A(I,J,K)*I/J+K	DO 5 I = 1,10 DO 5 J = 1,10 DO 5 K = 1,10 5 A(I,J,K) = A(I,J,K)*I/J+K

Extended Range of the DO Loop

You can extend the range of a DO loop to include additional statements or program units. If there are nested DO loops and you extend the range of one of the loops, you must eventually transfer control back into this loop.

The extended range of the DO loop includes all statements executed outside the loop, including the transfer statement. For example:

```
      SUM = 0
      DO 30 I = 1,10
      DO 25 J = 1,20
      SUM = SUM + MATRIX (J,I)
      IF (SUM.GT.TOTAL) GO TO 50
25    CONTINUE
      .
      .
30    CONTINUE
      .
      .
50    SUBT = 0 }
      .        }  Extended Range
      .        }
      GO TO 25 }
```

We discourage using extended range DO loops, as it may incur a loss of program efficiency.

DO Loop Restrictions

1. You may not transfer control into the range of a DO from elsewhere in the program unit, unless the statement is part of the loop's extended range (assuming the DO loop has already passed control to this statement).

2. You may not terminate the range of a DO with another DO statement; however, you may terminate it with a GO TO or IF statement.

3. You may not redefine the control variable, initial value, final value, or increment within the range of a DO loop.

CONTINUE

Provides a place for a statement label.

Format

CONTINUE

Statement Execution

The execution of a CONTINUE statement has no effect. It is most frequently the last statement in the range of a DO loop. This makes your program more readable and helps you to avoid using a prohibited control statement (e.g., a DO statement) as the terminal statement of a loop. You must label a CONTINUE statement when using it for this purpose. Otherwise, you do not need to label a CONTINUE statement and it may appear anywhere in your program.

Example

```
    S = 0
    DO 15 I = 1,N
    S = S + C(I) * B(I)
15  CONTINUE
```

is equivalent to:

```
    S = 0
    DO 15 I = 1,N
15  S = S + C(I) * B(I)
```

PAUSE

Temporarily suspends program execution, waiting for operator intervention.

Format

PAUSE *[s]*

Where:

s is a string of ASCII characters.

Statement Execution

FORTRAN 5 displays the PAUSE message on the terminal along with a text string, if you specify one, and execution halts. Press any console key and execution resumes with the first executable statement following the PAUSE statement.

Because FORTRAN 5 eliminates blanks when translating the program, enclose the text string in quotation marks. For example, FORTRAN 5 outputs the statement PAUSE TWO□TIMES as PAUSE TWOTIMES. Single or double quotation marks will preserve all blanks.

Example

```
PAUSE "ACTIVATE□PRINTER"
```

STOP

Causes unconditional termination of program execution.

Format

STOP *[s]*

Where:

s is a string of ASCII characters.

Statement Execution

FORTRAN 5 displays the STOP message on the terminal along with a text string(s), if you specify one, and then execution terminates. Control returns to the operating system, unless you called in the program with a call to FSWAP or SWAP, in which case control returns to the first executable statement following the CALL FSWAP or CALL SWAP in the program unit containing the call. (See the *FORTRAN 5 Programmer's Guide* for more information on these runtime routines.)

Because FORTRAN 5 eliminates blanks when translating the program, enclose the text string in quotation marks. For example, FORTRAN 5 outputs the statement STOP□AT□THIS□POINT as STOPATTHISPOINT. Single or double quotation marks will preserve all blanks.

Example

```
STOP 'LABEL□70'
```

END

Marks the end of the program unit.

Format

END

Statement Execution

The END statement must be the last physical statement in a program unit. The letters E, N, and D must be on the same line, in that order, and must start on or after character position 7. You must end this statement with a new-line character.

The END statement does not directly terminate the program unit; it merely marks its end (see the RETURN and STOP statements for information on program termination).

You may label an END statement. For example, if a DO loop is the last set of instructions in a program unit, and you label the END statement, it acts as a CONTINUE statement as well as marking the end of the program.

End of Chapter

Chapter 5
Input/Output Statements

Input/Output statements are executable statements that transfer data between internal storage and an input/output unit. The FORTRAN 5 input/output statements discussed later on in this chapter are ACCEPT, BACKSPACE, CLOSE, DECODE, DELETE, ENCODE, ENDFILE, OPEN, PRINT, PUNCH, READ, READ BINARY, READ FREE, RENAME, REWIND, TYPE, WRITE, WRITE BINARY, and WRITE FREE.

Input/Output Unit Numbers

An *input/output unit* (or *unit*) refers to either a device (such as a card reader or line printer) or a file (a named collection of data stored by some device). For the purposes of this discussion, the two are indistinguishable.

An I/O statement specifies an input/output unit by a number. You may associate this *unit number* with a unit by using an OPEN statement. The FORTRAN 5 system initially associates certain numbers with specific files or devices. You may override these associations in an OPEN statement (discussed in more detail later on in this chapter), or you may define your own file preconnections (see Part I, Chapter 1 in the *FORTRAN 5 Programmer's Guide*).

The execution of a CLOSE or ENDFILE statement terminates the association between the unit number and the unit, at which time you may associate the number with a different unit.

The following example shows the two ways in which you may represent a unit number:

1.
```
OPEN 6,"FILENM"
READ (6) XYZ
CLOSE 6
```

2.
```
J = 6
OPEN J, "FILENM"
READ (J) XYZ
CLOSE J
```

Format Identifiers

Data transfers between internal storage and an input/output unit may occur under the control of a format specification. A *format specification* defines the structure of the data and the nature of the manipulations performed on the data during the course of the I/O transfer.

A formatted I/O statement may contain a format identifier that specifies either the statement label of a FORMAT statement or the name of an array that contains a variable format specification.

The variable format specification must include left and right parentheses. It must not include the word FORMAT or the statement label.

You reference the array containing the variable format specification by specifying the array name in place of the FORMAT statement label in the associated I/O statement. The array name specifies the location of the first word of the format information and may appear with or without a subscript.

The following is an example of the two ways you may write a format identifier. (See Chapter 6 for more information on the FORMAT statement.)

The following is a variable format specification:

```
(E15.7,I6,2G16.6,I9)
```

You can punch this information in an input card and read it by the following statements:

```
   DIMENSION JARRAY (10)
   READ 9,(JARRAY(I), I = 1,10)
10 FORMAT (10A2)
```

FORTRAN 5 places the elements of the input card in storage as follows:

```
JARRAY(1):   (E
JARRAY(2):   15
JARRAY(3):   .7
.
.
.
JARRAY(9):   ,I
JARRAY(10):  9)
```

The following output statement references the above variable format specification:

```
PRINT JARRAY,X,J,P,D,K
```

The above example produces the same results as the statements:

```
    PRINT 20,X,J,P,D,K
20  FORMAT (E15.7,I6,2G16.6,I9)
```

Input/Output Records

I/O statements transmit data in terms of records. A *record* is either a sequence of characters or a sequence of bytes, depending on whether the given I/O statement interprets it as a formatted (ASCII) or an unformatted (binary) record. A *formatted record* ends with a line terminating character, and is called a line. An *unformatted record* is a stream of bytes; it does not end with a line terminating character.

FORTRAN 5 numbers the first record of any file 1.

File Access

In FORTRAN 5, you may access a file sequentially, by lines, as a stream of characters, or by record number.

Sequential Access

When performing a *sequential* read or write, the record accessed is the record immediately following the record most recently accessed. The records are either all formatted or all unformatted. Each record is the same length, specified by the *LEN=* option when you OPEN the file.

Prior to data transfer, FORTRAN 5 positions the file at the initial point of the next record. After data transfer, the file is positioned at the terminal point of the last record read or written. That record becomes the preceding record.

Line Access

A *line file* consists of varying-length lines of formatted output written according to a format specification or list-directed (free-formatted) formatting.

Stream Access

The order of the records in a *stream file* is the order in which they were written. The records are all unformatted.

Prior to data transfer, FORTRAN 5 does not change the position of the file. After data transfer, the file is positioned after the last value read or written and the current record remains the current record.

Direct Access

The order of the records in a *direct access file* is the order of their record numbers. A record number, which must be a positive integer, uniquely identifies a record. This number is specified when the record is written. Records are either all formatted or all unformatted, and all have the same length.

FORTRAN 5 does not distinguish between sequential and direct access, but it does distinguish between sequential and direct access I/O statements. An I/O statement employs direct access if the file upon which it acts was OPENed with the *REC=nn* option; otherwise, the access is sequential.

Prior to data transfer, FORTRAN 5 positions the file at the initial point of the record specified by the record number. After data transfer, the file is positioned at the terminal point of the last record read or written. That record becomes the preceding record.

Data Transfers

You specify data transfers between records and entities by using the list in an I/O statement. FORTRAN 5 processes the entities of the list in the order in which you give them.

FORTRAN 5 distinguishes three types of data transfers: formatted (or edit-directed) transfers, free-formatted (or list-directed) transfers and unformatted (or binary) transfers.

During formatted data transfers, FORTRAN 5 transfers data with editing between the entities specified by the I/O list and the file. The associated format specification performs editing as required.

During free-formatted data transfers, FORTRAN 5 transfers data with editing between the current record of the external file and the entities specified by the I/O list. FORTRAN 5 performs editing according to the data types of the entities in the I/O list. In addition, free-formatted I/O permits conversational I/O from a terminal (the ACCEPT and TYPE statements).

During unformatted data transfers, FORTRAN 5 transfers binary data without editing between the current record of the external file and the entities specified by the I/O list.

Types of File Organization

As mentioned previously, I/O statements transmit data in terms of records. A record's characteristics, though, vary from file to file and are determined either by an I/O statement's interpretation of the file or by the manner in which you OPEN the file.

A *line-oriented file* contains formatted records, each one being a sequence of characters delimited by a new-line character. FORTRAN 5 associates a specific length with each record. A line-oriented file can also be a *print-oriented file* if you prepare it for printing (OPEN it with the "P" attribute). In this case, FORTRAN 5 does not print the first character of each record. This character determines vertical carriage control. For example, if the first character of a record is 0, FORTRAN 5 prints the record after a double line feed. See the section "Carriage Control" in Chapter 6 for further details.

A *stream-oriented file* contains records that are all unformatted. Each record is a sequence of bytes and has no structure associated with it; i.e., the length of each record is dynamic.

A *record-oriented file* contains formatted or unformatted records. Each record has a fixed length which is determined when you OPEN the file.

Table 5-1 illustrates the relationship between the type of data transfer, the I/O statement, and the kind of file access that is allowed.

Input/Output Lists

An I/O list names the data to be sent or received by the I/O statement containing the list. An input list specifies the names of the variables, arrays, and array elements to which you assign values on input. An output list specifies the references to variables, arrays, array elements, constants, and expressions whose values you want to transmit.

An I/O list can contain simple lists and DO-implied lists. FORTRAN 5 treats the entities of an I/O statement list in the order in which they appear, from left to right.

Simple Lists

A *simple input list* may consist of a variable, an array, an array element, or a combination of these. A *simple output list* may consist of the same entities as a simple input list, but also permits constants and expressions.

If you specify an unsubscripted array name in a READ statement's input list, FORTRAN 5 inputs the data needed to fill all elements of the array. Data transmission begins with the first array element and continues in the order of subscript progression (the leftmost subscript varies most rapidly). For example, in the statement READ (11) ARAY1, where you define the array as ARAY1(3,3), the READ statement assigns values from the input record(s) to ARAY1(1,1), ARAY1(2,1), ARAY1(3,1), ARAY1(1,2), etc., up to ARAY1(3,3) inclusive.

Table 5-1. Data Transfers

Type of Data Transfer	Input/Output Statement	External File Organization			
		Line	Print	Stream	Record
Formatted with format specification	READ/WRITE	●	●		●
Free-Formatted	READ FREE/WRITE FREE	●	●		●
	ACCEPT/TYPE	●			
Unformatted	READ BINARY/WRITE BINARY READ/WRITE			●	●

DO-implied Lists

An I/O list may include a DO-implied list to specify iteration within the list, to transmit only part of an array, or to transmit elements of an array in an order differing from normal subscript progression. A DO-implied list is a list followed by a comma and a DO-implied specification, all enclosed in parentheses. The format is:

(list, i = m_1, m_2 *[,m_3]*)

Where:

list	is a simple list.
i	is the control variable.
m_1	is the initial value.
m_2	is the final value.
m_3	is the increment of i; if not given, the increment is 1.

The range of a DO-implied specification is the list. For input lists, i, m_1, m_2, and m_3 may appear within that range only in subscript expressions and may be any type of variable or expression. FORTRAN 5 converts it to a type integer if necessary upon evaluation.

An example of a DO-implied list is:

```
READ (13,5) G, B(3), C, (D(I), I = 1,3)
```

where the statement might read a new value for the variable G, the third element of array B, the entire array C, and the first three elements of array D.

You may nest DO-implied lists to any depth. For example, you may write the list

```
A(1,1),A(1,2),A(1,3),A(2,1),A(2,2),A(2,3)
```

in the alternate form:

```
((A(I,J),J = 1,3),I = 1,2)
```

For functional purposes, you may enclose any portion of an I/O list in parentheses except the loop specification of a DO-implied loop.

Table 5-2 shows the actions taken by I/O statements if you do not specify an I/O list.

Table 5-2. I/O Statements with No I/O Lists

Format	Statement	Action
Free-Formatted	READ	Skip a record.
	WRITE	Output a blank record.
Unformatted (Binary)	READ	Skip a record.
	WRITE	Write record of nulls if LEN= option is given; otherwise ignore statement.
Formatted	READ WRITE	Perform any format editing.

Examples

(Examples with the same numbers are equivalent.)

```
1a. DIMENSION A(2,3)
    READ (11,5) A
1b. READ (11,5),A(1,1),A(2,1),A(1,2),A(2,2),A(1,3),A(2,3)

2a. WRITE (12) I,A,A(I,J)
2b. WRITE (12) (I,A),(A(I,J))

3a. READ (13,20) A,B,(C(I),I=1,3)
3b. READ (13,20) A,B,C(1),C(2),C(3)

4a. WRITE (10,20) (A,B,C,I=1,2)
4b. WRITE (10,20) A,B,C,A,B,C
```

Options List

A READ or WRITE statement can contain an options list consisting of any or all of the following:

1. *ERR = label,* where *label* is the label of a statement to which control passes if an error occurs. If you do not give the *END* = option, the *ERR* = option applies if an end-of-file or out-of-space condition occurs.

2. *END = label,* where *label* is the label of a statement to which control passes if an end-of-file or out-of-space condition occurs.

3. *REC = expression,* where FORTRAN 5 evaluates *expression* as an integer that gives the number of the record at which reading or writing will begin. Use this option only with disk files which have a record length *(LEN =)* associated with them when they are OPENed. The type of file (C, R, S) does not affect this option in any way.

FORTRAN 5 Input/Output Statements

The following section discusses all FORTRAN 5 I/O statements as follows:

I. Formatted (edit-directed) I/O: READ, WRITE, ENCODE DECODE, and Simple Formatted READ, PRINT, and PUNCH.

II. Free-formatted (list-directed) I/O: READ FREE, WRITE FREE, ACCEPT, and TYPE.

III. Unformatted (binary) I/O: READ BINARY and WRITE BINARY.

IV. Auxiliary I/O: OPEN, CLOSE, ENDFILE, DELETE, RENAME, REWIND, and BACKSPACE.

I. Formatted Input/Output

READ

Transfers data from a file to internal storage according to specifications in the corresponding FORMAT statement.

Format

READ (unit,format *[,options]*) *[list]*

Where:

unit	is an integer that specifies the FORTRAN 5 unit number of the file you want to read.
format	is the number of a FORMAT statement or the name of an array containing the formatting specification.
options	is a list of one or more options (see the section "Options List").
list	is an input list.

Statement Execution

FORTRAN 5 interprets and converts data to an internal representation according to the corresponding FORMAT statement, which you may place anywhere in the program.

The statement READ format *[,list]* assumes default card reader input and is equivalent to READ (9,format) *[list]* (unless you change the preconnected unit number as described in Part I, Chapter 1 of the *FORTRAN 5 Programmer's Guide)*.

Alternate Form

READ INPUT TAPE (unit,format) *[list]*

NOTE: The options list is not permitted in this construct.

Examples

```
READ (11,20) (C(I,I), I = 1,41)

READ (13,25) A(I,J)

READ (11,ARRAY) A

N = 9
.
.
READ (N,45) A,B,I,K
```

WRITE

Writes data from internal storage to a file or device specified by a unit number.

Format

WRITE (unit,format *[,options]*) *[list]*

Where:

unit	is an integer that specifies the FORTRAN 5 unit number of the file or device to which you want to write.
format	is the number of a FORMAT statement or the name of an array containing the format specification.
options	is a list of one or more options (see the section "Options List").
list	is an output list.

Statement Execution

FORTRAN 5 converts internal data to ASCII. It then displays or stores the data on output according to the corresponding FORMAT statement, which you may place anywhere in the program.

The statement PUNCH format *[,list]* assumes default punched output and is equivalent to WRITE (14,format) *[list]*. The statement PRINT format *[,list]* assumes default line printer output and is equivalent to WRITE (12,format) *[list]*. (You may alter the default for these two conditions by changing the preconnected unit numbers as described in Part I, Chapter 1 of the *FORTRAN 5 Programmer's Guide.)*

Alternate Form

WRITE OUTPUT TAPE (unit,format) *[list]*

NOTE: The options list is not permitted in this construct.

Examples

```
WRITE (12,90) A,B,I

WRITE (10,95) (A(I),B(I+1),C(2A1),D(I),I = 1,2)

WRITE (12,ARRAY) B
```

ENCODE

Performs data transfers strictly between variables or arrays internal to your program according to a format specification, generating object code similar to that generated for a formatted WRITE.

Format

ENCODE (array,format *[,ERR=label]*) *[list]*

Where:

array	is the name of the array that receives the contents of *list*.
format	is the number of a FORMAT statement or the name of an array containing the format specification.
label	is the label of a statement to which control passes if an error occurs.
list	is a list of names of variables whose values you want to transfer to elements of array.

Statement Execution

FORTRAN 5 stores the ASCII value of the elements in *list* in array beginning with the first element.

Unlike conventional formatted I/O statements, data transfers take place entirely between variables and arrays within the program.

Example

```
ENCODE (MAT,100) A,B,C(2,1)
```

DECODE

Performs data transfers according to a format specification, generating object code similar to that generated for a formatted READ.

Format

DECODE (array, format *[,ERR=label]*) *[list]*

Where:

array	is the name of the array from which you want data transferred to the variables given in *list.*
format	is the number of the FORMAT statement or the name of an array containing the format specification.
label	is the label of a statement to which control passes if an error occurs.
list	is a list of names of variables to which you want successive data from the array transferred.

Statement Execution

FORTRAN 5 interprets and converts data in array according to the format specification. It then assigns the data to the elements in *list.* FORTRAN 5 transfers elements of the array in the order of subscript progression.

Unlike conventional formatted I/O statements, data transfers take place entirely between variables and arrays within the program.

Example

```
DECODE (ARRY,FOR1) X,Y,Z
```

Simple Formatted Input/Output

READ, PRINT, PUNCH

Transfer data between internal storage and a default device.

Format

type format *[,list]*

Where:

type	is READ, PRINT, or PUNCH.
format	is the number of a FORMAT statement or name of an array containing the format specification.
list	is an input or output list.

Statement Execution

Data conversion occurs between internal storage representations and the ASCII set, according to the corresponding FORMAT statement.

The three statement types are shown below along with equivalent I/O statements:

READ format *[,list]*
READ (9,format) *[list]*

PRINT format *[,list]*
WRITE (12,format) *[list]*

PUNCH format *[,list]*
WRITE (14,format) *[list]*

Unless you assign channels 9, 12, or 14 to other files using an OPEN statement or the preconnection mechanism, the default devices for this form of READ, PRINT, and PUNCH are the card reader, the line printer, and the paper tape punch, respectively.

In addition, the preconnection mechanism allows you to alter the unit numbers used by the aforementioned statements. (For more information on the preconnection mechanism, see Part I, Chapter 1 of the *FORTRAN 5 Programmer's Guide*.)

II. Free-Formatted Input/Output

READ FREE

Provides a standard formatting without programmer intervention.

Format

READ FREE (unit *[,options]*) *[list]*

Where:

unit	is an integer that specifies the FORTRAN 5 unit number of the file you want to read.
options	is a list of one or more options (see the section "Options List").
list	is an input list.

Statement Execution

Data are externally recognizable numbers in ASCII code that FORTRAN 5 reads and converts to their internal computer representation.

On input, you must separate entities of *list* with a comma, a blank space, or an end-of-record indicator (pressing return/new line, null, or form feed). Two or more separators between values have the same effect as a single separator.

FORTRAN 5 interprets a number containing a blank as two separate numbers, except when expecting additional digits, such as after a sign or following the letter E or D.

Examples

```
READ FREE (11) A,B,C

READ FREE (13) (A(I),R(I),I = 1,3)
```

WRITE FREE

Provides a standard formatting without programmer intervention.

Format

WRITE FREE (unit *[,options]*) *[list]*

Where:

unit	is an integer that specifies the FORTRAN 5 unit number of the file or device to which you want to write.
options	is a list of one or more options (see the section "Options List").
list	is an output list.

Statement Execution

FORTRAN 5 converts data according to a standard field format, the length of which depends on the data type.

FORTRAN 5 applies storage limits to free-formatted ASCII outputs as follows:

integer datum	8 characters
single precision datum	16 characters
double precision datum	24 characters
single precision complex datum	32 characters
double precision complex datum	48 characters

FORTRAN 5 formats the fields for output as follows:

1. The leftmost character position is left blank as a delimiter between numbers.
2. A sign occupies one character position, but if no sign exists, a digit can occupy the position.
3. FORTRAN 5 formats real numbers according to G-format conversion, i.e., using either E format or F format depending on the magnitude of the number. If the datum is in E format, the conversion formula used is 1PEw.d (the datum will have one digit preceding the decimal point).

Examples

```
WRITE FREE (12) A,B,C

WRITE FREE (10) (B(K), K = 2,5)
```

ACCEPT

Allows input of data from the input console. In addition, you may specify certain items to be output to the console for use as prompting information.

Format

ACCEPT list

Where:

list is a list of variables, arrays, and/or array elements whose values are input from the console when the statement is executed, and constants and/or expressions whose values are output at the console when the statement is executed.

Statement Execution

The ACCEPT statement opens both the current input and output consoles by default (unless you change the preconnected unit number as described in Part I, Chapter 1 of the *FORTRAN 5 Programmer's Guide*). You cannot use ACCEPT with a file prepared for printing (OPENed with the "P" attribute) or a record-oriented file.

When the console signals a prompt, separate your input entries by commas or new-line characters. FORTRAN 5 formats the values the same way as free-formatted READ statements do.

FORTRAN 5 converts values to the data type of the internal variable where required. Upon statement execution, FORTRAN 5 simply types out string constants or expressions appearing in the ACCEPT statement. If you want to write a variable rather than read it, you must prefix that variable with a unary plus sign (+).

Unlike the TYPE statement, the ACCEPT statement does not output a new-line character when the list is exhausted. When using this statement for both input and output, enter all appropriate input items, then terminate the line with a new-line character in order to force output.

Example

Code

```
ACCEPT "ENTER□A:□□", A, "ENTER□B:□□", B,
1"SUM□=", A+B, "<NL>"
```

Possible Result

```
ENTER □ A: □□1.2)

ENTER □ B: □□3.4)

SUM □= □□□□□4.60000□□□□)
```

See also the sample program in the next section, "TYPE".

TYPE

Allows interaction between you and your program using the console for output.

Format

TYPE list

Where:

list is a list of variables, arrays, array elements, expressions, and/or constants for which you want values output to the console upon statement execution.

Statement Execution

The TYPE statement opens the output console by default (unless you change the preconnected unit number as described in Part I, Chapter 1 of the *FORTRAN 5 Programmer's Guide*). You cannot use TYPE with a file prepared for printing (OPENed with the "P" attribute) or a record-oriented file.

FORTRAN 5 outputs the contents of variables in the list in the form of free-formatted I/O. Upon execution, the TYPE statement outputs a new-line character when the list is exhausted.

The list of a TYPE statement follows the same rules for output lists discussed in the previous section, "Input/Output Lists".

Example

TYPE "DIMENSIONS ARE:",A,B,I

Sample Program

```
      DIMENSION RARRAY (2000)
      COMMON RARRAY,AUTOC,SD
100   ACCEPT "ARRAY SIZE = ",IAS
     1"INITIAL RANDOM NUMBER = ",RNI
      IF (IAS-2000) 130,130,120
120   TYPE "ARRAY SIZE MAX IS 2000"
      GO TO 100
130   CALL XRANDOM (RN1,IAS,RARRAY)
      CALL CORRELATE (IAS)
      TYPE "AUTOCORRELATION = ",AUTOC
     1"<15>STANDARD DEVIATION = ",SD
      PAUSE
      GO TO 100
      END
```

Possible Result

```
ARRAYSIZE = 500)

INITIAL RANDOM NUMBER = .93826)

AUTOCORRELATION = □□□□□□.73152E-□1)
STANDARD DEVIATION = □□□□□.201552E□□0)
PAUSE
```

III. Unformatted Input/Output

READ BINARY

Transfers a single data record from a file to internal storage with no interpretation.

Format

READ BINARY (unit *[,options]*) *[list]*

Where:

unit	is an integer that specifies the FORTRAN 5 unit number of the file you want to read.
options	is a list of one or more options (see the section "Options List").
list	is an input list.

Statement Execution

If the given file is a record-oriented file, FORTRAN 5 always reads a record's worth of characters. If the file is a stream-oriented file, the size of the record transferred depends on the input list you give in the READ BINARY statement.

Alternate Form

READ TAPE (unit *[,options]*) *[list]*

Example

```
READ BINARY (5) I,J
```

WRITE BINARY

Transfers a single data record from internal storage to a file with no interpretation.

Format

WRITE BINARY (unit *[,options]*) *[list]*

Where:

unit	is an integer that specifies the FORTRAN 5 unit number of the file or device to which you want to write.
options	is a list of one or more options (see the section "Options List").
list	is an input list.

Statement Execution

If the given file is a record-oriented file, FORTRAN 5 always writes a record's worth of characters. If the file is a stream-oriented file, the size of the record transferred depends on the output list you give in the WRITE BINARY statement.

Alternate Form

WRITE TAPE (unit *[,options]*) *[list]*

Example

```
WRITE BINARY (4) A,B
```

IV. Auxiliary Input/Output Statements

OPEN

Assigns a unit number to a file and creates the file, if necessary, according to specifications given.

Format

OPEN unit,name *[,options]*

Where:

unit	is an integer that specifies the FORTRAN 5 unit number (between 0-63 decimal) you want to assign to the file being opened.
name	is a string constant or array name.
options	is a list of one or more options (details given below).

Statement Execution

Upon execution of the OPEN statement, the operating system allocates an available channel to the file. If you open a file and specify no options, the file is line-oriented by default and has the standard default line length associated with the operating system (see the reference manual appropriate to your operating system).

The options may appear in any order, separated by commas, consisting of the following:

1. *ATT=string,* where *string* is a string constant or array name enclosed in quotation marks. The contents of the string must be one or more of the following letters and no more than one letter from each group. There are no order restrictions.

   a. Type of file to create if creation is necessary (default file type is "R"):

C	contiguous file
R	random file
S	sequential file

   b. Type of opening to be performed:

I	Input (READ) only
O	Output (WRITE) only
A	Appending
E	Exclusive access to user who opened the file

   If you do not specify an opening attribute, the use of the file determines the type of opening. However, certain files have predetermined characteristics, e.g., the line printer cannot read. If you do not specify the "A" attribute, writing to a file overwrites the previous contents of the file.

   The term *exclusive access* has different meanings for different operating systems. See the reference manual appropriate to your operating system.

   c. Device specifications:

P	Prepares the file for printing; FORTRAN 5 will use the first character of each record for carriage control on formatted I/O. FORTRAN 5 does not output this character.
M	Overrides all default device attributes provided automatically by the system; use this attribute with caution.
L	Designates the file to be line-oriented rather than record-oriented (see *LEN=* option for further explanation).
B	Provides blank padding when the characters run out for formatted READ statements that read more characters than exist in an input record.

2. *LEN=n,* where *n* is an integer expression that represents the number of bytes (characters) in each record.

Table 5-3 shows how the options of the OPEN statement affect data transfers.

Table 5-3. OPEN Statement Options

Transfer	LEN=n Given	ATT="L" Given	Output Record Padding	Description
Formatted	Yes	Yes	No	Line-oriented; transfer up to *n* characters per line. No spilling.
		No	Blanks	Record-oriented; transfer exactly *n* characters per record. No spilling.
	No	Yes	No	Line-oriented; transfer characters up to the system-default line length according to I/O list. No spilling.
		No	No	Line-oriented; transfer characters up to the system-default line length according to I/O list. No spilling.
Unformatted (Binary)	Yes	Yes	No	Stream-oriented; transfer bytes according to I/O list (no maximum).
		No	Nulls	Record-oriented; transfer exactly *n* characters per record.
	No	Yes	No	Stream-oriented; transfer bytes according to I/O list (no maximum).
		No	No	Stream-oriented; transfer bytes according to I/O list (no maximum).
Free-Formatted	Yes	Yes	No	Line-oriented; transfer up to *n* bytes per line. Spilling may occur.
		No	Blanks	Record-oriented; transfer exactly *n* bytes per record. Spilling may occur.
	No	Yes	No	Line-oriented; transfer characters up to system-default line length according to I/O list. Spilling may occur.
		No	No	Line-oriented; transfer characters up to the system-default line length according to I/O list. Spilling may occur.

You must give the *LEN=* option without giving the *ATT="L"* option (i.e., the file must be record-oriented) if you plan to reference the file in a BACKSPACE statement, position it by means of a call to FSEEK (a FORTRAN 5 runtime routine), or position it implicitly in a random READ/WRITE statement by specifying *REC=n.*

Whatever you specify for *LEN=* will be the length of every record, padded if necessary; when a line of formatted output exceeds this length, FORTRAN 5 signals the error OUTPUT RECORD TOO LONG. When a line of free-formatted output exceeds this length, the excess *spills* to the next record. FORTRAN 5 does not split numeric data items across two records, but does split Hollerith and string constants. If a data item other than a Hollerith or string constant spills from one record to a second, and cannot fit on the second record, FORTRAN 5 signals the error OUTPUT RECORD TOO LONG. If you also specify *ATT="L", LEN=* will represent the maximum record length, rather than the absolute record length.

3. *REC=n,* where *n* is an integer expression that represents the number of records you want to reserve for a contiguous file. Use this option only if you specify the "C" option (file type), and if you need to create the file. If you specify *REC=n,* but do not specify the *LEN=* option, *n* will be the number of disk blocks allocated to the file. Each record will be 1 disk block in length (512 bytes).

4. *ERR=label,* where *label* is the label of a statement within the program to which control passes if an error occurs. Upon execution, the OPEN statement attempts to open the specified file. If it does not exist, and you specified the *ERR=* option, FORTRAN 5 does not create the file and transfers control to the statement specified.

Standard File Assignments

FORTRAN 5 recognizes certain assignments of unit numbers to files. These numbers need not appear in an OPEN statement unless you want to alter the standard assignment. The first time you reference a file in an I/O statement, if you opened it explicitly (by using an OPEN statement), processing continues according to the information you gave in the OPEN statement. If you did not OPEN the file, and a standard assignment exists for that file, FORTRAN 5 opens it (implicitly) according to the standard file assignment.

The standard file assignments are:

Unit Number	File
6	Incremental Plotter
8	TTY Punch
9	Card Reader
10	Current Console Printer
11	Current Console Keyboard
12	Line Printer (has "P" attribute)
13	Paper Tape Reader
14	Paper Tape Punch
15	TTY Tape Reader

If you open unit number 12 by default (writing to it without an explicit OPEN statement), the physical line length is the default associated with the operating system (see the reference manual appropriate to your operating system). Opening files 10 or 11 by default assigns a physical line length of 80 characters.

FORTRAN 5 provides you with the ability to establish your own file preconnections. It also allows you to use IBM-style preconnections. For information on both methods, see Part I, Chapter 1 in the *FORTRAN 5 Programmer's Guide.*

Examples

```
OPEN 2, "X1",ATT="A",ERR=100

OPEN 6, XARAY,ATT="OC",LEN=40,REC=100
```

CLOSE, ENDFILE

Close an opened file.

Format

CLOSE unit

ENDFILE unit

Where:

unit is an integer that specifies the FORTRAN 5 unit number of an opened file.

Statement Execution

If you close a file by using either a CLOSE or ENDFILE statement, you can reopen that file by using an OPEN statement.

Examples

```
CLOSE 4
ENDFILE 4
```

DELETE

Deletes a file from disk.

Format

DELETE filename

Where:

filename is the name of the file you want to delete.

Statement Execution

FORTRAN 5 deletes information about the file from the system directory. If the file does not exist, FORTRAN 5 does not signal an error. If the file has an abnormal condition (if permanent, if open, etc.), FORTRAN 5 signals an error.

Example

```
DELETE "MUM"
```

RENAME

Changes the name assigned to an existing file.

Format

RENAME filename1, filename2

Where:

filename1 is the name of the file you want to change.

filename2 is the new name you want to assign.

Statement Execution

If you attempt to rename a file to an existing name, processing will terminate and FORTRAN 5 will signal an error.

Example

```
RENAME "SEARCH","PROBE"
```

REWIND, BACKSPACE

Alter the position of the record pointer in a file.

Format

REWIND unit

BACKSPACE unit

Where:

unit is an integer that specifies the FORTRAN 5 unit number of an opened file or device.

Statement Execution

The REWIND statement repositions the record pointer to the beginning of the specified file. The BACKSPACE statement backspaces the file's record pointer and positions it to the beginning of the previous record.

If FORTRAN 5 executes a REWIND or BACKSPACE statement when either the specified device is at its beginning or the record pointer is pointing to the first record of the specified file, neither statement has an effect. If you use the BACKSPACE statement, you must have specified the *LEN=* option when you OPENed the file and must not have specified *ATT="L"*. If you do not specify *LEN=*, FORTRAN 5 signals an error at runtime.

Examples

```
REWIND 2

BACKSPACE 5
```

End of Chapter

Chapter 6
FORMAT Statement

The FORMAT statement designates the structure of the records and the form of the data fields within the records of a file. These records appear in an associated input or output statement (READ, WRITE, PRINT, PUNCH, ENCODE, or DECODE). A FORMAT statement is a nonexecutable statement that consists of editing and specification codes.

The FORMAT statement has the form:

n FORMAT (f_1 [s_1] f_2 [s_2] ... f_m [s_m])

Where:

n is a statement label that appears in the referencing I/O statement.

f is a field descriptor, a group of field descriptors, or a Hollerith or string constant.

s is a field separator (details listed below).

You must label all FORMAT statements.

The list of field descriptors, field separators, and Hollerith and string constants is called the *format specification*. You must enclose the list in parentheses.

A field descriptor determines the form of the datum being transferred and the type of conversion you want performed, if any. The form of a field descriptor is:

[r]qw[.d]

Where:

r is a repeat count indicating the number of times you want FORTRAN 5 to interpret this descriptor.

q is a format code.

w is an integer constant that specifies the field's width in characters.

d is the number of characters to the right of the decimal point.

For further information on the repeat count, see the section "Group Repeat Specifications".

Note that w must be large enough to include the sign for all numbers, and the decimal point and exponent for real, double precision, complex, and double precision complex numbers.

The field separators are slashes and commas. A slash indicates the end of a record and a comma indicates the end of a datum within the record. For more information, see the section "Field Separators".

The following is a list of field descriptors:

1. **Numeric**

Iw	Integer Data
Gw	Integer Data (may substitute for I, L, or S)
Ow	Octal Data (Base 8)
Yw	Hexadecimal Data (Base 16)
Bw	Binary Data (Base 2)
Fw.d	Real Data
Ew.d	Explicit Exponent Real Data
Dw.d	Double Precision Real Data
Gw.d	Generalized Real Data (may substitute for F, E, or D)

2. **Logical**

Lw	Logical Data

3. **String and Editing**

Aw	Alphanumeric Data
Rw	Alphanumeric Data, Right-justified
Sw	String Data
nH	Hollerith Data
"..."	String Data
'...'	String Data
nX	Column Positioning Control
Tn	Tabulation Control

You may precede any of the numeric field descriptors by a scale factor of the form *nP,* where *n* is an integer constant that specifies the number of positions you want to scale the decimal point. For further information see the section "Scale Factor".

The n you specify with some of the string and editing field descriptors indicates a number of characters or character positions.

FORTRAN 5 scans the format specification as well as the corresponding input or output list from left to right. It associates the field descriptors with the entities of the input or output list in the order in which you write them. For further information on the interaction between these statements see the section "Format and I/O Statement Association".

On input, if the width of the field specified by the field descriptor (w) is less than the number of characters in the input field, FORTRAN 5 processes w characters. If w is greater than the number of characters in the input field, FORTRAN 5 processes w characters, but you cannot be certain of the result. For example, if you input the integer 520 according to the field descriptor I4, the results are as follows:

1. If a digit from 0 through 9 immediately follows the integer, FORTRAN 5 stores the value of the four digits.

2. If an alphabetic character or special symbol immediately follows the integer, FORTRAN 5 signals an error.

3. If a record terminator immediately follows the integer, and you specified the "B" attribute when you opened the file, FORTRAN 5 will store and blank pad the integer 520.

4. If a record terminator immediately follows the integer, and you did not specify the "B" attribute when you opened the file, FORTRAN 5 will signal the error INPUT RECORD TOO SHORT.

On output, the results depend on the field descriptor you select. See the individual format code for an explanation.

Numeric Field Descriptors

I Format Code

Transmits numeric data in integer format.

Format

[r]Iw

Where:

r is a repeat count.

w is an integer constant that specifies the field's width in characters.

Input

FORTRAN 5 reads an integer value from the input field. It interprets an input datum preceded by a minus sign as a negative integer, and an input datum preceded by a plus sign or unsigned as a positive integer.

An input field containing all blanks has a value of zero. FORTRAN 5 ignores leading blanks; embedded and trailing blanks have a value of zero.

If you input a real value in integer format, FORTRAN 5 does not signal an error and truncates the real value to an integer value.

Output

FORTRAN 5 converts the internal value of the datum to integer format, right justifies it according to w with leading blanks if needed, and signs it if negative. The sign immediately precedes the first significant digit and follows any leading blanks. If the field width is not wide enough to output the datum, FORTRAN 5 outputs a field of asterisks (*).

Examples

Input

Input Characters	Format	Internal Value
50	I2	+50
50□	I3	+500
-5□□3	I5	-5003
□□□□	I4	0
□1234	I3	+12
+981	I6	Undefined Results (format larger than input field)

Output

Internal Value	Format	Output Characters
+50	I2	50
+50	I3	□50
-50	I2	** (format too small)
-50	I3	-50
-50	I5	□□-50

O Format Code

Transmits one word of any datum in unsigned octal format.

Format

[r]Ow

Where:

r is a repeat count.

w is an integer constant that specifies the field's width in characters.

Input

For any given octal field, FORTRAN 5 stores only one word (16 bits). If the value of the octal digits is greater than 177777, FORTRAN 5 signals an error.

The corresponding list entity should be of type integer. If the input field contains a nonoctal character, FORTRAN 5 signals an error. Legal octal characters are 0 through 7 and blank. If you sign an input datum, FORTRAN 5 signals an error.

An input field containing all blanks has a value of zero. FORTRAN 5 ignores leading blanks; embedded and trailing blanks have a value of zero.

Output

FORTRAN 5 does not output more than six octal digits for one field. If the field contains less than six digits, FORTRAN 5 right justifies it and blank pads the field on the left. If the digits require more than the width specified by the field descriptor (w), FORTRAN 5 outputs the leftmost w characters. If you wish to output a field that contains more than one word, you may put the field in array form, as in the following:

```
     REAL X
     INTEGER IA(2)
     EQUIVALENCE (X,IA(1))
     WRITE (12,100) IA
100  FORMAT (1X,2O6)
```

Examples

Input

'nput Characters	Format	Internal Value (Decimal)
31	O2	+25
200000	O6	Error (input value too big)
25□	O3	+168
635	O2	+51
635	O3	+413
635	O4	Undefined Results (format larger than input field)

Output

Internal Value (Decimal)	Format	Output Characters
+33	O5	□□□41
+100	O2	14
+100	O3	144

Y Format Code

Transmits one word of any datum in unsigned hexadecimal format.

Format

[r]Yw

Where:

r is a repeat count.

w is an integer constant that specifies the field's width in characters.

Input

Internally, FORTRAN 5 stores only one word or four hexadecimal digits for any given field. It signals an error if you input a number greater than FFFF.

The corresponding list entity should be of type integer. If the input field contains a nonhexadecimal character, FORTRAN 5 signals an error. Legal hexadecimal characters are 0 through 9, A through F, and blank. If you sign an input datum, FORTRAN 5 signals an error.

An input field containing all blanks has a value of zero. FORTRAN 5 ignores leading blanks; embedded and trailing blanks have a value of zero.

Output

FORTRAN 5 does not output more than four digits for one field. If the field contains less than four digits, FORTRAN 5 right justifies the digits and blank pads the field on the left. If the digits require more than the width specified by the field descriptor (w), FORTRAN 5 outputs the leftmost w characters.

Examples

Input

Input Characters	Format	Internal Value (Decimal)
42	Y2	+66
1FFFF	Y5	Error (input value too big)
1□□	Y3	+256
1D2	Y2	+29
1D2	Y3	+466
1D2	Y4	Undefined Results (format larger than input field)
-3C2	Y4	Error (minus sign not legal)

Output

Internal Value (Decimal)	Format	Output Characters
+128	Y5	□□□80
+433	Y2	1B
+433	Y3	1B1

B Format Code

Transmits one word of any datum in unsigned binary format.

Format

[r]Bw

Where:

r is a repeat count.

w is an integer constant that specifies the field's width in characters.

Input

FORTRAN 5 stores only one word or 16 binary digits for any given field. If you sign an input datum, enter a nonbinary character (a character other than 0, 1, or blank), or enter a number greater than 111111111111111, FORTRAN 5 signals an error.

An input field containing all blanks has a value of zero. FORTRAN 5 ignores leading blanks; embedded and trailing blanks have a value of zero.

Output

FORTRAN 5 does not output more than 16 binary digits for one field. If the field contains less than 16 digits, it right justifies the digits and blank pads the field on the left. If the digits require more than the width specified by the field descriptor (w), FORTRAN 5 outputs the leftmost w characters.

Examples

Input

Input Characters	Format	Internal Value (Decimal)
101	B3	+5
-111	B3	Error (minus sign not legal)
11111111111111111	B17	Error (input value too big)
1□□	B3	+4
101□11	B2	+2
101□11	B5	+21
101□11	B7	Undefined Results (format larger than input field)
1□32	B4	Error (nonbinary digits)

Output

Internal Value (Decimal)	Format	Output Characters
+22	B5	10110
+15	B2	11
+15	B4	1111
+15	B6	□□1111

F Format Code

Transmits numeric data in real format.

Format

[s][r]Fw.d

Where:

- s is a scale factor.
- r is a repeat count.
- w is an integer constant that specifies the field's width in characters.
- . is a decimal point.
- d is an integer constant that specifies the number of characters to the right of the decimal point.

Input

The input field consists of a string of digits and, optionally, a sign and/or decimal point. If there is a decimal point in the input datum, it overrides the position of the decimal point implied by the field descriptor. If the input datum does not contain a decimal point, FORTRAN 5 assumes one such that the rightmost d digits of the datum are to the right of the decimal point.

Since w is a count of all characters in the field, it includes the sign (if any) and the decimal point.

FORTRAN 5 interprets an input datum preceded by a minus sign as a negative number; if a plus sign precedes the datum or if it is unsigned, FORTRAN 5 interprets the datum as a positive number. An input field containing all blanks has a value of zero. FORTRAN 5 ignores leading blanks; embedded and trailing blanks have a value of zero.

Output

FORTRAN 5 right justifies the digit string with leading blanks if its length is less than w, signs it if it is negative, and places the decimal point according to the position determined by d. If the number of digits following the decimal point in the digit string's internal representation is greater than d, FORTRAN 5 rounds the fraction at the d position.

The field width (w) must be large enough to accommodate a minus sign, if any, (FORTRAN 5 suppresses plus signs on output) and a decimal point. If w is not large enough, FORTRAN 5 outputs a field of asterisks.

Examples

Input

Input Characters	Format	Internal Value
-123.41	F7.2	-123.41
12345	F5.2	+123.45
+6□3.□53	F8.3	+603.053
+6□3.□53	F7.3	+603.05
6□3.□53	F5.2	+603.0
4598.3	F20.1	Undefined Results (format larger than input field)

Output

Internal Value	Format	Output Characters
-98.7	F7.2	□-98.70
+100.67	F8.1	□□□100.7
+123.54	F5.2	***** (format too small)
+42.35	F9.3	□□□42.350

E Format Code

Transmits numeric data in exponential format.

Format

[s][r]Ew.d

Where:

s	is a scale factor.
r	is a repeat count.
w	is an integer constant that specifies the field's width in characters.
.	is the decimal point.
d	is an integer constant that specifies the number of characters to the right of the decimal point.

Input

The input field consists of a string of digits and, optionally, a sign and/or decimal point. You may follow this by an exponent in one of two forms:

- a signed integer constant or
- an E followed by a signed or unsigned integer constant.

If there is a decimal point in the input datum, it overrides the position of the decimal point implied by the field descriptor. If the input datum does not contain a decimal point, FORTRAN 5 assumes one such that the rightmost d digits of the nonexponent part of the datum are to the right of the decimal point. The exponent, therefore, does not override an implied decimal point; however, it does override any scale factor you specify.

Since w is a count of all characters in the field, it must include the sign (if any), the decimal point, and the exponent.

FORTRAN 5 interprets an input datum preceded by a minus sign as a negative number; if a plus sign precedes the datum or if it is unsigned, FORTRAN 5 interprets the datum as a positive number. An input field containing all blanks has a value of zero. FORTRAN 5 ignores leading blanks; embedded and trailing blanks have a value of zero.

Output

FORTRAN 5 right justifies the output characters within the external field, outputs leading blanks if its length is less than w, and signs it if it is negative. The decimal point is followed by d digits. The letter E and a decimal exponent follow the number. FORTRAN 5 signs the exponent if negative. If the number of fraction digits is greater than d, FORTRAN 5 rounds the fraction at the d position.

The field width (w) must be large enough to accommodate a minus sign, if any (FORTRAN 5 suppresses plus signs on output), a decimal point, and an exponent. Therefore, w must be greater than or equal to d + 5 or else FORTRAN 5 outputs a field of asterisks.

Examples

Input

Input	Format	Internal Value
24.42E600	E9.3	Error (exponent too large)
-.21E5	E6.4	-21000
.456	E4.3	+.456
-1□3.99E+5	E7.1	-103.99
-1□3.99E+5	E9.1	Error (format too small -- only part of exponent read in)
.456	E4.2	+.456
456E-2	E6.3	+4.56
31-01	E5.1	+3.1
.31-01	E6.1	+.031

Output

Internal Value	Format	Output
+12.34	E10.3	□□.123E□02
-19.54	E6.2	****** (format not wide enough to include exponent)
+31.987	E10.3	□□.320E□02

D Format Code

Transmits numeric data in double precision format.

Format

[s][r]Dw.d

Where:

s is a scale factor.

r is a repeat count.

w is an integer constant that specifies the field's width in characters.

. is a decimal point.

d is an integer constant that specifies the number of characters to the right of the decimal point.

Input

The input field consists of a string of digits and, optionally, a sign and/or decimal point. You may follow this by an exponent in one of two forms:

- A signed integer constant; or
- A D followed by a signed or unsigned integer constant.

If there is a decimal point in the input datum, it overrides the position of the decimal point implied by the field descriptor. If the input datum does not contain a decimal point, FORTRAN 5 assumes one such that the rightmost d digits of the nonexponent part of the datum are to the right of the decimal point. The exponent, therefore, does not override an implied decimal point; however, it does override any scale factor you specify.

Since w is a count of all characters in the field, it must include the sign (if any), the decimal point, and the exponent.

FORTRAN 5 interprets an input datum preceded by a minus sign as a negative number; if a plus sign precedes the datum or if it is unsigned, FORTRAN 5 interprets the datum as a positive number. An input field containing all blanks has a value of zero. FORTRAN 5 ignores leading blanks; embedded and trailing blanks have a value of zero.

Output

FORTRAN 5 right justifies the output characters within the external field, outputs leading blanks if its length is less than w, and signs it if negative. The decimal point is followed by d digits. The letter D and a decimal exponent follow the number. FORTRAN 5 signs the exponent if it is negative. If the number of fraction digits is greater than d, FORTRAN 5 rounds the fraction at the d position.

The field width (w) must be large enough to accommodate a minus sign, if any (FORTRAN 5 suppresses plus signs on output), a decimal point, and an exponent. Therefore, w must be greater than or equal to d + 5 or else FORTRAN 5 outputs a field of asterisks.

Examples

Input

Input Characters	Format	Internal Value
1234.56789012345	D16.0	1234.56789012345
1234567890123456	D16.5	12345678901.23456
1234567890123456	D20.0	Undefined Results (format larger than input field)
-12345□□□□□□□□□□□	D16.5	-1234500000.
□□123.45D5	D10.7	12345000.

Output

Internal Value	Format	Output Characters
12345.6789012345	D24.15	□□□□.123456789012345D□05
-123.4567	D12.6	-.123457D□03
123.4567	D12.7	.1234567D□03
-123.4567	D12.7	************ (format not large enough to include sign)

G Format Code

Transmits data using a generalized format code.

Format

[s][r]Gw.d

Where:

s is a scale factor.

r is a repeat count.

w is an integer constant that specifies the field's width in characters.

. is a decimal point.

d is an integer constant that specifies the number of characters to the right of the decimal point.

You can use this code with integer, logical, real, and double precision data, and string constants, where it is equivalent to using the I, L, R, D, and S field descriptors respectively. The equivalent form used depends on the type of the corresponding element in the I/O list.

Input

If there is a decimal point in the input datum, it overrides the position of the decimal point implied by the field descriptor. If the input datum does not contain a decimal point, FORTRAN 5 assumes one such that the rightmost d digits of the nonexponent part of the datum are to the right of the decimal point. The exponent, therefore, does not override an implied decimal point; however, it does override any scale factor you specify.

FORTRAN 5 interprets an input datum preceded by a minus sign as a negative number; if a plus sign precedes the datum or if it is unsigned, FORTRAN 5 interprets the datum as a positive number. An input field containing all blanks has a value of zero. FORTRAN 5 ignores leading blanks; embedded and trailing blanks have a value of zero.

Output

The external representation depends on the magnitude of the datum you want to convert. The output is in E format except when the magnitude of the internal datum (say n) is: .1 < n < 10**d. Within this range FORTRAN 5 applies F format according to the following formula:

Magnitude of Datum	Conversion
0.1 ≤ n < 1	F(w-4).d,4x
1 ≤ n < 10	F(w-4).(d-1),4x
.	.
.	.
.	.
10**(d-2) ≤ n < 10**(d-1)	F(w-4).1,4x
10**(d-1) ≤ n < 10**d	F(w-4).0,4x

(The field descriptor 4x means that four blanks will follow the numeric data representation.)

The field width (w) must be large enough to accommodate a minus sign, if any (FORTRAN 5 suppresses plus signs on output), a decimal point, and an exponent. Therefore, w must be greater than or equal to d + 5 or else FORTRAN 5 outputs a field of asterisks.

Examples

Input - See INPUT examples for F and E format codes.

Output

Format	Internal Value	Output Characters
G11.4	.012346	□□.1235E-01
	.12346	□□.1235□□□□
	1.2346	□□1.235□□□□
	12.346	□□12.35□□□□
	123.46	□□123.5□□□□
	-1234.6	□-.1235E□04

Logical Field Descriptor

L Format Code

Transmits data in logical format.

Format

[r]Lw

Where:

r is a repeat count.

w is an integer constant that specifies the field's width in characters.

Input

The input field must consist of a T or F (for true and false respectively). You may precede the T or F with blanks and may follow it with other characters. If the first nonblank character is T, FORTRAN 5 stores the value true; if the first nonblank character is F, it stores the value false. Any characters following the T or F are ignored.

Output

If the internal value is true, FORTRAN 5 outputs w-1 blanks followed by a T. If the internal value is false, it outputs w-1 blanks followed by an F.

Examples

Input

Input Characters	Format	Internal Value
□□TRUE	L6	True
□F120	L5	False
FALSE	L10	Undefined Results (format larger than input field)
12345	L5	Error (nonlogical characters)
FT	L2	False

Output

Internal Value	Format	Output Characters
True	L4	□□□T
False	L1	F

A Format Code

Transmits alphanumeric data in character (ASCII) format (see also R Format Code).

Format

[r]Aw

Where:

r is a repeat count.

w is an integer constant that specifies the field's width in characters.

Rules

FORTRAN 5 expresses characters as 8-bit quantities according to the ASCII code set. It transmits characters without alteration and does not convert them into a form suitable for computation.

Input

FORTRAN 5 reads w characters for an input field under the control of an A format code. If w is greater than the size of the corresponding input list entity, say v, the leftmost w - v characters in the input field are lost; FORTRAN 5 reads and stores the remaining v characters, left justifies them, and blank pads the field on the right.

The maximum number of characters that FORTRAN 5 can store for a variable or array element depends on the data type of the datum. The restrictions are as follows:

I/O List Entity	Maximum No. of Characters
Integer	2
Real	4
Double Precision	8
Complex	8
Double Precision Complex	16
Logical	2

Output

FORTRAN 5 outputs characters to an external field w characters wide.

If w is greater than the number of characters in the entity in the I/O list (v), FORTRAN 5 outputs v characters, right justifies them, and blank pads the field on the left. If w is less than v, FORTRAN 5 outputs the leftmost w characters and ignores the remaining characters.

Examples

See R Format Code for a composite example of both R and A formats.

R Format Code

Transmits alphanumeric data in character (ASCII) format (see also A Format Code).

Format

[r]Rw

Where:

r is a repeat count.

w is an integer constant that specifies the field's width in characters.

Input

Same as for A-type conversion, except FORTRAN 5 right justifies w characters in the element of the I/O list.

Output

Same as for A-type conversion, except FORTRAN 5 outputs the rightmost w characters in the element.

Examples

(comparing A and R format codes)

Input

The input field contains the characters ABCD.

Format	Data Type	Characters Stored
A1	Integer	A□
R1	Integer	□A
A2 or R2	Integer	AB
A3 or R3	Integer	BC (example with w>v)
A2	Real	AB□□
R2	Real	□□AB
A3	Real	ABC□
R3	Real	□ABC
A4 or R4	Real	ABCD

Note that in cases where w < 4, FORTRAN 5 does not transmit all characters of the input datum. Leftover characters on input may give you unpredictable results.

Output

On output, we will use a real variable that contains ABCD and an integer variable that contains AB.

Format	Data Type	Output Characters
A1	Integer	A
R1	Integer	B
A2 or R2	Integer	AB
A3 or R3	Integer	□AB
A2	Real	AB
R2	Real	CD
A3	Real	ABC
R3	Real	BCD
A4 or R4	Real	ABCD

S Format Code

Transmits alphanumeric data in character (ASCII) format with no storage limitations.

Format

[r]Sw

Where:

r is a repeat count.

w is an integer constant that specifies the field's width in characters.

Input

One internal word contains two ASCII characters. There are no storage limitations as with A and R format codes; i.e., the data type of the corresponding entity in the I/O list does not determine the number of characters FORTRAN 5 will transmit. You must terminate input of a string with a new-line character.

FORTRAN 5 associates the S field descriptor with an entity in the I/O list and, beginning at the location of this entity, transmits up to w characters ignoring the data type. (Because FORTRAN 5 does not recognize data boundaries, you must specify the correct field width.) Character transmission terminates if:

- FORTRAN 5 transmits w characters;
- FORTRAN 5 encounters a record delimiter (CR, NL, NULL, or FF). The delimiter is not transferred.

FORTRAN 5 appends a null character to the end of the input string. Note that this null byte may increase the number of words required to store the string.

Output

FORTRAN 5 outputs up to w characters. Transfer begins with the first character position you reference and terminates after one of the following conditions is met:

- w characters are transferred;
- A null character is encountered. In this case, the transfer of characters stops and FORTRAN 5 outputs enough blanks to make the string w characters in length. The null character is not transferred. (Note that the other record delimiters -- CR, NL, and FF -- do not have the same effect as a null character; FORTRAN 5 transmits them as any other ASCII character.)

Examples

The symbol # represents a null character and the symbol * represents an unused byte.

Input

Input	Format	Internal String	Total Number of Words
123456	S2	12#*	2
	S3	123#	2
	S4	1234#*	3
	S6	123456#*	4
	S8	123456#***	5

Output

Internal String	Format	Output
□NOW□IS□THE□TIME#	S16	□NOW□IS□THE□TIME
	S20	□NOW□IS□THE□TIME□□□□
	S11	□NOW□IS□THE

Hollerith and String Constants

Reads or writes an ASCII character string directly into or from a FORMAT statement.

Format

"string"
'string'
nHstring

Where:

" and '	are valid delimiters containing the specified character string.
n	is an unsigned integer constant that specifies the number of characters in the string.
H	is the designation for a left-justified Hollerith constant.

Rules

No entity in an I/O list corresponds to the string data.

An apostrophe cannot appear within a string delimited by apostrophes; a quotation mark cannot appear within a string delimited by quotation marks.

Input

FORTRAN 5 stores each character of the string in ASCII format. If you specify H format, it reads n characters.

Output

If the internal string is in H format, FORTRAN 5 outputs n characters. If the internal string is delimited by apostrophes or quotation marks, it outputs the enclosed characters.

Examples

Input

Putting blank Hollerith or string constants in a FORMAT statement used for input allows you to change the format specification at program execution time. For example, if X = 3.2 and Y = 7.4 and you are given the statements:

```
         READ (11,100) X,Y
         WRITE (11,100) X
         WRITE (11,100) Y
100      FORMAT (10H□□□□□□□□□□,F6.2)
```

if the input record contains VALUE□IS:□, the output records will contain:

```
VALUE□IS:□□□3.20
VALUE□IS:□□□7.40
```

Output

```
100 FORMAT ("□OUTPUT□IS:□",F3.1,2X,F3.1)
101 FORMAT '□OUTPUT□IS:□',F3.1,2X,F3.1)
102 FORMAT 12H□OUTPUT□IS:□,F3.1,2X,F3.1)
```

An output statement referencing any of the above statements results in the following output (values are plugged in):

```
□OUTPUT□IS:□3.4□□5.6
```

X Format Code

Allows for relative column-positioning control within a given record (see also T Format Code).

Format

nX

Where:

n is an unsigned integer constant that specifies a number of character positions.

Input

On input of the external field, FORTRAN 5 skips n character positions. You must specify n even if you want to skip only one character position.

Output

In the external output field, FORTRAN 5 inserts n ASCII blank characters. You must specify n even if you want to insert only one blank character.

Examples

See T Format Code for a composite example of both X and T formats.

T Format Code

Allows for tabbing control on a given record (see also X Format Code).

Format

Tn

Where:

n is an integer constant that specifies a character or print position.

Input

Tabbing occurs to the character position in the record specified by n. You may tab forward or backward. In tabbing forward, FORTRAN 5 will read the character(s) beginning at position n; in tabbing backward, it will reread the character(s) beginning at position n.

Output

Tabbing occurs to the character position in the record specified by n. Even if you prepare the file for printing (open it with the "P" attribute), tabbing still occurs to character position n. However, if there is data at the beginning of the record, FORTRAN 5 will use the first character for carriage control and will not print it.

You may tab forward or backward. An example of backward tabbing, given a file prepared for printing, is as follows: FORTRAN 5 will output the format specification ("ABCDE",T3,"Q",T10) as BQDE (after a line feed); if you had not prepared the file for printing, FORTRAN 5 would output ABQDE. Note that Q replaced C in both examples.

FORTRAN 5 blank fills the output buffer before the execution of an I/O operation, so character positions which are not filled contain explicit blanks.

Examples

Given files that are not prepared for printing, the following compares X and T format codes on output.

Program Statement	**Resulting Output**
FORMAT (I4,5X,F5.2)	□425□□□□□98.63
FORMAT (I5,T10,A3)	□-321□□□□XXX Print position 10
FORMAT (T8,3X,4HHERE)	□□□□□□□□□□HERE Print position 8

Scale Factor

The scale factor changes the location of the decimal point in the external representation of a real number, with respect to its internal value.

Format

nP

Where:

n is an integer constant that specifies the number of positions you want to move the decimal point and in what direction.

Rules

You may affix a scale factor to a field descriptor in the format specification, such as 2PE15.5, or let it stand alone, as 2P,E15.5. Once you specify a scale factor, it remains in effect for all subsequent D, E, and F (and G when involving real numbers) field descriptors within the same format specification until FORTRAN 5 encounters another scale factor. (When FORTRAN 5 encounters the rightmost parenthesis, rescanning the format specification does not reset the scale factor to zero.) To reset the scale factor to zero, specify 0P.

Input

If the number you enter does not contain an explicit exponent, the scale factor conversion formula is:

internal value = number entered x 10**(-n)

Note that 10**(-n) is equivalent to 1/10**n.

The internal representation of the number is equivalent to the external representation with the decimal point shifted to the left n places if the scale factor is positive, and to the right n places if the scale factor is negative. (You will see that on output you can simply undo this process by specifying the same scale factor.) If the number entered contains an explicit exponent, FORTRAN 5 ignores the scale factor. For example, if you enter the number 3.E02 and apply the field descriptor 2PE8.2, the stored result is 300, i.e., the scale factor has no effect.

Output

The conversion formula for scale factoring on output is:

external number = internal value x 10**n

You can specify the scale factor for all real numbers on output. With an F field descriptor, the effect is the same as for input except FORTRAN 5 moves the decimal point in the opposite direction: it shifts the decimal point to the right n places if the scale factor is positive, and to the left n places if the scale factor is negative. For example, if you want to output the internally stored value 12.987 as 1298.76, use the format code 2PF7.4. With an E or D field descriptor, the scale factor given alters the position of the decimal point, adjusting the exponent to account for this change. The result is that no change occurs in the value of the number.

As mentioned above, a scale factor applies to all subsequent F, E, D, and certain G field descriptors within the same format specification until FORTRAN 5 encounters another scale factor. Therefore, if it encounters an I, O, Y, or B field descriptor, the scale factor does not apply.

Use of a scale factor allows you to retain more precise decimal values. The binary system of numbers expresses integers precisely and approximates most decimal fractions. A good example of where decimal fractions must be as precise as possible is in accounting. Given the entry $8.59, the dollars and cents are of equal importance during processing. To achieve decimal precision, the field descriptor of the number on input might be -2PF7.2. The internal representation is then 859., making the number an integer. After performing all calculations, you output the number using the same format, -2PF7.2, resulting in a number that reflects precise dollars and cents.

SCALE FACTOR (continued)

Examples

Input

Input Characters	Format	Internal Value
-25.44 345.71	FORMAT (2PF10.2,F9.2)	-.2544 +3.4571
12.345	3PE8.3	+.012345
12.345	-3PE8.3	+12345.

Output

Internal Value	Format	Output Characters
+501.33	E10.3	□□.501E□03
+501.33	1PE10.3	□5.013E□02
+501.33	2PE10.3	50.133E□01
+12.217	F7.3	□12.217
+12.217	-1PF7.3	□□1.222

Group Repeat Specifications

To repeat one field descriptor or a group of field descriptors you must specify a repeat count. The affected field descriptors are interpreted the specified number of times.

Format

r(field descriptor(s))

Where:

r is an unsigned integer constant that specifies the number of times you want to repeat the descriptor(s).

Rules

You may affix a repeat count to any basic field descriptor except those of the form H, X, T, string constants, and the format code Z.

You may precede a group of field descriptors or field separators with a repeat count, enclosing the group in parentheses. This form of group is called a *basic group.* FORTRAN 5 interprets the entire group the number of times specified. If you do not specify a group repeat count, the repeat count is one.

You may form a further grouping by enclosing field descriptors, field separators, or basic groups (or a combination of the three) in parentheses, and may specify a repeat count for the group. You may indicate up to 16 levels of nested parenthesized groups within the same FORMAT statement.

Examples

(Forms listed together are equivalent provided there is no rescanning.)

```
FORMAT (2I2,3F11.2)
FORMAT (I2,I2,F11.2,F11.2,F11.2)

FORMAT (F9.2,(I2,I3))
FORMAT (F9.2,1(I2,I3))

FORMAT (G13.2,2(F10.1,3A2))
FORMAT (G13.2,F10.1,A2,A2,A2,F10.1,A2,A2,A2)
```

Carriage Control

If you want to format a file's output, and you prepare the file for printing (open it with the "P" attribute), FORTRAN 5 does not print the first character of each output record; it uses this character for vertical carriage control. To specify printed output, designate the default print output unit, issue a PRINT statement, or specify a file which you opened with the "P" attribute. FORTRAN 5 treats the first character of a record in a file not prepared for printing as data.

The printing control characters are:

0	Print after double line feed
-	Print after triple line feed
1	Print after form feed
+	Print with no carriage advance (for overprinting)
blank or any other character	Print after line feed

For examples, see Table 6-1.

Suppressing a New-line Character (Z)

Normally, a new-line character is the last character in a record output to a file. You use the Z format code to suppress the new-line character on output of the current record. It must appear in the format specification before the slash that indicates the end of that record, or before the terminating parenthesis of the format specification. In addition, if you prepared the file for printing, the output record immediately following the record ending in Z must begin with the carriage control character "+" or the Z will have no effect. (Logically, a Z followed by a plus sign (+) leaves the character pointer in its current position.) If you do not prepare the file for printing, the plus sign is not necessary.

For examples, see Table 6-1.

Field Separators

A format specification contains field descriptors which you must separate unambiguously. In general, you separate the field descriptors of a given record by commas. You can use a slash to separate field descriptors, but a slash also terminates input or output of the current record and initiates a new record. For example:

```
    REAL A,B,C,D
    WRITE (12,20) A,B,C,D
20  FORMAT (F3.1,D12.7/2G10.5)
```

is equivalent to:

```
    REAL A,B,C,D
    WRITE (12,20) A,B
20  FORMAT (F3.1,D12.7)
    WRITE (12,30) C,D
30  FORMAT (2G10.5)
```

Use of repetitive slashes allows you to either bypass input records or output blank records. The first slash terminates input or output of the current record and the remaining slashes indicate blank or skipped records. However, if repetitive slashes appear at the beginning or end of a format specification, each slash indicates a blank or skipped record; the left-hand parenthesis of the format specification initiates a new record and the right-hand parenthesis terminates the current record. For example, the following statements (using a file prepared for printing):

```
      WRITE (12,100)
100   FORMAT (1X,T21,"CHAPTER 6"///"□",
    1"TASKS"//"□","TEXT"//)
```

result in the output:

```
      CHAPTER 6

(blank line)
(blank line)
TASKS
(blank line)
TEXT
(blank line)
(blank line)
```

Because a slash is a field descriptor of sorts, you may follow or precede it by a comma.

Although you may separate all field descriptors, separation is not mandatory if FORTRAN 5 can identify two field descriptors unambiguously. For example:

```
9 FORMAT (I4"DATA□IS:"E15.5)
```

The quotation marks set the string constant off from preceding and following descriptors.

Table 6-1. Print File Examples

Program Statement	***Resulting Output***
Given File Prepared for Printing	
Given Datum = 125	
1. FORMAT (I4) FORMAT (1X,I3) FORMAT ("□",I3) FORMAT ('□',I3) FORMAT (1H□,I3)	125 (after one line feed)
2. FORMAT (I3)	25 (after one form feed)
Given X = '+ABC'	
3. WRITE (12,20) X 20 FORMAT (A4)	ABC (with no carriage advance)
Given I=249 and J=133	
4. WRITE (12,9) I WRITE (12,10) J 9 FORMAT ("□",I6,Z) 10 FORMAT ("+",I6)	□□□249□□□133
5. WRITE (12,9) I WRITE (12,10) J 9 FORMAT ("□",I6,Z) 10 FORMAT (I6)	□□□249 □□133
Given File Not Prepared for Printing	
6. WRITE (12,9) I WRITE (12,9) J 9 FORMAT (I6,Z)	□□□249□□□133

Format and I/O Statement Association

Each noncomplex data element in a formatted input or output statement list corresponds to a single entity in the associated format specification. Each complex and double precision complex datum corresponds to two format items (see discussion below). Except for the effect of repeat specifications, FORTRAN 5 scans the format specification and the corresponding I/O list from left to right. It associates the field descriptors of the format specification with the entities of the I/O list in the order in which you wrote them.

If an I/O statement contains an I/O list, the format statement it references must contain at least one field descriptor in a form other than H, X, T, P, or a string constant; otherwise, an infinite loop occurs.

Upon execution of an input statement, FORTRAN 5 reads one record. The FORMAT statement associated with the input statement determines if FORTRAN 5 will read additional records and in what form it will store these records. FORTRAN 5 reads a new record whenever it encounters a slash in the format specification, or whenever it reads the rightmost right parenthesis of the format specification (and I/O list entities remain to be read).

On output, FORTRAN 5 writes one record to the specified unit. It outputs the additional records when encountering a slash in the referenced FORMAT statement or the rightmost right parenthesis (and I/O list entities remain to be written).

Whenever FORTRAN 5 encounters an I, O, B, Y, F, E, D, G, L, A, R, or S field descriptor in a format specification, it determines if there is a corresponding entity in the I/O list (each descriptor corresponds to one I/O list entity). If there is, it performs the conversion and passes control to the next field descriptor. If there is no corresponding entity, format control terminates.

If the entity in an I/O list is an unsubscripted array name, FORTRAN 5 inputs or outputs the elements of the array in the order of subscript progression; each element must have a field descriptor with which to associate.

String constants and H, X, T, and P field descriptors appearing in a format specification do not correspond to any I/O list entity. FORTRAN 5 communicates the given information directly with the record. If there are no more I/O list entities to process, yet the next field descriptor of the format specifications is in either H, X, or T form, or is a string constant, FORTRAN 5 processes each field descriptor until it encounters a field descriptor that must associate with an I/O list entity. At that point, format control terminates.

Complex Data in I/O Lists

FORTRAN 5 treats a complex datum appearing in the I/O list of a formatted I/O statement differently than other list entities. Because a complex datum consists of an ordered pair of real values, you must specify a field descriptor for each of the values.

On input, FORTRAN 5 reads two entities of the I/O list. FORTRAN 5 stores the entities as one complex datum, having a real part and an imaginary part.

On output, FORTRAN 5 outputs the internal value according to a repeated field descriptor (such as 2F10.2) or two or more successive field descriptors (you may specify string constants and H, X, and T field descriptors between the two field descriptors that govern the structure of the complex datum).

Multiple-Record Formatting

FORTRAN 5 can output more than one record using a single FORMAT statement if either of the following conditions exists:

1. If there is a slash (/) in the format specification, it designates the termination of one record and, if another record exists, the start of the next.
2. If the field descriptors in a FORMAT statement are exhausted and there are more I/O list entities to process, the current record terminates, a new one starts, and FORTRAN 5 *reuses* the FORMAT statement.

Any field descriptor or basic group without a repeat count has a repeat count of 1. For example:

```
FORMAT (2I2,F9.3,(A2,2(L2)))
```

can be written as:

```
FORMAT (2I2,1F9.3,1(1A2,2(1L2)))
```

As mentioned previously, you can nest up to 16 levels of parenthesized groups. FORTRAN 5 assigns level numbers to each pair of nested parentheses starting with the outermost delimiting parentheses as level 0.

Note, in Figure 6-1, that the beginning of a level includes the repeat count if you specify one.

When the entities of an I/O list use all the field descriptors of its corresponding FORMAT statement, and there are more entities to format, FORTRAN 5 returns to the last preceding level 1 left parenthesis (including the repeat count if any), or, if no level 1 exists, to the first left parenthesis of the format specification. For example:

```
20 FORMAT (I4,2F10.2,3(A2,R6),4(A6))
40 FORMAT (F10.2,E15.7,I5,10X,3HEND)
```

If there are additional I/O list entities to process, repetition of FORMAT statement 20 starts at the repeat count 4 of 4(A6); repetition of FORMAT statement 40 starts at F10.2.

Examples

Table 6-2 is a list of examples showing how format control processes the output of various I/O statements.

Summary of Rules for Formatted I/O

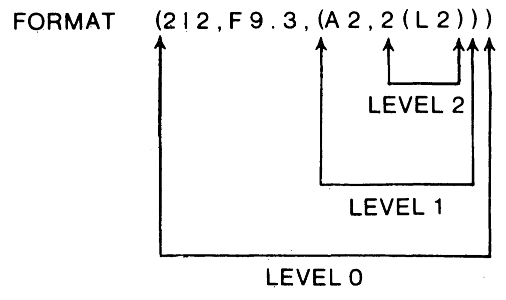
1. You must label a FORMAT statement.
2. Repeat counts (r), field width specifiers (w), decimal point specifiers (d), and character counts (n) that appear in field descriptors must all be unsigned integer constants.
3. You may not prefix an H, X, or T field descriptor, a Z format code, or a string constant with a repeat count (r). They may, however, appear within a parenthesized group that is repeated.
4. If a formatted I/O statement contains an I/O list, the FORMAT statement it references must contain at least one field descriptor of a form other than H, X, T, P, a string constant, or the format code Z. If not, an infinite loop occurs.

Input

1. FORTRAN 5 ignores leading blanks input to a numeric field. Embedded and trailing blanks have a value of zero. An input field containing all blanks has a value of zero.
2. A decimal point entered in a real input field overrides the decimal point position specified by d of the field descriptor.
3. If the field width (w) specified in the field descriptor is greater than the field you input, the results are undefined.
4. If the field width (w) specified in the field descriptor is smaller than the input field, w characters are input.
5. A field that is input in O, Y, or B format must be an unsigned integer.
6. An exponent in an input field does not override the decimal point implied by its field descriptor (d); however, the exponent does override any given scale factor.

Output

1. The field width specifier (w) in a field descriptor must be large enough to include (where applicable) a sign, a decimal point, and an exponent; if it is not, FORTRAN 5 will signal an error.
2. In field descriptors where you specify the field width (w) and the decimal point position (d), w must be greater than d, or FORTRAN 5 will signal an error.
3. If the number of digits following the decimal point in the internal representation of a real number is greater than d in the field descriptor, FORTRAN 5 rounds the fraction at the d position.
4. If you prepare a file for printing (open it with the "P" attribute), the first character of each record in the specification is a carriage control character; FORTRAN 5 does not print it.
5. If the number of nonblank characters to output is less than that specified in the field descriptor (w), FORTRAN 5 right justifies the characters, and precedes it by the appropriate number of blanks.
6. If the number of nonblank characters to output is greater than that specified in the field descriptor (w) (except for O, Y, B, A, and R field descriptors), FORTRAN 5 outputs a field of asterisks.
7. If a field descriptor is in either O, Y, B, or A format and the internal field's width is greater than that specified by the field descriptor (w), FORTRAN 5 outputs the leftmost w characters.



SD-01197 *Figure 6-1. Nested Parentheses Example*

Table 6-2. Format Control Examples

Given A = 2.75, B = 51.70, C = (1.0,2.0), I = 22, J = 39, K = 6, and L = 132.

Program Statements	Resulting Output
(Given file is prepared for printing.)	
WRITE (12,20) A,I,B,J 20 FORMAT (1X,F7.2,I5)	□□□2.75□□□22 □□51.70□□□39
WRITE (12,30) I,A,J,K,L 30 FORMAT (1X,I3,F10.2,2I4)	□22□□□□□□2.75□□39□□□6 132
WRITE (12,35) I,J 35 FORMAT (1X,I2,4H□RED,3X,I2,6H□BLACK,5(1H*),E15.4,3HEND)	22□RED□□□39□BLACK*****
WRITE (12,40) L,J,J,J,K,K,K 40 FORMAT (1X,I4,3(2H□!,I3))	□132□!□39□!□39□!□39 !□□6□!□□6□!□□6
WRITE (12,50) B,A 50 FORMAT (1X,2F5.2)	51.70□2.75
WRITE (12,60) C 60 FORMAT (1X"REAL:□"F5.2"□□□IMAG:□"F5.2)	REAL:□□1.00□□□IMAG:□□2.00

End of Chapter

Chapter 7
Specification Statements

Specification statements are nonexecutable statements that provide the FORTRAN 5 compiler with information about storage allocation, data types and dimensions of variables and arrays, and other basic program information. Specification statements include the COMMON, COMPILER, DATA, DIMENSION, EQUIVALENCE, EXTERNAL, IMPLICIT, OVERLAY, PARAMETER, STATIC, and Type-statements (INTEGER, REAL, DOUBLE PRECISION, COMPLEX, DOUBLE PRECISION COMPLEX, and LOGICAL).

You may place specification statements anywhere in your source program and in any order. But, do not label them.

DIMENSION

Names arrays and specifies their dimensions.

Format

DIMENSION a_1 (d_1) *[/clist$_1$ /],...,* a_n (d_n) *[/clist$_n$ /]*

Where:

a	is the name of an array (declarator name).
d	is a dimension declarator.
clist	is a list of values to which you want array elements initialized.

Statement Execution

You can specify an array's dimensions only once, using either a DIMENSION, COMMON, STATIC, or Type-statement. Declaring an array allocates a storage location for each element in the array(s) it names. The data type of the array determines the size of the element.

Array dimensioning information may consist of integer constants or expressions involving constants. It may also consist of expressions involving dummy arguments if, and only if, the array name is a dummy argument name.

When using the *clist* option, list the values as single numbers separated by commas, enclosed in slashes. If you want to initialize more than one element to the same value, indicate the number of array elements to be affected (must be sequential), then the value to initialize them to, separated by an asterisk. For example, DIMENSION ARAY(4)/4*3.0/ initializes the 4 elements of array ARAY to the value 3.

DIMENSION (continued)

Declarators

Each a(d) is an array declarator, where a is the symbolic name that identifies an array, and d, the dimension declarator, gives the size or structure of the array. The form of the dimension declarator is:

[ld:] ud

where *ld* is the lower dimension bound and ud is the upper dimension bound. (Note that d may consist of more than one such declarator.) Together they are called a dimension bound expression. If you give both bounds, separate them by a colon. If you specify only one number as the declarator, it represents the upper bound; the lower dimension bound is 1. For example, DIMENSION A (5,6) is equivalent to DIMENSION A(1:5,1:6). Note that you must separate dimension bounds by commas.

FORTRAN 5 calculates the number of storage locations to allocate to an array by taking the product of the dimension declarators' sizes. For example, DIMENSION AMY(4,6,6) illustrates an array containing 144 elements. DIMENSION FRED(0:3,-1:2) illustrates an array containing 16 elements.

Constant and Adjustable Dimensions

If each dimension bound expression in an array declarator is a constant expression, we call the declarator a *constant array declarator.* For example, DIMENSION MARIE(4,5,2:6) is a constant array declarator.

On the other hand, if a dimension bound expression in an array declarator contains a variable, we call this declarator an *adjustable array declarator.* Neither a constant nor adjustable array declarator can include array elements or function references.

The next distinction to make is between an actual array declarator and a dummy array declarator. An *actual array declarator* is simply any constant array declarator whose array name is not a dummy argument; you may use an actual array declarator in a DIMENSION, Type-, STATIC, or COMMON statement. A *dummy array declarator* is any constant or adjustable array declarator whose array name is a dummy argument. You may specify a dummy array declarator only in a DIMENSION or Type-statement and only in a function or subroutine subprogram.

The advantage of using an adjustable array declarator is that it enables a subprogram to process a different set of data with each execution. To accomplish this, you must define one or more arrays with actual array declarators in the program unit that references the subprogram. When control transfers to the subprogram containing the adjustable DIMENSION statement, you pass the actual array name and actual dimensioning information (for the current execution of the subprogram) as arguments of the function reference or CALL statement in the subprogram. Actual values replace the array name and adjustable dimensions in the DIMENSION statement. Thus, you may pass different values for each execution of the subprogram. The size of the adjustable array in the subprogram must not exceed the actual array size specified in the program unit that references the subprogram.

Examples

```
DIMENSION MARY (3,5,2,2)

DIMENSION X1 (0:2,1:5)

DIMENSION A(5), B(4,1:6),C(9)/6*0.0/

SUBROUTINE R(A,I,J,K,B)
DIMENSION A(I,J,K),B(20)
```

Type-Statements

Specify the type of data that can be assigned to a symbolic name.

Format

type v_1 *[/clist$_1$ /],...,* v_n *[/clist$_n$ /]*

Where:

type	means INTEGER, REAL, DOUBLE PRECISION, COMPLEX, DOUBLE PRECISION COMPLEX, or LOGICAL.
v	is the name of a variable, array, function, dummy argument, or an array name with dimensioning information.
clist	is a value (or list of values in the case of an array) to which you want the immediately preceding v initialized.

Statement Execution

You may explicitly specify the data type of a symbolic name only once. You may use a Type-statement to override the specifications of an IMPLICIT statement or the default data typing; i.e., names beginning with I, J, K, L, M, or N are type integer and all others are type real.

You may dimension an array in a Type-statement exactly as you would in a DIMENSION statement.

When using the *clist* option, list the values as single numbers separated by commas, enclosed in slashes. If you want to initialize more than one element to the same value, indicate the number of array elements to be affected (must be sequential), then the value to initialize them to, separated by an asterisk. For example, INTEGER A(7)/6*1,3/ initializes the first 6 consecutive elements of array A to 1 and the seventh element to 3.

Examples

```
INTEGER X,Y

REAL MEAN,MEDIAN

DOUBLE PRECISION DBL,LONG (10)/10*)0.D0/

COMPLEX IMAG /(2.1,-3.5)/

LOGICAL QUES, WHICH (1:9,2:9)
```

IMPLICIT

Changes or confirms the default data type of symbolic names.

Format

IMPLICIT $type_1$ ($list_1$) *[,...,type$_n$ (list$_n$)]*

Where:

type	means REAL, DOUBLE PRECISION, COMPLEX, DOUBLE PRECISION COMPLEX, or LOGICAL.
list	is a list of single letters, a range of letters, or both.

Statement Execution

In the format above, list specifies that symbolic names beginning with the letter(s) given will have the data type that precedes the left parenthesis of list. Using a range of letters in list is identical to listing each letter individually, separated by commas. You indicate a range by the first and last letters of the range separated by a minus sign. There is no illegal range of letters in this statement; for example, the range R-B is equivalent to the range B-R.

An IMPLICIT statement affects the data type of all variables, arrays, or functions (except intrinsic functions).

You may override the effect of an IMPLICIT statement by specifying FORTRAN 5 components that begin with the same letters in a Type-statement.

Examples

```
IMPLICIT INTEGER (S,T,V),REAL (H,I,L)

IMPLICIT DOUBLE PRECISION (A,Q-M)

IMPLICIT COMPLEX (R),REAL (A,B,C,D),LOGICAL (E-G)

;DEFAULT
IMPLICIT REAL(A-H),INTEGER(I-N),REAL(O-Z)

;SAME AS PREVIOUS EXAMPLE
IMPLICIT REAL(A-Z),INTEGER(I-N)
```

COMMON

Allocates an area of data storage accessible to multiple program units, and names the variables and arrays which are to reside in this area.

Format

COMMON *[/[block$_1$]]* list$_1$ *[... /[block$_n$]/list$_n$]*

Where:

block	is the name of a common block.
list	is a list of names of variables and arrays, or arrays with dimensioning information.

Statement Execution

A *common block* is a storage area shared by two or more program units. It allocates the specified variables and arrays in the order in which you specify them in the COMMON statement(s). A common block name takes the form of a variable name; its permissable length is 1 to 5 characters. A COMMON block name may appear more than once in a COMMON statement and more than once in a program unit. The given block simply continues to allocate variables and arrays in the order in which you give them. For example, the statements

```
COMMON /ARAY/A,B,C/GAF/PINT,QT
COMMON /GAF/E,K,Q/ARAY/MOND,SATD
```

allocate the entities of the common blocks in the same order as:

```
COMMON /ARAY/A,B,C,MOND,SATD/GAF/PINT,QT,E,K,Q
or
COMMON /GAF/PINT,QT,E,K,Q/ARAY/A,B,C,MOND,SATD
```

You can increase the size of a common storage area by using either another COMMON statement or an EQUIVALENCE statement, but you cannot extend the area backwards (for more information see the EQUIVALENCE statement).

Give only names of variables and arrays in a common block list. Dummy argument names are not legal.

You may use either a DIMENSION, Type-, or COMMON statement to dimension an array, but you may dimension the same array only once within a single program unit. An example of dimensioned arrays in a COMMON statement is:

```
COMMON A,B,C(2,4)/DAR/D(3,4)
```

There are two main reasons for using common storage. The first is to conserve storage space by allowing several program units to allocate storage only once for commonly used variables and arrays. Secondly, common storage allows you to implicitly transfer arguments between a calling program and a subprogram.

Named and Blank Common Storage

If a common block name precedes a list of variables and arrays in a COMMON statement, it is called *named common.* FORTRAN 5 stores the variables and arrays in a common storage area, labeled with the common block name given.

If a common block name does not precede a list of variables and arrays, FORTRAN 5 places the variables and arrays in an unlabeled common area, called *blank common.* To indicate blank common, write two successive slashes preceding the variables and arrays to be affected; if it is the first item of the COMMON statement list, however, slashes are not necessary. For example, the statement COMMON A,B/DIG/R(2,3)//F,G stores A, B, F, and G in blank common.

DGC's FORTRAN 5 allows you to data-initialize entities in blank common as well as named common.

Examples

1. This example references array IA (in program unit 1) as array IX (in program unit 2).

Unit 1

```
DOUBLE PRECISION D
COMMON /BLK1/D,IA(10,10)
```

Unit 2

```
COMMON /BLK1/X(2),IX(100)
```

To access IA you must use X(2) to leave four dummy locations (one word each) in the common area. These locations correspond to those occupied by the variable D. Storage in common area BLK1 is as shown in Figure 7-1.

2. This example transfers arguments between a calling program and a subprogram.

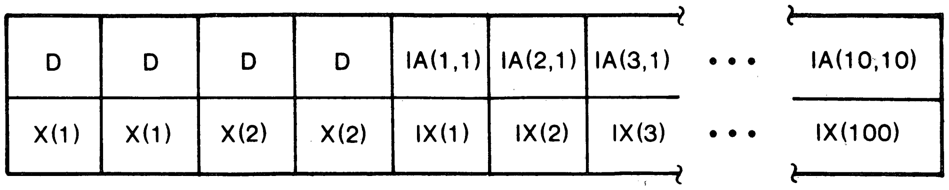
Calling Program

```
COMMON FRANC,P/MONEY/CHGE,J
.
.
.
CALL SUBR1
```

Subprogram

```
SUBROUTINE SUBR1
COMMON /MONEY/RANK,ILIST//PENCE,H
.
.
.
RETURN
END
```

The subprogram associates RANK and ILIST with CHGE and J in the common area labeled MONEY (note that the types correspond). In addition, PENCE and H associate with FRANC and P in the blank common area.



SD-01196

Figure 7-1. Storage in Common Area BLK1

STATIC

Places specified variables and arrays in a fixed area in memory rather than on the runtime stack.

Format

STATIC v_1 *[(d_1)] [/$clist_1$ /],...,* v_n *[(d_n)] [/$clist_n$ /]*

Where:

v	is a variable or array name.
d	is a dimension declarator for an array.
clist	is a value (or list of values in the case of an array) to which you want the immediately preceding v initialized.

Statement Execution

The STATIC statement initializes variables and array elements to zero unless you data-initialize them using the *clist* option. The variables and/or array elements retain the values they last had; these values are available upon subsequent re-entry to the subprogram. This is not true of stack variables.

When using the *clist* option, list the values as single numbers separated by commas, enclosed in slashes. If you want to initialize more than one element to the same value, indicate the number of array elements to be affected (must be sequential), then the value to initialize them to, separated by an asterisk. For example, STATIC A(7)/6*1.0,3.0/ initializes the first 6 consecutive elements of array A to the value 1 and the seventh element to the value 3.

The STATIC(NOSTACK) option described for the COMPILER statement differs in that it provides static storage for all non-COMMON variables and arrays (see the COMPILER Statement).

Static storage variables, unlike variables in common, are not position-dependent and the instructions given for EQUIVALENCEing common variables do not apply to static variables.

Examples

```
STATIC A(5,6,5),B

STATIC C(20),D/1.1/

STATIC R(12)/3*0.0,7*1.0,5.0,4.0/
```

EQUIVALENCE

Associates two or more entities in the same storage area.

Format

EQUIVALENCE ($list_1$) *[, ($list_2$),..., ($list_n$)]*

Where:

list	is a list of names of variables, arrays, and/or array elements having constant subscripts.

Statement Execution

Each parenthesized list contains entities to share the same storage locations. You must specify at least two entities per list.

The EQUIVALENCE statement enables you to do the following:

1. Control the allocation of storage for data within a program unit. By forcing data to share the same storage locations, you reduce the total storage needed by your program.

2. Give aliases to variables.

3. Access data of one type as data of a different type. You are able to access portions of numbers that you normally are unable to, such as the imaginary part of a complex number. For example:

   ```
   COMPLEX C(10)
   REAL R(20)
   COMMON C
   EQUIVALENCE (C,R)
   ```

   A complex number consists of two parts, the real and the imaginary. When you EQUIVALENCE array R to array C, every other element of R accesses an imaginary number.

Neither the lengths nor the data types of the entities you EQUIVALENCE have to match. Type conversion does not occur. If you EQUIVALENCE an array element and a variable, the variable does not take on the properties of the array element or vice versa. Problems result if you EQUIVALENCE a real variable and a double precision variable: the real variable shares only the first half of the storage unit which the double precision variable occupies.

EQUIVALENCE statements cannot contain dummy argument or function names.

You may reference array elements by any of the following methods:

1. A Complete Subscript. The number of entities in the subscript is equal to the number of dimensions in the array. For example, A(2,3,4) indicates an array A having three dimensions.

2. A Single Subscript. This indicates the linear position of an element in an array. For example, A(12) indicates the 12th element of array A. You may use this method with all arrays, including multiply-dimensioned ones.

3. Nonsubscripted Array. This indicates the first element of an array.

Because FORTRAN 5 stores arrays in a predetermined order, when you EQUIVALENCE an element of one array to an element of another array you are implicitly EQUIVALENCEing all other elements of the two arrays. For example, the statements

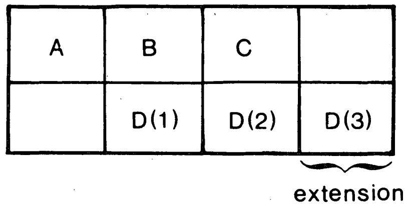
```
DIMENSION A(5,4),B(20)
EQUIVALENCE (A(1,1),B(1))
```

not only indicate that array elements A(1,1) and B(1) share the same storage locations, but that A(2,1) and B(2) (etc.) also share the same storage locations.

If a given entity is part of a common block, you cannot EQUIVALENCE it to another entity in that block or to an entity in another common block. You can EQUIVALENCE an array element in the program unit to an entity of a common block and it may extend that block only if it adds storage locations to the end of the common block. For example, the statements

```
COMMON A,B,C
DIMENSION D(3)
EQUIVALENCE (B,D(1))
```

extend the common block in the following manner:

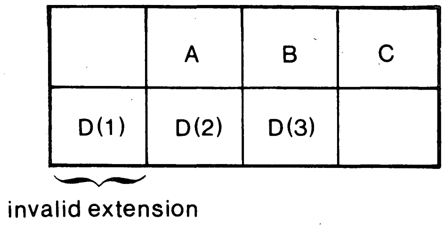


SD-01195

You cannot EQUIVALENCE an array element to an entity in a common block if it would extend the block by adding array elements before the block's beginning. The statements

```
COMMON A,B,C
DIMENSION D(3)
EQIVALENCE (B,D(3))
```

attempt to extend the common block invalidly:



SD-01195

Examples

```
DIMENSION R(10),S(15,15),T(5,10,10)
EQUIVALENCE (Q,R(2),S(2,4)),(T(2,8,4),V)
```

Q,R(2) and S(2,4) share the same storage locations. T(2,8,4) and V share the same storage locations.

Implicitly R(3) and S(3,4) (etc.) share the same storage locations. In this case, EQUIVALENCEing R(3) explicitly with any element of S other than S(3,4) would be invalid.

In the first part of your program, you use the array X. You then use array Y and Z, but no longer need array X.

```
DIMENSION X(40,40),Y(10,10),Z(4,5)
EQUIVALENCE (X(1),Y(1)),(X(101),Z(1))
```

You do not need array X, so you EQUIVALENCE array Y and Z to use the same storage locations as X. You do not want array Y and array Z to overlap: Because array Y takes up the first 100 array elements of X, array Z must begin at array element 101 or later. Note that only 120 of the 160 elements of array X are used by arrays X and Z.

PARAMETER

Assigns a symbolic name to a constant or to an expression.

Format

PARAMETER $v_1 = e_1$ *[*$, v_2 = e_2, \ldots, v_n = e_n$ *]*

Where:

v	is a symbolic name.
e	is a numeric or logical expression, or a Hollerith or string constant.

Statement Execution

FORTRAN 5 associates each symbolic name, v, with the value given by the e that follows the equal sign (e may not contain an exponentiation operation). Once you assign a symbolic name to a constant, you may then use that name to replace the constant anywhere in the program.

Use of the PARAMETER statement is a time-efficient coding method. It can make a program easier to read and easier to maintain. For example, a particular constant might appear in several places in a program. This constant might be a unit number, an array bound, or a constant value to be used in calculations. You may improve program maintenance by replacing this constant with a meaningful name. If for some reason you must change the value of this constant or increase its precision, you will see that it is easier to change one PARAMETER statement than all the occurrences of the constant throughout the program. A section of the hypothetical program here might look like this:

```
PARAMETER IN=17,LPT=12,N=25
DIMENSION A(N),B(2,N),C(N,N)
.
.
.
OPEN IN, 'DATABASE'
READ (IN,10) A
READ (IN,15) B
.
.
.
WRITE (LPT,20) (A(I),B(1,I),B(2,I),I=1,N)
```

Each symbolic name (v) may appear in only one PARAMETER statement within the same program unit. Because it is a constant or expression, you cannot redefine its value during program execution.

The symbolic name of a constant may appear as part of an expression. However, it must not be part of a FORMAT statement and must not appear in a Type-statement except in dimension bound expressions within array declarators.

The first letter of the symbolic name of a constant does not specify its type; it has no type.

Examples

Ways to use the PARAMETER statement:

```
PARAMETER PI=3.141592653,Q1=.1731D-7

PARAMETER J=1,K=4-J,BETA=J/2*K*K
```

Ways to use a parameter:

```
PARAMETER K=8
.
.
COMMON /COMLABEL/C1(K),C2(K)
DATA C1/K*0.0/
DO 5 I=1,K
C1(1)=5*K
;NOTE THAT SINCE K IS NOT A VARIABLE
;IT MUST NOT BE MODIFIED BY SUBROUT
CALL SUBROUT(C1,AB,K)
```

EXTERNAL

Allows you to use an externally defined subprogram name or overlay name as an argument.

Format

EXTERNAL s_1 *[,s_2, ...,s_n]*

Where:

s   is the name of a function subprogram, subroutine subprogram, or overlay.

Statement Execution

The EXTERNAL statement causes the compiler to recognize a name as a subprogram or overlay name rather than a variable or array name. You must use the EXTERNAL statement with a function, subroutine, or overlay name that appears as an argument in a function reference or in a CALL statement. However, using the name of a generic function in an EXTERNAL statement disables generic function selection on that name. For example,

```
EXTERNAL SIN
CALL F(SIN)
Y=SIN(X)
```

causes SIN to lose its generic selection properties. To pass the name of a generic function to a subprogram, a function reference must appear previous to the calling statement. For example,

```
EXTERNAL SIN
Y=SIN(X)
CALL F(SIN)
```

A symbolic name may appear only once in an EXTERNAL statement in a program unit.

You cannot use a statement function name in an EXTERNAL statement.

Example

Subroutine MULT is called with real function NROOT as the last argument.

```
REAL NROOT
EXTERNAL NROOT
.
C=NROOT(D)
.
CALL MULT (A,B,NROOT)
.
SUBROUTINE MULT(X,Y,Q)
.
.
.
Y=Q(X)
RETURN
END
```

DATA

Defines initial values for variables and array elements.

Format

DATA vlist$_1$ /clist$_1$ /*[,..., vlist$_n$ /clist$_n$ /]*

Where:

vlist	is a list of variables.
clist	is a list of constants.

Statement Execution

FORTRAN 5 pairs the variable and constant lists in a DATA statement. A *variable list* is of the same form as an I/O list and can contain variables, arrays, array elements, and DO-implied lists enclosed in parentheses. A *constant list* may include numeric constants (optionally signed), Hollerith and string constants, logical constants, and parameters.

You may precede a constant by a repeat count of the form:

```
n*constant
```

where n is the number of times FORTRAN 5 assigns the constant value to variables in the corresponding vlist. If n is an expression, you must enclose it in parentheses. For example,

```
DATA R/(HB-LB+1)*0.0/
```

There is a positional correspondence between a variable list and its associated constant list: the number of entities defined by the variable list must be the same as the number of entities defined by the constant list. If a variable list is longer than its corresponding constant list, FORTRAN 5 will signal an error. If a constant list is longer than its corresponding variable list, FORTRAN 5 signals an error. For example, the following DATA statements generate an error:

```
DIMENSION X(11)
DATA X/10*1.0/

DIMENSION X(11)
DATA X/10*1.0,2.0,3.0/
```

The order and grouping of variables and constant lists is unimportant as long as the correct variable/constant association is not altered. For example, the following three statements are identical in their effects:

```
DATA B,P(1,1),L,S,J/2*1.0,.TRUE.,'PRIC','E'/
DATA B,P(1,1)/2*1.0/,L/.TRUE./,S,J/'PRIC','E'/
DATA L,B,S/.TRUE.,1.0,'PRIC'/,P(1,1),J/1.0,'E'/
```

If the data type of a variable and its corresponding constant do not match, FORTRAN 5 converts the constant to the variable's type according to the rules for assignment (see Chapter 3).

You can initialize any variable to a Hollerith or string constant. Each character of a Hollerith or string constant occupies one byte, i.e., there are two characters per 16-bit word. If you specify a Hollerith or string constant that is shorter than its corresponding variable, FORTRAN 5 left justifies and blank fills the constant. If the Hollerith or string constant is longer than its corresponding variable, FORTRAN 5 truncates the extra right-hand characters.

If a variable list contains a DO-implied list, the control variable must be an integer variable and the index parameters must be expressions involving integer constants or index variables of an outer DO-implied loop. If a variable list contains subscripted variables, the subscripts must be either integer constants or index variables.

If you want to initialize all elements of the array, the variable list may contain an unsubscripted array name. If you do not indicate enough values in the corresponding constant list to initialize all elements of the array, FORTRAN 5 will signal an error.

You must enclose the components of a complex number in parentheses. For example, given the complex variable C, it is incorrect to initialize C with two real constants, such as DATA C/1.0,1.0/. When the compiler encounters this statement, it selects the first constant as the initial value of C and promotes the constant to the correct complex form, (1.0,0.0). The compiler then encounters the second constant, 1.0, sees that there is no corresponding variable, and signals an error. Thus, this statement should be written as DATA C/(1.0,1.0)/.

Two entities of a variable list cannot split a Hollerith or string constant. For example,

```
DATA I,J/"ABCD"/
```

is incorrect. Because each integer can only store two ASCII characters, you can correctly write the above statement as:

```
DATA I,J/"AB","CD"/
```

DGC's FORTRAN 5 allows you to use a DATA statement to initialize entities in blank and named COMMON in any program unit.

Examples

Initialize A to 7.1 and the 8 elements of R to 3.3.

```
DIMENSION R(8)
DATA A,R/7.1,8*3.3/
```

Initialize the 9 elements of A to 0.0 and the first 5 elements of R to 9.

```
DIMENSION A(3,3),R(10)
PARAMETER X=9.0
DATA A,(R(I),I=1,5)/9*0.0,5*X/
```

You must enclose the DO-implied list in parentheses.

Initialize the third element of R to .0007 and store the string constant "SIZE□WAS" in A and B.

```
DIMENSION R(10)
DATA R(3),A,B/7E-4,"SIZE","□WAS"/
```

COMPILER

Permits you to specify STATIC, NOSTACK, FREE, and/or DOUBLE PRECISION as compile-time options.

Format

COMPILER option$_1$ *[,option$_2$, option$_3$]*

Where:

option means STATIC, NOSTACK, FREE, or DOUBLE PRECISION.

Statement Execution

The STATIC option places all non-COMMON variables and arrays in a fixed area in memory rather than on the runtime stack. It initializes all static variables to zero at load time, unless you data-initialize them. You may retrieve the values of static variables within a subprogram upon re-entry to the subprogram for the second and subsequent times. NOSTACK is simply an alternate name for STATIC.

An example of when to use the STATIC option is: if your program compiles successfully (using the DGC FORTRAN 5 compiler) without using the STATIC option, but does not run correctly. Use the STATIC option to see if you were expecting either memory to be zeroed or variables to remain unchanged across successive subprogram invocations. If so, you should recode the program to generate the most efficient code. First use the STATIC statement to place only the necessary variables in static storage. Secondly, recompile the program without the STATIC option.

If you specify the FREE option, all READ and WRITE statements of the form

READ (unit *[,options]*) *[list]*
WRITE (unit *[,options]*) *[list]*

transfer free-form ASCII data rather than binary data. (However, this has no effect on READ BINARY or WRITE BINARY.)

The DOUBLE PRECISION option forces promotion of all variables and constants of type real to type double precision for compilation. It also forces all complex variables and constants to type double precision complex. In summary, the DOUBLE PRECISION option does the following:

1. Overrides any REAL or COMPLEX statements and any data typing to real or complex using the IMPLICIT statement.

2. Forces all real constants to double precision.

3. Recognizes calls generated to the appropriate double precision functions.

You must specify at least one option when using the COMPILER statement. If you specify more than one, the order in which you state the options is of no importance, but you must separate them by commas.

Example

```
COMPILER FREE,STATIC
```

OVERLAY

Specifies a subprogram as an overlay and names it.

Format

```
OVERLAY name
```

Where:

name is a name to identify the current overlay.

Statement Execution

Overlays are portions of a program that alternately share areas of memory during program execution. You must assign each overlay a name by using an OVERLAY statement.

An OVERLAY statement must be a statement in one of the program units belonging to the overlay. If an overlay contains more than one subprogram, you can use an OVERLAY statement in each subprogram to associate a name with that overlay. You can reference the overlay by using any of the given names. You may not specify the same name in more than one OVERLAY statement.

An overlay name is an external symbol, like the name of a subprogram. Therefore, you must declare each overlay name in an EXTERNAL statement in any program unit that references the overlay. The permissible length of an overlay name is 1 to 5 characters. You reference overlay names when loading or releasing overlays using FORTRAN 5 calls to runtime routines.

Example

You want FORTRAN 5 to compile program units MAIN, A, B, C, D, E, F, and G into separate, relocatable binary files. Program units A, B, D, and F contain OVERLAY statements as follows:

Unit	Statement
A	OVERLAY OV1
B	OVERLAY OV2
D	OVERLAY OV3
F	OVERLAY OV4

Program unit MAIN contains the EXTERNAL statement:

```
EXTERNAL OV1,OV2,OV3,OV4.
```

Program unit MAIN can call any of the overlays because it contains an EXTERNAL statement that specifies all of them.

End of Chapter

Chapter 8
Subprograms

In this chapter we discuss three types of subprograms: function subprograms, subroutine subprograms, and block data subprograms.

Functions

A *function* is a procedure that you reference by specifying its name in an expression and following it by a parenthesized list of arguments. A function returns a single value to the point where it was invoked. A function, other than a statement function, has a data type associated with its name.

No function reference can appear on the left-hand side of an equal sign except for a statement function reference (under certain conditions), a FLD function reference, and a BYTE function reference.

There are three types of FORTRAN 5 functions:

1. Statement Functions -- single statements written and compiled as part of a program unit;
2. Function Subprograms -- externally compiled and user written;
3. Library Functions -- supplied with the FORTRAN 5 compiler.

Statement Functions

Define a function in a single statement that is internal to the current program unit.

Format

func (arg$_1$, arg$_2$, ...,arg$_n$) = exp

Where:

func	is a function name.
arg	is a dummy argument name.
exp	is an expression.

Execution

We call the above statement the statement function definition; it is a nonexecutable statement and simply defines the function. It must precede the function reference (in the same program unit) which is of the form:

func (arg$_1$, arg$_2$, ...,arg$_n$)

where func is the name of the statement function and arg is an actual argument.

FORTRAN 5 saves the arithmetic expression on the right-hand side of the statement function and associates it with the function name on the left-hand side.

To illustrate the actions the compiler takes when it encounters a statement function reference, take the following example:

```
;STATEMENT FUNCTION DEFINITION
AVG(X,Y,Z)=(X+Y+Z)/3.
.
.
;FUNCTION REFERENCE
OUTCM=AVG(TIME1,TIME2,TIME3)
```

Statement Functions (continued)

The following occurs at compile time:

1. The compiler encounters the function reference and associates the reference's actual arguments with the dummy arguments of the companion statement function definition: it associates actual argument TIME1 with dummy argument X, TIME2 with Y, and TIME3 with Z.

2. A substitution of the actual arguments for the dummy arguments takes place on the right-hand side of the equal sign in the statement function definition: the expression (TIME1+TIME2+TIME3)/3. replaces (X+Y+Z)/3. FORTRAN 5 evaluates the expression and returns the result to the point of invocation.

Actual arguments and dummy arguments must agree in number and order. Dummy arguments have no type. You must assign values to the actual arguments before the statement function reference is evaluated.

A statement function has no data type of its own. FORTRAN 5 determines the data type of its result by the data type of the expression resulting from the replacement of the dummy arguments by the actual arguments. The function reference substitutes actual argument values for the statement function definition's dummy arguments; FORTRAN 5 evaluates the expression and then assigns the expression to the function name.

A statement function definition is primarily a short, concise way of defining and referencing a function. For example, given the function reference X=B(2,3), you can write a corresponding function subprogram as:

```
FUNCTION B(I,J)
COMMON A(100)
B=A((I-1)*10+J)
RETURN
END
```

However, you can define the same function more simply by using the statement function:

```
;ASSUMING A IS ALREADY DIMENSIONED
B(I,J)=A((I-1)*10+J)
```

Unlike other function references, a statement function reference can appear on the left-hand side of an equal sign. It may do so only after the statement function definition, and only if the function expands to one of the following:

- a variable name
- an array element name
- a FLD reference
- a BYTE reference

The name of a statement function is internal to the program unit and cannot appear in an EXTERNAL statement.

Example

Finding the roots of a quadratic equation.

Statement Function

```
ROOT(A,B,C)=(-B+SQRT(B**2-4.*A*C))/2.*A
```

Function Reference

```
VAL=ROOT(D(6),122.6,(X-Y))+Z**3
```

The compiler substitutes D(6) for A, 122.6 for B, and (X-Y) for C. The expression then becomes:

```
(-122.6 + SQRT(122.6 ** 2-4. * D(6) * (X-Y))) / 2. * D(6)
```

When FORTRAN 5 evaluates this expression at runtime, it returns the value to the function reference, adds it to Z**3, and assigns the result to VAL.

Function Subprograms

A *function subprogram,* generally referred to as a *function,* is an external program unit that consists of a series of statements defining a procedure. It is invoked by a function reference of the form:

func(arg_1, arg_2, ...,arg_n)

where func is the name of the external function and arg is an actual argument. The compiler will not recognize a statement as a function reference unless you enclose the actual arguments in parentheses.

The function should begin with a FUNCTION statement (although it may appear anywhere within the subprogram), end logically with a RETURN statement, and end physically with an END statement. Both RETURN and END statements pass control back to the function reference in the calling program unit, unless you route a return through a dummy argument (see the RETURN statement).

The function returns the function name's value to the function reference. You must assign a value to the function name before reaching a RETURN or END statement in the function.

In addition, the function reference in the calling program unit supplies actual arguments. FORTRAN 5 associates the values of these actual arguments with dummy arguments in the FUNCTION statement. The function may modify the values of these arguments.

A function cannot contain statements defining other program units, i.e., SUBROUTINE, BLOCK DATA, or other FUNCTION statements.

FUNCTION

Begins and defines a function subprogram.

Format

[type] FUNCTION name(arg_1, arg_2, ...,arg_n)

Where:

type	is INTEGER, REAL, COMPLEX, DOUBLE PRECISION, DOUBLE PRECISION COMPLEX, or LOGICAL.
name	is the name of the function.
arg	is a dummy argument.

Statement Execution

The FUNCTION statement should be the first statement of a function subprogram, but may appear anywhere within the subprogram. The compiler ignores any label on a FUNCTION statement. The function executes its statements and, upon reaching a RETURN statement, returns a value to the function reference. If you do not supply a RETURN statement, FORTRAN 5 automatically generates one immediately preceding the END statement.

You can specify the data type of the returned value in *type* of the FUNCTION statement. If you do not indicate a data type, the function is typed according to the name rule.

The permissible length of a function name is 1 to 5 characters. You must assign a value to the function name at least once before the function returns control to the function reference. For example, a function beginning with the statement FUNCTION CAN(Y) might also contain a statement such as CAN=5. to assign a value to the function.

The dummy arguments of a FUNCTION statement must correspond in order, number, and type with the actual arguments of the function reference that invoked it. Failure to do so is a common source of obscure errors in FORTRAN 5 programs. Each dummy argument may be a variable name, an array name, an external subprogram name, or $. The $ symbol corresponds to an actual argument that is a statement label through which you can route an alternate return (see the RETURN statement). You cannot include these dummy argument names in EQUIVALENCE, COMMON, STATIC, or DATA statements that are within the subroutine. Execution of a RETURN or END statement passes the value of an argument back to the function reference in the calling program unit.

A FUNCTION statement can execute normal or alternate RETURN statements (see the RETURN statement).

Examples

Function Statements

```
FUNCTION MARGO(L)

INTEGER FUNCTION ROYCE(I,J,M)

DOUBLE PRECISION FUNCTION FLAG(X,Y)
```

Function Subprogram

```
    ;FUNCTION REFERENCE IN MAIN PROGRAM
    X=SWITCH(A)

    ;FUNCTION SUBPROGRAM
    FUNCTION SWITCH(X)
    IF(X.LE.0.) GO TO 5
    SWITCH=1.
    RETURN
5   SWITCH=0.
    RETURN
    END
```

Arguments of Function and Subroutine Subprograms

The execution of a function or subroutine reference establishes an association between the actual arguments of that reference and the dummy arguments of the subprogram called. This association is valid only if the arguments agree in order, number, and type. The one exception is that a statement function's associated arguments need only agree in number and order.

An actual argument cannot be the name of a statement function defined in the program unit containing the reference. The actual argument's type must agree with the type of its associated dummy argument except when the actual argument is a subprogram name or when it is an argument in a statement function.

A dummy argument name of type integer may appear in an adjustable dimension in a dummy array declarator. It cannot appear in EQUIVALENCE, DATA, PARAMETER, or COMMON statements, except as a common block name.

An actual argument you associate with a dummy argument that is a variable name must be a variable name, an array element name, or an expression. If it is an expression, FORTRAN 5 evaluates it before argument association occurs.

FUNCTION (continued)

An actual argument you associate with a dummy argument that is an array name should be an array name or an array element name. If the actual argument is an array name, the size of the dummy array must be less than or equal to the size of the actual array. If the actual argument is an array element name, FORTRAN 5 evaluates its subscript just before argument association occurs. The length of the dummy array must be less than or equal to the length of the portion of the array that begins with the designated element. The value of the subscript remains constant as long as this argument association occurs, despite possible changes in the value of the variables making up the subscript.

An actual argument associated with a dummy argument used as a function reference must be a function name. You must not assign a value to such a dummy argument in the function.

A dummy argument that is referenced as a subroutine name must be associated with a subroutine name. It must not appear in a Type-statement and you must not reference it as a function. You cannot assign a value to the dummy argument in the subroutine.

Subroutine Subprograms

A *subroutine subprogram,* generally referred to as a *subroutine,* is a procedure external to the main program unit. A CALL statement in the calling program unit references it.

A subroutine should begin with a SUBROUTINE statement (although it may appear anywhere within the subroutine), end logically with a RETURN statement, and end physically with an END statement. Both RETURN and END statements pass control back to the calling program unit.

The CALL statement in the calling program unit supplies actual arguments whose values FORTRAN 5 associates with dummy arguments in the SUBROUTINE statement. The subroutine may modify the values of these actual arguments.

Control returns to the statement following the referencing CALL statement, unless you route a return through a dummy argument (see the RETURN statement).

A subroutine cannot contain statements defining other program units, i.e., FUNCTION, BLOCK DATA, or other SUBROUTINE statements.

SUBROUTINE

Begins and defines a subroutine.

Format

SUBROUTINE name *[(arg$_1$, arg$_2$, ...,arg$_n$)]*

Where:

name	is the name of a subroutine.
arg	is a dummy argument.

Statement Execution

The SUBROUTINE statement should be the first statement of the subroutine subprogram, but may appear anywhere within the subroutine. The permissible length of a subroutine name is 1 to 5 characters.

Each dummy argument may be a variable name, an array name, a subprogram name (function or subroutine), or $. The $ symbol corresponds to an actual argument that is a statement label through which you can route an alternate return (see the RETURN statement). You cannot include a dummy argument name in EQUIVALENCE, COMMON, STATIC, or DATA statements that are within the subroutine.

The subroutine may assign actual values to its dummy arguments and return these values to the calling program unit.

Control returns to the calling program unit upon execution of a RETURN statement. If you do not supply a RETURN statement, one is implicitly generated immediately preceding the END statement.

Example

SUBROUTINE Statements

```
SUBROUTINE ELECT(X,YARD,$,J)

SUBROUTINE NAME9
```

Subroutine Subprogram

```
      SUBROUTINE REV(ARRAY,K1,K2)
      DIMENSION ARRAY(100)
      K12=K1+K2
      MID=K12/2
      DO 50 K=K1,MID
      J=K12-K
C     USE TEMPORARY TO REVERSE ELEMENTS OF
C     ARRAY
      TEMP=A(K)
      A(K)=A(J)
      A(K)=TEMP
50    CONTINUE
      RETURN
      END
```

CALL

Invokes a subroutine, transferring control from one program unit to another.

Format

CALL name *[(arg$_1$, arg$_2$, ...,arg$_n$)]*

Where:

name	is the name of a subroutine or a dummy variable.
arg	is an actual argument.

Statement Execution

The CALL statement invokes the designated subroutine which FORTRAN 5 then executes. When execution completes, control returns to the statement in the calling program unit following the CALL statement, unless you route a return through a dummy argument (see the RETURN statement). This return completes execution of the CALL statement.

If a CALL statement appears in a subroutine, it may directly or indirectly reference that subroutine. That is, a subroutine is allowed to call itself.

If the SUBROUTINE statement in the named subroutine contains dummy arguments, the referencing CALL statement must contain actual arguments to replace the dummy arguments. Actual arguments must agree in order, number, and type with the corresponding dummy arguments. However, you may specify typeless constants (Hollerith or string constants) as actual arguments.

Examples

```
CALL QUAD (9,Q/R,$20,R-S**2.0,X1,HALF)

CALL OPTIONS
```

RETURN

Marks the logical end of a function or subroutine and returns control from that subprogram to the calling program unit.

Format

RETURN *[v]*

Where:

v	is an integer constant or expression that specifies an alternate statement in the calling program to which control returns.

Statement Execution

A subprogram may contain one, more than one, or no RETURN statement. If there is no RETURN statement, one is implicitly generated immediately preceding the END statement.

Normal RETURN Statement

You may use a RETURN statement (no *v* specified) in both a subroutine and a function. If the RETURN statement is in a subroutine, control returns to the statement following the CALL statement in the calling program unit. If the RETURN statement is in a function, control returns to the function reference that invoked it, and FORTRAN 5 returns a value.

Alternate RETURN Statement

You may use the alternate RETURN statement, RETURN *v*, in either a function or a subroutine. It allows you to return control to any labeled statement in the calling program whose label you pass as an argument to the subprogram. The *v*, which must follow the word RETURN, can be either an integer constant or an expression (evaluated and converted to type integer if necessary). In a subroutine, *v* represents the position of a dummy argument in the SUBROUTINE statement's argument list. For example, the statement RETURN I, where I equals 5, refers to the 5th argument in the list. In a function, v-1 represents the position of a dummy argument in the FUNCTION statement's argument list (the value of the function itself is considered the first argument). For example, RETURN 5 refers to the 4th argument in the list.

If *v* has a value of zero, a value greater than the number of arguments in the dummy argument list, or if it refers to an argument that is not a statement label, FORTRAN 5 signals an error and returns control as it normally would from the subroutine or function.

RETURN (continued)

In routing an alternate RETURN, you supply a statement label as an actual argument in the CALL statement or function reference. You must precede each statement label by a dollar sign ($). In addition, you must represent each dummy argument in the SUBROUTINE or FUNCTION statement that corresponds to an actual statement label as $. For example:

Calling Program Unit

```
CALL SUBR2(A,B,$10,C,$20)
```

Subprogram Unit

```
   SUBROUTINE SUBR2(X,Y,$,Z,$)
   IF (X.LT.1.) GO TO 15
   RETURN 5
15 IF(X.GT.1.) GO TO 25
   RETURN
25 J=IFIX(X/Y*COS(X))
   ;UNLESS J=3 OR 5, THIS WILL GENERATE AN
   ;ERROR
   RETURN J
   END
```

Subroutine SUBR2 shows how FORTRAN 5 returns control to one of several statements in the calling program unit depending upon the result of its execution. If it executes the statement RETURN 5, control returns to the statement labeled 20 in the calling program unit, because $20 is the fifth element of the argument list. If it executes the statement RETURN, control returns to the statement following the CALL statement in the calling program unit. If it executes the statement RETURN J, when the value of J is 3, control returns to the statement labeled 10 in the calling program unit; if J has the value 5, control returns to the statement labeled 20 in the calling program unit; otherwise, FORTRAN 5 will signal an error.

Execution of a RETURN, a RETURN v, or an END statement in a subprogram terminates the association between the subprogram's dummy arguments and the current actual arguments. In addition, all entities within the subprogram become undefined, except for the following:

1. Entities specified by STATIC or DATA statements;

2. Entities in blank and named common blocks.

Library Functions

The FORTRAN 5 compiler supplies you with a set of library functions. You invoke a library function by specifying its symbolic name in a statement together with the arguments upon which you want it to act. For example, the statement X=ABS(SIN(X)) validly references the library functions ABS and SIN.

FORTRAN 5 assigns a generic name to each set of functions. This simplifies library function referencing because you may use the same function name with more than one type of argument. The following are the generic names of functions:

```
ACOS  ATAN   COS   SIN    TAN
ASIN  ATAN2  COSH  SINH   TANH

MAX   ABS    DIM   IMAG   CONJG
MIN   SQRT   MOD   CMPLX  REAL

EXP   LOG10  INT   FLOAT
LOG   SIGN   AINT  DFLOAT
```

When you reference a library function, you may use either the data typed name (see Tables 8-1 and 8-2) or the generic name. The argument and function name need not match in data type. However, conversion occurs if the function value and arguments differ in type. (Chapters 2 and 3 discuss mixed mode evaluation.) For example, the statements

```
DSIN (integer-argument)
SIN (double-precision-argument)
```

both return double precision results.

You may use library routines to convert the data type of an argument. In some cases, FORTRAN 5 may convert the argument's data type more than once in one function. For example:

```
CMPLX (integer)
```

FORTRAN 5 first converts the argument to type real by the function FLOAT and then converts it to a complex value by the function CMPLX.

Tables 8-1 and 8-2 list the DGC FORTRAN 5 library functions. All angular quantities are expressed in radians.

Table 8-1. Trigonometric Functions

GENERIC NAME(S)	EXPLICIT NAME(S)	MEANING	ARGUMENT TYPE	RESULT TYPE	CALLING SEQUENCE EXAMPLE
ACOS	ACOS DACOS	Arccosine	Real Double	Real Double	ACOS(arg)
ASIN	ASIN DASIN	Arcsine	Real Double	Real Double	ASIN(arg)
ATAN	ATAN DATAN	Arctangent	Real Double	Real Double	ATAN(arg)
ATAN2	ATAN2 DATAN2	Arctangent of arg_1 /arg_2; the result lies in the interval-π to π and is in same quadrant as the point (arg_2, arg_1)	Real Double	Real Double	ATAN2(arg_1, arg_2)
COS	COS DCOS CCOS DCCOS	Cosine	Real Double Complex DP Complex	Real Double Complex DP Complex	COS(arg)
COSH	COSH DCOSH CCOSH DCCSH	Hyperbolic cosine	Real Double Complex DP Complex	Real Double Complex DP Complex	COSH(arg)
SIN	SIN DSIN CSIN DCSIN	Sine	Real Double Complex DP Complex	Real Double Complex DP Complex	SIN(arg)
SINH	SINH DSINH CSINH DCSNH	Hyperbolic sine	Real Double Complex DP Complex	Real Double Complex DP Complex	SINH(arg)
TAN	TAN DTAN CTAN DCTAN	Tangent	Real Double Complex DP Complex	Real Double Complex DP Complex	TAN(arg)
TANH	TANH DTANH CTANH DCTNH	Hyperbolic tangent	Real Double Complex DP Complex	Real Double Complex DP Complex	TANH(arg)

Table 8-2. Arithmetic and Conversion Functions

GENERIC NAME(S)	EXPLICIT NAME(S)	MEANING	ARGUMENT TYPE	RESULT TYPE	CALLING SEQUENCE EXAMPLE
ABS	ABS IABS DABS	Absolute value	Real Integer Double	Real Integer Double	ABS(arg)
AINT	AINT DINT	Extract integral value	Real Double	Real Double	AINT(arg)
None	ANINT DNINT	Round to nearest integral value	Real Double	Real Double	ANINT(arg)
None	CABS DCABS	Complex modulus; SQRT(REAL(arg)**2.+ IMAG(arg)**2.)	Complex DP Complex	Real Double	CABS(arg)
None	CEIL DCEIL	Obtain least integral value >= argument	Real Double	Real Double	CEIL(arg)
CMPLX	CMPLX DCMPLX	Convert 1 or 2 real arguments to complex	Real Double	Complex DP Complex	CMPLX(arg_1 *[,arg_2]*)
CONJG	CONJG DCONJG	Obtain conjugate of Complex argument; CMPLX(REAL(arg), -IMAG(arg))	Complex DP Complex	Complex DP Complex	CONJG(arg)
None	CXDCX	Convert complex argument to double precision complex	Complex	DP Complex	CXDCX(arg)
None	DBLE	Convert real argument to double precision	Real	Double	DBLE(arg)
None	DCXCX	Convert double precision complex argument to complex	DP Complex	Complex	DCXCX(arg)
DIM	DIM IDIM DDIM	Positive difference; arg -MIN(arg_1, arg_2)	Real Integer Double	Real Integer Double	DIM(arg_1, arg_2)
EXP	EXP DEXP CEXP DCEXP	Exponential; e^{arg}	Real Double Complex DP Complex	Real Double Complex DP Complex	EXP(arg)
FLOAT DFLOAT	FLOAT DFLOAT	Convert integer argument to real	Integer Integer	Real Double	FLOAT(arg)
None	FLOOR DFLOOR	Obtain greatest integral value <= argument	Real Double	Real Double	FLOOR(arg)
None	IFIX	Convert from real to integer by truncation	Real	Integer	IFIX(arg)
IMAG	AIMAG DIMAG	Extract imaginary part of complex argument	Complex DP Complex	Real Double	AIMAG(arg)
INT	INT IDINT	Convert to integer by truncation	Real Double	Integer Integer	INT(arg)

Table 8-2. Arithmetic and Conversion Functions (continued)

GENERIC NAMES	EXPLICIT NAMES(S)	MEANING	ARGUMENT TYPE	RESULT TYPE	CALLING SEQUENCE EXAMPLE
LOG	ALOG DLOG CLOG DCLOG	Natural logarithm; $\log_e$ arg	Real Double Complex DP Complex	Real Double Complex DP Complex	ALOG(arg)
LOG10	ALOG10 DLOG10	Common logarithm; $\log_{10}$ arg	Real Double	Real Double	ALOG10(arg)
MAX	MAX0 MAX1 AMAX0 AMAX1 DMAX1	Choose the largest value	Integer Real Integer Real Double	Integer Integer Real Real Double	MAX0(arg_1, arg_2 [,...., arg_n])
MIN	MIN0 MIN1 AMIN0 AMIN1 DMIN1	Choose the smallest value	Integer Real Integer Real Double	Integer Integer Real Real Double	MIN0(arg_1, arg_2 [,...., arg_n])
MOD	MOD AMOD DMOD	Remainder; arg -INT(arg_1 /arg_2)* arg_2	Integer Real Double	Integer Real Double	MOD(arg_1, arg_2)
REAL	REAL DREAL	Extract real part of complex argument	Complex DP Complex	Real Double	REAL(arg)
SIGN	SIGN ISIGN DSIGN	Transfer sign; (Sign of arg_2)*arg_1	Real Integer Double	Real Integer Double	SIGN(arg_1, arg_2)
None	SNGL	Convert double precision argument to real by dropping least significant part	Double	Real	SNGL(arg)
SQRT	SQRT DSQRT CSQRT DCSQRT	Square root	Real Double Complex DP Complex	Real Double Complex DP Complex	SQRT(arg)

Table 8-2. Arithmetic and Conversion Functions (continued)

Notes for Tables 8-1 and 8-2

1. For *a* of type integer, INT(a) = a. For *a* of type real or double precision, there are two cases: if $|a| < 1$, INT(a) = 0; if $|a| > 1$, INT(a) is the integer whose magnitude is the largest integer that does not exceed the magnitude of *a* and whose sign is the same as the sign of *a*. For example, INT(-3.7) = -3.

   For *a* of type complex, INT(a) is the value obtained by applying the above rule to the real part of *a*.

   If *a* is of type real and is outside the range -32,768 through 32,767, INT(a) will give integer overflow, but IFIX(a) will not.

2. For *a* of type real, REAL(a) is *a*. For *a* of type integer, REAL(a) is the same as FLOAT(a). For *a* of type double precision, REAL(a) is as much precision of the significant part of *a* as a real datum can contain. For *a* of type complex, REAL(a) is the real part of *a*.

3. For *a* of type double precision, DBLE(a) is *a*. For *a* of type integer, DBLE(a) is the same as DFLOAT(a). For *a* of type complex, DBLE(a) is a double precision complex datum.

4. COMPLX in a generic function reference may have one or two arguments, which may be of type integer, real, double precision, or complex.

   For *a* of type complex, CMPLX(a) is *a*. For *a* of type integer, real, or double precision, CMPLX(a) is the complex value whose real part is REAL(a) and whose imaginary part is zero.

   CMPLX(a_1, a_2) is the complex value whose real part is REAL(a_1) and whose imaginary part is REAL(a_2).

5. The result of a complex function is the principal value.

Restrictions on the Range of Arguments and Results

Restrictions on the range of arguments and results for intrinsic functions when referenced by their specific names are as follows:

1. Remaindering: The result for MOD, AMOD, and DMOD is undefined when the value of the second argument is zero.

2. Transfer of Sign: If the value of the first argument of ISIGN, SIGN, or DSIGN is zero, the result is zero which is neither positive nor negative.

3. Square Root: The value of the argument of SQRT and DSQRT must be greater than or equal to zero. The result of CSQRT is the principal value with the real part greater than or equal to zero. When the real part of the result is zero, the imaginary part is greater than or equal to zero.

4. Logarithms: The value of the argument of ALOG, DLOG, ALOG10, and DLOG10 must be greater than zero. The value of the argument of CLOG must not be (0.0,0.0). The range of the imaginary part of the result of CLOG is: $-\pi <=$ imaginary part $<= \pi$.

5. Arcsine: The absolute value of the argument of ASIN and DASIN must be less than or equal to one. The range of the result is: $-\pi/2 <=$ result $<= \pi/2$.

6. Arccosine: The absolute value of the argument of ACOS and DACOS must be less than or equal to one. The range of the result is: $0 <=$ result $<= \pi$.

7. Arctangent: The range of the result for ATAN and DATAN is: $-\pi/2 <=$ result $<= \pi/2$. If the value of the first argument of ATAN2 or DATAN2 is zero or positive, the result is zero or positive, respectively. If the value of the first argument is negative, the result is negative. The value of both arguments must not be zero. The range of the result of ATAN2 and DATAN2 is: $-\pi <$ result $<= \pi$.

The above restrictions on arguments and results also apply to the intrinsic functions when referenced by their generic names.

Logical Word Operations

FORTRAN 5 permits operations on integers at the bit level using library functions.

Each function in Table 8-3 treats its arguments as 16-bit integers without regard to sign. An argument can be either a variable, an array element, or an expression and must be of type integer. After FORTRAN 5 evaluates a function, it returns an integer value to the function reference that invoked the function.

FORTRAN 5 performs the boolean operations (ANDing, ORing, and exclusive ORing) on a bit-by-bit basis. That is, the *nth* bit of arg1 and the *nth* bit of arg2 together determine the *nth* bit of the value returned. All 16 bits of each argument participate in the function. Table 8-3 describes the specific actions the functions take, given all possible bit combinations.

Table 8-3. Logical Word Operation

			Result in a given bit position if:			
Name	**Format**	**Definition**	b1=0 b2=0	b1=0 b2=1	b1=1 b2=0	b1=1 b2=1
IAND	IAND(arg1,arg2)	16-bit ANDing	0	0	0	1
IOR	IOR(arg1,arg2)	16-bit ORing	0	1	1	1
IEOR	IEOR(arg1,arg2)	16-bit exclusive ORing	0	1	1	0
IXOR	IXOR(arg1,arg2)	Alternate name for IEOR	0	1	1	0
NOT*	NOT(arg)	Change all bits containing 0 to 1 and all bits containing 1 to 0.				
ISHFT	ISHFT(arg,n)	To shift a 16-bit word: n=0 no shift n>0 shift left n bits n<0 shift right n bits The valid range is -16 to +16: incorrect results may be expected outside this range; FORTRAN 5 shifts bits out of the word; it does not rotate them.				

*Note the spelling of this function; it is the only logical function that does not begin with an I.

Single Bit Operations

Table 8-4 shows one library function, ITEST, and two runtime routines, ICLR and ISET, that perform single bit operations on a given integer.

Each of the statements below treats its arg1 as a 16-bit integer without regard to sign. This argument can be either a variable, an array element, or an expression and must be of type integer. FORTRAN 5 does not perform any conversions.

In each of the statements below, arg2 represents the bit position to be operated on and must be either a variable, a constant, an array element, or an expression of type integer. Returned values are also of type integer.

Note the bit numbering scheme: 15 is the most significant bit and 0 is the least significant bit (left to right). If the given bit number is not in this range, ITEST returns 0, and ICLR and ISET do nothing (FORTRAN 5 does not report an error).

Table 8-4. Single Bit Operations

				Result if Bit is:	
Name	**Statement Type**	**Format**	**Definition**	**1**	**0**
ITEST	Function reference	ITEST(arg1,arg2)	To test a bit	1	0
ICLR	Call to run-time routine	CALL ICLR(arg1,arg2)	To set a bit to zero	0	0
ISET	Call to run-time routine	CALL ISET(arg1,arg2)	To set a bit to one	1	1

FLD Function

Accesses and/or modifies the contents of any bit field.

Format

FLD (var,ibit$_1$ *[,ibit$_2$]*)

Where:

var	is the name of a variable, array, or array element.
ibit$_1$	is the number of the first bit in the field.
ibit$_2$	is the number of the last bit in the field.

Function Execution

Both ibit$_1$ and *ibit$_2$* must be either an integer or an expression that FORTRAN 5 converts to integer, if necessary. If you do not specify *ibit$_2$*, it assumes the same value as ibit$_1$. FORTRAN 5 numbers the bits 1 through *n*, from left to right (most significant to least significant), where *n* is the number of bits in the variable, array, or array element.

Be sure to specify a valid bit range in a FLD function because the FORTRAN 5 compiler does not check it.

The FLD function allows arithmetic manipulation of the value contained in any bit field having 1-16 bits. When you modify bit contents, FORTRAN 5 right justifies the stored value in the bit field and truncates the leftmost bits, if necessary. For example, FLD (I,3)=7 sets the third bit of I to 1. Because this statement deals with a single bit, FORTRAN 5 assigns only the rightmost bit of 7 (binary representation 111), namely 1. In the example FLD (I,2,3)=7, FORTRAN 5 sets the 2nd and 3rd bits of I to the rightmost two bits of 7, or the value 3.

The following constraints apply:

ibit$_1$ $\geq$1

ibit$_1$ $\leq$ibit$_2$

ibit$_2$ - ibit$_1$ $<$ 16

Unlike most functions, the FLD function can appear on the left-hand side of an equal sign (see Example 1 below).

Examples

1. Replace the contents of bits 2 through 5 of I with the value 7.

   ```
   FLD (I,2,5) = 7
   ```

2. An argument passed to SWITCH is an integer representing the value contained in bits 100 through 110 of array Z.

   ```
   CALL SWITCH (FLD(Z,100,110))
   ```

3. Add 1 to the value contained in X, bits 1 through 6.

   ```
   FLD (X,1,6) = FLD (X,1,6) + 1
   ```

   If the resulting value is too large, the leftmost bit is lost.

BYTE Function

Accesses and/or modifies the contents of any byte.

Format

BYTE (var,ibyte)

Where:

var	is the name of a variable, array, or array element.
ibyte	is the number of a byte.

Function Execution

Ibyte must be either an integer or an expression that FORTRAN 5 converts to an integer, if necessary. FORTRAN 5 numbers bytes 1 through *n*, from left to right (most significant to least significant), where *n* is the number of bytes in the variable, array, or array element.

Be sure to specify a valid bit range in a BYTE function because the FORTRAN 5 compiler does not check this.

When you modify the contents of a byte, FORTRAN 5 converts the value to type integer, right justifies the integer value, and truncates the leftmost bits, if necessary, before storing it.

Unlike most functions, the BYTE function can appear on the left-hand side of an equal sign (see Example 1 below).

Examples

1. Replace the contents of byte 12 in variable K with the value 134.

   ```
   BYTE (K,12) = 134
   ```

2. Two arguments passed to DINE1 are integers representing the values contained in bytes 21 and 22 of array X.

   ```
   CALL DINE1 (BYTE(X,21),BYTE(X,22))
   ```

3. The following expressions are equivalent:

   ```
   FLD(V,8*I-7,8*I)

   BYTE(V,I)
   ```

Block Data Subprogram

A *block data subprogram* assigns initial values to variables and arrays in both named and blank COMMON blocks. The subprogram begins with a BLOCK DATA statement of the form:

```
BLOCK DATA
```

This should be the first statement of the subprogram, but it may appear anywhere within the subprogram. The compiler ignores any labels on a BLOCK DATA statement. This statement is followed by nonexecutable statements (specification statements). An END statement terminates the subprogram.

A block data subprogram functions at compile time; that is when FORTRAN 5 assigns data to the specified block(s).

If you want to access a variable or array in a common block that is not at the beginning of the block, you must specify all entities up to and including that entity in the COMMON statement (i.e., the block is position-dependent).

Example

```
BLOCK DATA
COMMON /ELN/C,A,B/RMC/Z,Y
COMMON P,Q
DIMENSION B(4),Z(3)
DOUBLE PRECISION Z
COMPLEX C
DATA (B(I),I=1,3)/1.1,2*1.2/,C/(2.4,3.769)/,Z(1)/7.64D-8/
DATA P/4./
END
```

End of Chapter

Chapter 9
Multitask Programming

Introduction

Multitasking is a technique that allows a single program to deal easily and efficiently with two or more tasks at one time. You can apply it most efficiently when the various tasks you want to perform are independent of one another with respect to timing. Since the computer can execute only one instruction at a time, the various tasks take turns executing. CPU control switches rapidly from task to task. The system handles this switching automatically; switching is transparent to the individual tasks involved.

The following example illustrates how multitasking makes programming solutions to certain problems easier and, at the same time, more efficient. A process control environment typically includes input devices which require monitoring and output devices which the system must control according to the data read from the input devices. The real-time program you design to handle this situation needs to perform several logically distinct tasks. With multitasking, you may program these tasks independently. When you run the program, it divides its execution into a number of separate tasks, each task performing its specified function asynchronously and in real-time. In this way, multitasking relieves you of the burden of keeping track of the various tasks and of appropriately switching control among the tasks. Moreover, multitasking handles the environment more efficiently: While one or more tasks await the completion of their I/O operations, other tasks may use the time available for computation.

Multitasking allows you to exercise a fine control over the tasks which the system selects for execution and when they are selected for execution. When you define a task and specify the instructions it will execute, you also assign the task a priority relative to other tasks. You may change task priorities during program execution, allowing you to control which tasks receive CPU control and when. A task scheduler allocates CPU control to the highest priority task that is ready either to perform or to continue to perform its function.

Although each task in a multitask environment may execute independently, a certain amount of interaction between the tasks is often required. DGC's multitasking allows you to communicate between tasks conveniently, providing for task synchronization. For example, a task may suspend its own execution at a certain point, awaiting another task's signal to continue.

In certain situations it is appropriate for two or more tasks to execute the exact same sequence(s) of instructions, yet still be independent of one another. It is more efficient that these several tasks share a single copy of the instruction sequences than it is to duplicate the code several times. There is no reason why such a sharing cannot take place, provided that the code does not modify itself, and that FORTRAN 5 stores the data it accesses in a separate location for each task. We call such code *re-entrant.* All compiled FORTRAN 5 programs and all FORTRAN 5 routines are re-entrant if you allocate all variables on the stack rather than in static or common storage.

Tasks

A *task* is a logically complete unit of program execution that requires the use of system resources, such as memory and CPU control. A program, the current executable contents of a user address space, contains the code paths (sequences of instructions) that tasks execute. Singletask operation is characterized by a single flow of control through a program, no matter how complex the branching structure of that program may be. Multitask operation consists of multiple, concurrent flows through a program, where the various flows (tasks) compete for CPU control. In a multitask environment, the tasks may execute different code paths or they may execute the same code paths, provided that the paths are re-entrant.

Task Priorities

Multitask operation uses CPU time more efficiently by allowing multiple, asynchronous tasks to receive CPU control on a priority and readiness basis. You assign tasks relative priorities ranging from 0 (the highest) to 255 (the lowest). A *task scheduler* allocates CPU control to the highest priority ready task. You give a task an initial priority when you initiate it. However, you may change a task's priority during program execution.

Several tasks may exist at the same priority. Equal priority tasks receive CPU control on a *round-robin* basis. This means that, of a group of tasks at the same priority, the task which most recently received control will be the last to receive control again, unless the other tasks in the group are not ready to receive control when their turns come. Whenever a task has a priority change, the system places the task at the end of the group of all tasks at its new priority.

Task Identification Numbers

To enable your program to further differentiate among tasks, the system permits you to assign a unique identification number (ID) to each task. This facility gives an added dimension of clarity to task management, because many tasks may exist at the same priority, but each task's ID remains both unique and constant. A task ID must be in the range of 1 to 255; you assign it when you initiate the task. No two tasks may have the same ID. However, any number of tasks may have no ID.

The Multitask Environment

When you execute a FORTRAN 5 program, an initial task executes the FORTRAN 5 main program unit. To create a multitask environment, this task must initiate one or more additional tasks. Once it accomplishes this, either the original task or a newly initiated task may initiate other tasks. Task initiation is usually performed by the TASK statement.

Once initiated, a task may be in one of two basic states: suspended or ready. A task may become suspended due to a number of actions: it may be awaiting the completion of a system call or a message from another task; another task may suspend it; or the task may suspend itself. The system cannot place a suspended task back into execution until it has resolved the reason(s) for its suspension.

Commonly, a task may be suspended for more than one reason. For example, when a task issues a call to the operating system, the task is suspended until the system call processing is finished. While the task is waiting for the system call to complete, another task may suspend it. When the system call completes, the system call suspension is lifted, but the task remains suspended and needs to be readied by some other task before it can run again.

A task is ready if it is not suspended. Although many tasks may be ready at the same time, only one can be executing. The task scheduler always chooses for execution that ready task which currently has the highest priority, i.e., the task with the lowest priority number.

Task Control Blocks

Tasks operate asynchronously; that is, there need be no specific order of execution among the tasks. They share certain physical resources of the system (accumulators, carry, floating point unit, and special memory locations, such as the stack control words). So that a task may correctly resume its operation when the task scheduler selects it for execution, the system must maintain certain state information about and for each task that is not currently executing. The system stores this information in a structure called a *task control block* (TCB). Each initiated task requires one TCB; therefore, the number of tasks that may exist at one time is limited to the number of TCBs available. The binder/loader allocates a pool of TCBs at relocatable load time. (See the description of this command in Part I, Chapter 1 of the *FORTRAN 5 Programmer's Guide.*)

The task scheduler maintains two chains of TCBs, the "active chain" and the "inactive chain". The *active chain* is a collection of all active TCBs linked in priority order. The *inactive chain* consists of free TCBs available for use in initiating other tasks. When you initiate a task, the system removes an empty TCB from the inactive chain, initializes and links it into the active chain in a position appropriate to the new task's priority. When you terminate a task, the system removes its TCB from the active chain and returns it to the inactive chain.

FORTRAN 5 Multitasking Statements

The FORTRAN 5 language includes six statements for creating and managing a multitask environment. The TASK statement initiates a task and specifies the initial subroutine that the task will execute. A TASK statement also specifies the task's ID, its initial priority, and any actual arguments to pass to that subroutine which the task begins executing. The KILL statement terminates a task. The WAIT, SUSPEND, WAKEUP, and ANTICIPATE statements control whether a given task will be in a ready or suspended state. These four statements use the concept of an *event,* which we describe below.

In addition to the six statements mentioned above, there are numerous runtime library routines which assist in the management of a multitask environment in FORTRAN 5. These routines are described in the *FORTRAN 5 Programmer's Guide.* For additional details on multitask operation, consult the reference manual appropriate to your operating system. Many of the FORTRAN 5 runtime routines for multitask management are direct interfaces to operating system task calls.

Events

An *event* is a programmer-defined occurrence which synchronizes tasks in a multitask environment. A task may be suspended pending the occurrence of a particular event; the task will be readied when that event occurs. A FORTRAN 5 program identifies an event by a positive integer; you may represent at most 32,767 different events. The value 0 represents the lack of an event association. Negative values are not valid event numbers.

There are four statements that reference events: WAIT, SUSPEND, WAKEUP, and ANTICIPATE. A task may suspend itself (or another task) pending the occurrence of a specified event with a WAIT (or SUSPEND) statement. The suspended task can be readied by a WAKEUP statement appearing in another ready task. The ANTICIPATE statement allows a task to respond to a WAKEUP statement even if the system executes the WAKEUP statement before it executes the corresponding WAIT statement.

Suspension pending the occurrence of an event is independent of all other types of suspension. That is, a task suspended by a WAIT or SUSPEND statement can be readied only by an appropriate WAKEUP statement. Moreover, a WAKEUP statement is not sufficient to ready a task suspended by both WAIT and SUSPEND statements (or for some other reason(s)); all reasons for suspension must be lifted before the task will be ready.

In the following descriptions of multitasking statements, when we refer to an entity as an *integer,* that entity may be an integer constant, an integer variable, an integer array element, or an integer expression.

TASK

Initiates a task.

Format

TASK subr *[(arg$_1$,...,arg$_n$)][,ID=i][,PRI=j][,STK=k]*
[,ERR=label]

Where:

subr	is the initial subroutine the task will execute.
arg	is the name of an actual argument you want to pass to that subroutine.
i	is an integer between 1 and 255 inclusive, specifying an identification number for the new task, or 0 to specify no identification number.
j	is an integer between 0 and 255 inclusive, specifying the initial priority for the new task.
k	is an integer that specifies a FORTRAN 5 stack size for the new task.
label	is a statement label in the calling program to which control passes if an error occurs during initiation of the task.

Statement Execution

The optional arguments above are independent and you may give them in any order. The argument list, if present, must precede any *ID=*, *PRI=*, *STK=*, or *ERR=* specifiers.

When you initiate a task, you specify a subroutine for the task to execute. This subroutine may call other subprograms. When the task has completed its work, it may kill itself or another task may kill it. Alternately, if the system allows the task to return from the subroutine it initially began executing, it is automatically killed. Such a return may be made by executing a RETURN statement in the subroutine or by allowing control to pass to the subroutine's END statement. You may not specify alternate returns from a subroutine executed by a task.

You may pass arguments to the initial subroutine executed by a task. The actual arguments passed replace the dummy arguments in the subroutine and must agree in order, number, and type. The conventions governing actual and dummy argument correspondence are the same as those discussed in Chapter 8.

TASK (continued)

When passing arguments in the TASK statement, you must take certain precautions if the creator task can terminate before the created task does. The issue is that when a task terminates, certain classes of data associated with that task become inaccessible. A task initiated by a TASK statement may not access certain classes of arguments passed to its subroutine after the creator task terminates. There are two cases to consider. If the creator task terminates but the code it was executing remains where it is in memory, the arguments it passes in a TASK statement must be either variables, arrays, or array elements that are common, static, data-initialized, or constants. You must not try to pass other variables, arrays, array elements, or expressions of any sort. On the other hand, if the creator task terminates and its code does not remain (for example, if the code resides in an overlay which is overwritten after the system kills the task), you may pass as arguments only variables, arrays, or array elements that are in COMMON; you must not try to pass other variables, arrays, array elements, expressions of any sort, or constants.

You may assign an identification number (ID) to a task when you initiate it with the TASK statement. The task ID must be an integer expression in the range 1 to 255 inclusive. If you specify 0 or do not specify any task ID, no ID is assigned. If you specify a nonzero task ID, it must be unique among all task ID's in use at the time or an error occurs.

When you initiate a task, you must assign it an initial priority. You may assign this priority explicitly in the TASK statement with the *PRI=j* option, where *j* is an integer expression in the range 0 to 255 inclusive. If you specify a nonzero priority, it is assigned as the new task's initial priority. If you specify a priority of 0 (or if you omit the *PRI=* option altogether), the system assigns the initiated task an initial priority equal to the current priority of the creator task.

When you create a task in a FORTRAN 5 environment, the system assigns that task an area of memory to use for its runtime stack. It retains this stack area until it is killed; at this time the area is returned to the pool of available stack areas. Since different tasks may require different amounts of memory for their execution, FORTRAN 5 provides a facility for allocating memory unequally to the various tasks in your program. As you prepare your program for execution, you may specify how memory should be partitioned (see Part I, Chapter 1 of the *FORTRAN 5 Programmer's Guide)*. When you execute your program, the system sets up memory partitions of the appropriate sizes. When you initiate a task, you may use the *STK=k* option to specify which partition the task will use for its stack area, where *k* is an integer whose value is interpreted as follows:

Stack Size Specification	**Result**
0 or 1	Select an available default-size partition.
2	Select the smallest available partition.
3	Select the largest available partition.
k > 3	Select a partition of size exactly k words.
100000K	Select no partition.

If you omit the *STK=* option entirely, FORTRAN 5 uses a stack size specification of 0. If a partition of the appropriate size is not available, FORTRAN 5 does not create the task and signals an error.

The *ERR=* option allows you to specify a statement to which control will pass if an error occurs during task initiation. The system cannot create a task if there is no empty TCB available, or if a memory partition of the correct size is not available.

Examples

```
TASK T1(3,X,Y),ID=5,ERR=50

TASK T2,ID=6,PRI=4,STK=3

TASK T3(I+1)
```

KILL

Terminates a task.

Format

```
KILL id
```

Where:

id	is an integer that specifies the identification number of a task.

Statement Execution

A KILL statement terminates the identified task. The system releases the task's resources, including TCB and runtime stack area, and makes them available to initiate other tasks. However, killing a task does not close any of the I/O units the task is using, since all tasks share the unit numbers.

A KILL statement must contain a legal identification number in the range 1 to 255. If the number you give is outside this range, or if no task has that ID, an error occurs.

Examples

```
KILL 10

KILL J + 2
```

WAIT

Allows a task to suspend itself.

Format

```
WAIT event
```

Where:

event	is a positive integer that specifies the event number.

Statement Execution

Execution of the WAIT statement suspends a running task. Control passes to the ready task with the highest priority.

The event number you specify in the WAIT statement identifies the event that must occur before the system readies the suspended task. Valid event numbers are strictly positive integers. An error occurs if you specify an event number that is negative. If you specify an event number of 0, no action is taken (no waiting occurs).

A task suspended by a WAIT statement can be readied only by a WAKEUP statement executed in another task that specifies the event number for which the suspended task is waiting. However, if you SUSPEND the task in which the WAIT occurred, the task is readied only when you specify a WAKEUP on the event associated with the SUSPEND statement. For example, if Task 1 executes the statement:

```
WAIT 3
```

then Task 2 executes the statements:

```
SUSPEND 1,4
WAKEUP 3
WAKEUP 4
```

The WAKEUP on event 3 in Task 2 will have no effect on Task 1, but the WAKEUP on event 4 will ready Task 1.

If the task that executes a WAIT statement previously executed an ANTICIPATE statement for the same event, and if the WAKEUP statement for that event has already been executed, then the task is not suspended and remains ready.

Example

```
WAIT 33
```

SUSPEND

Allows a task to suspend itself or another task.

Format

SUSPEND id,event

Where:

id	is an integer that specifies the identification number of a task.
event	is a positive integer that specifies the event number.

Statement Execution

The SUSPEND statement allows a running task to suspend itself or another task pending the occurrence of the event you specify in the SUSPEND statement. Valid identification numbers are in the range 1 to 255 inclusive. If the number you give is outside this range, or if no task has that ID, an error occurs.

Valid event numbers are strictly positive integers. An error occurs if you specify an event number that is negative. If you specify an event number of 0, no action is taken (no suspension occurs).

A task suspended by a SUSPEND statement can be readied only by a WAKEUP statement executed in another task that specifies the event number for which the suspended task is waiting.

If the task specified by id in the SUSPEND statement previously executed an ANTICIPATE statement for that same event, and if the WAKEUP statement for that event has already been executed, then the task is not suspended and remains ready.

Example

```
SUSPEND 2,5
```

WAKEUP

Readies a task suspended by a WAIT or SUSPEND statement.

Format

WAKEUP event

Where:

event	is a positive integer that specifies the event number.

Statement Execution

Execution of a WAKEUP statement removes the event suspension for all tasks waiting for the given event. Unless the system suspends such a task for another reason as well, it readies the task. Control then passes to the highest priority ready task, which may be one of the newly readied tasks.

The WAKEUP statement also registers a *wakeup* with all tasks that have ANTICIPATEd the given event, but have not yet been suspended. (See the ANTICIPATE statement for more information.)

Valid event numbers are strictly positive integers. An error occurs if you specify an event number that is negative. If you specify an event number of 0, no action is taken (no wakeup occurs). If you specify an event that is not currently associated with any task, no error occurs and the WAKEUP statement has no effect.

Example

```
WAKEUP 33
```

ANTICIPATE

Associates an event with a task in order to register a WAKEUP on the event that occurs before the WAIT or SUSPEND for that task.

Format

ANTICIPATE event

Where:

event is an integer that specifies the event number.

Statement Execution

When you use the ANTICIPATE statement you safeguard against a task being suspended indefinitely when the WAKEUP statement for the associated event occurs before a WAIT or if a SUSPEND statement suspended that task. The execution of an ANTICIPATE statement in a task associates the specified event number with that task. The ANTICIPATE statement affects only the task in which it occurs.

If the WAIT or SUSPEND statement for that particular event occurs before the WAKEUP statement for that event, the system suspends the task in the normal manner and places it in the ready state when it encounters the WAKEUP.

If the WAKEUP for the event occurs after the ANTICIPATE but before the WAIT or SUSPEND, the WAKEUP statement indicates a WAKEUP condition for the referenced task. Then when the WAIT or SUSPEND statement for that event does occur, the WAKEUP event is acknowledged as having already occurred, and execution of the task proceeds without a suspension occurring. If the ANTICIPATE statement was not present, the WAKEUP would be ignored, and the WAIT or SUSPEND could cause an indefinite suspension.

The event number in an ANTICIPATE statement can be a valid event number (a positive integer), or it can be zero. An event number of zero simply means "no longer anticipate the event I was anticipating." An error occurs if you specify an event number that is negative.

When a task that ANTICIPATEd an event executes another ANTICIPATE statement which specifies a different event number, any wakeup registered for that task on the original event number is lost. For example:

```
Task 1               Task 2

.                    .
.                    .
ANTICIPATE 3         .
.                    .
.                    WAKEUP 3
ANTICIPATE 4         .
.                    .
.
WAIT 4
.
.
```

Execution of the above statements in the order indicated will suspend Task 1 until a WAKEUP 4 occurs.

However, if a task issues an ANTICIPATE statement that specifies the event it is already anticipating, the task does not lose the registered wakeup on the event. For example:

```
Task 1 Execution      Time          Task 2 Execution

.                  increasing       .
.                      |            .
ANTICIPATE 3           |            .
.                      |            .
.                      |            WAKEUP 3
ANTICIPATE 3           |            .
.                      |            .
.                      |            .
WAIT 3                 |            .
.                      |            .
.                      v            .
```

Execution of the WAIT statement will not suspend Task 1. (Note that this example is time-dependent.)

FORTRAN 5 Multitasking Example

Figure 9-1 shows the use of some of the FORTRAN 5 multitasking statements. This example is not intended to be realistic, but only to show how these statements operate together.

The program consists of a main program and three subroutines, T1, T2, and T3. A task with no ID executes the main program. This task initiates a second task (with an ID of 1) to execute subroutine T1. The main program task then asks you to type in the number of a subroutine to execute. If you type in 1, the main program task wakes up task 1, which types *T1 REPORTING* on the console. If you type in 2 or 3, the main program task initiates a third task on the spot to execute either subroutine T2 or T3. This new task (with an ID of 2 or 3) identifies itself, then is killed when control passes to the end statement in its subroutine.

The main program task loops repeatedly, accepting the number of a subroutine to run and directing a task to execute that subroutine. You terminate execution of this example program by entering a subroutine number of 0.

Execution of subroutines T2 and T3 is straightforward; a new task is initiated to execute either of these subroutines. After initiating the new task, the main program task continues on about its business, while the new task runs and is killed automatically when the subroutine it executes ends. To execute subroutine T1, however, the main program task need only to wake up the existing task 1 in order to continue execution of the subroutine. FORTRAN 5 saves the time required to initiate and terminate a task each time subroutine T1 is executed, but a task control block (TCB) must be available continuously in order to execute task 1; the communication between the main program task and task 1 is more complicated. Task 1 suspends its own execution by waiting for event 8. The main program task executes a WAKEUP on event 8 to allow task 1 to continue its execution. The program uses Event 7 to make sure that the main program task does not try to tell task 1 to execute the body of subroutine T1 again until task 1 has completed its prior execution of the subroutine body. When task 1 finishes an execution of the subroutine body, it signals this by executing a WAKEUP on event 7. This allows the main program task to proceed through its wait 7 and call for another execution of subroutine T1's body. Note that the main program task is constantly ANTICIPATEing event 7, so it will not miss task 1's signal indicating it is ready to make another pass over subroutine T1. In a corresponding manner, subroutine T1 is constantly ANTICIPATEing event 8, so it will not miss the main program's signal to begin its next pass.

```
C         MAIN PROGRAM

          TYPE "MAIN PROGRAM STARTING"

          ANTICIPATE 7
          TASK T1, ID=1

10        TYPE "ENTER NUMBER OF SUBROUTINE TO RUN"
          ACCEPT I
          IF (I .EQ. 0) STOP
          GO TO (1,2,3), I
          TYPE "INVALID SUBROUTINE NUMBER"
          GO TO 10

1         WAIT 7
          ANTICIPATE 7
          WAKEUP 8
          GO TO 10

2         TASK T2, ID=2
          GO TO 10

3         TASK T3, ID=3
          GO TO 10

          END



          SUBROUTINE T1

          TYPE "T1 STARTING"

5         ANTICIPATE 8
          WAKEUP 7
          WAIT 8
          TYPE "T1 REPORTING"
          GO TO 5

          END


          SUBROUTINE T2

          TYPE "T2 REPORTING"

          END


          SUBROUTINE T3

          TYPE "T3 REPORTING"

          END
```

Figure 9-1. Multitasking Example

End of Chapter

Appendix A
Alphabetized List of FORTRAN 5 Statements

Statement	Function	Chapter
ACCEPT	Allows input/output of data from input console upon prompt.	5
ANTICIPATE	Associates an event with a task in order to register a WAKEUP on the event that occurs before the WAIT or SUSPEND task.	9
ASSIGN	Associates a statement label with an integer variable.	3
Assignment, arithmetic	Assigns the value of an expression to a specified entity.	3
Assignment, logical	Assigns the value of an expression to a specified logical entity.	3
BACKSPACE	Backspaces a file's record pointer and positions it to the beginning of the previous record.	5
BLOCK DATA	Assigns values to variables and arrays in both named and blank COMMON blocks.	8
CALL	Invokes a subroutine, transferring control from one program unit to another.	8
CLOSE	Closes an opened file.	5
COMMON	Allocates an area of data storage accessible to multiple program units, and names the variables and arrays which will reside in this area.	7
COMPILER	Permits you to specify the compile-time options STATIC, FREE, and DOUBLE PRECISION.	7
COMPLEX	Specifies a symbolic name to have the data type COMPLEX.	7
CONTINUE	Provides a place for a label.	4
DATA	Defines initial values for variables and array elements.	7
DECODE	Performs data transfers according to a format specification.	5
DELETE	Deletes a file from disk.	5
DIMENSION	Names arrays and specifies their dimensions.	7
DO	Executes a group of statements one or more times.	4
DOUBLE PRECISION	Specifies a symbolic name to have the data type DOUBLE PRECISION.	7
DOUBLE PRECISION COMPLEX	Specifies a symbolic name to have the data type DOUBLE PRECISION COMPLEX.	7
ENCODE	Performs data transfers strictly between variables or arrays internal to your program according to a format specification.	5

Statement	Function	Chapter
END	Marks the end of a program unit.	4
ENDFILE	Closes an opened file.	5
EQUIVALENCE	Associates two or more entities in the same storage area.	7
EXTERNAL	Allows you to use an externally defined subprogram name or overlay name as an argument.	7
FORMAT	Designates the structure of the records and the form of the data fields within the records of a file.	6
FUNCTION	Begins and defines a function subprogram.	8
GO TO, assigned	Transfers control to a previously ASSIGNed statement label.	4
GO TO, computed	Transfers control to one of several specified statements depending on the value of a specified variable.	4
GO TO, unconditional	Transfers control unconditionally to a specified statement.	4
IF, arithmetic	Transfers control conditionally to one of three statements based on the value of an arithmetic expression.	4
IF, logical	Conditionally executes and transfers control to a statement based on the value of a logical expression.	4
IMPLICIT	Changes or confirms the default data type of symbolic names.	7
INCLUDE	Allows you to insert a FORTRAN 5 source file in the current FORTRAN 5 program.	1
INTEGER	Specifies a symbolic name to have the data type INTEGER.	7
KILL	Terminates a task.	9
LOGICAL	Specifies a symbolic name to have the data type LOGICAL.	7
OPEN	Assigns a unit number to a file and creates the file, if necessary, according to specifications given.	5
OVERLAY	Specifies a subprogram as an overlay and names it.	7
PARAMETER	Assigns a symbolic name to a constant or to an expression.	7
PAUSE	Temporarily suspends program execution, waiting for operator intervention.	4
PRINT	Transfers data between internal storage and the line printer.	5
PUNCH	Transfers data between internal storage and the paper tape punch.	5
READ	Transfers data from a file to internal storage according to specifications in the corresponding FORMAT statement.	5
READ, simple	Transfers data between internal storage and the card reader.	5
READ BINARY	Transfers a single data record from a file to internal storage with no interpretation.	5

Statement	Function	Chapter
READ FREE	Transfers and converts externally recognizable data to their internal computer representation, providing a standard formatting without programmer intervention.	5
READ INPUT TAPE	Alternate form of READ.	5
READ TAPE	Alternate form of READ BINARY.	5
REAL	Specifies a symbolic name to have the data type REAL.	7
RENAME	Changes the name assigned to an existing file.	5
RETURN	Marks the logical end of a function or subprogram and returns control from that subprogram to the calling program unit.	8
REWIND	Repositions the record pointer to the beginning of a specified file.	5
STATIC	Places specified variables and arrays in a fixed area in memory rather than on the runtime stack.	7
STOP	Causes unconditional termination of program execution.	4
SUBROUTINE	Begins and defines a subroutine.	8
SUSPEND	Allows a task to suspend itself or another task.	9
TASK	Initiates a task.	9
TYPE	Allows interaction between you and your program using the console for output.	5
WAIT	Allows a task to suspend itself.	9
WAKEUP	Readies a task suspended by a WAIT or SUSPEND statement.	9
WRITE	Writes data from internal storage to a file or device specified by a unit number.	5
WRITE BINARY	Transfers a single data record from internal storage to a file with no interpretation.	5
WRITE FREE	Transfers and converts data according to a standard field format.	5
WRITE OUTPUT TAPE	Alternate form of WRITE.	5
WRITE TAPE	Alternate form of WRITE BINARY.	5

End of Appendix

Appendix B
Compiler Error and Warning Messages

```
0:  UNDEFINED ERROR CODE, CALL DGC
1:  ILLEGAL FIELD DESCRIPTOR IN FORMAT
2:  ILLEGAL FIELD WIDTH IN @ FIELD BEFORE A "/"
3:  ILLEGAL FIELD DESCRIPTOR IN FORMAT FOLLOWING A SEPARATOR
4:  ILLEGAL REPEAT OR HOLLERITH COUNT IN FORMAT
5:  ERROR AFTER "P", SCALE SPECIFIER, IN FORMAT
6:  ILLEGAL SCALE SPECIFIER IN FORMAT
7:  ZERO OR NEGATIVE REPEAT COUNT FOR @ FIELD
8:  FIELD WIDTH MUST FOLLOW @ FIELD
9:  FIELD WIDTH FOR @ DESCRIPTOR MUST CONTAIN "."
10:  ILLEGAL CHARACTER AFTER "." IN @ FORMAT
11:  @ FIELD DOES NOT END PROPERLY IN FORMAT
12:  ILLEGAL FIELD WIDTH FOR @ FORMAT PRECEDING ","
13:  ZERO OR NEGATIVE REPEAT COUNT FOR @ FORMAT
14:  FIELD WIDTH MUST FOLLOW @ FORMAT
15:  FIELD WIDTH FOR @ DESCRIPTOR DOES NOT END PROPERLY
16:  ZERO OR NEGATIVE REPEAT COUNT BEFORE A "("
17:  ILLEGAL FIELD WIDTH IN FORMAT PRECEDING ")"
18:  PARENTHESES DO NOT BALANCE IN FORMAT
19:  ZERO OR NEGATIVE REPEAT COUNT BEFORE AN "X" IN FORMAT
20:  ZERO OR NEGATIVE HOLLERITH COUNT IN FORMAT
21:  PARENTHESES DO NOT BALANCE IN FORMAT STATEMENT
22:  ZERO OR NEGATIVE REPEAT COUNT IN @ FORMAT BEFORE A QUOTE
23:  INTEGER MUST FOLLOW A "-"
24:  "O" FORMAT CAN NOT PRECEDE @ NUMERIC FIELD DESCRIPTOR
25:  UNCLOSED STRING CONSTANT IN STATEMENT
26:  UNCLOSED STRING CONSTANT IN FORMAT
27:  ILLEGAL STATEMENT NUMBER
28:  OCTAL CONSTANT MUST BE INTEGER
29:  ILLEGAL OCTAL DIGIT
30:  NEGATIVE COUNT BEFORE @ HOLLERITH CONSTANT
31:  ILLEGAL CONTINUATION LINE
99:  TOO MANY ITEMS IN LIST
100:  FORMAT STATEMENT REQUIRES STATEMENT NUMBER
101:  MISSING OR INVALID VARIABLE
102:  MISSING OR INVALID IDENTIFIER
103:  MISSING OR INVALID OVERLAY NAME
104:  NOT A LEGAL FORTRAN STATEMENT
105:  STATEMENT DOES NOT END PROPERLY
106:  MISSING "=" AFTER IDENTIFIER @
107:  MISSING OR INVALID EXPRESSION AFTER "="
108:  MISSING OR INVALID INITIALIZATION ITEM
109:  MISSING "/" AFTER INITIALIZATION LIST
110:  MISSING OR INVALID DIMENSION FOR ARRAY @
111:  MISSING ")" AFTER DIMENSION LIST
112:  MISSING ")" IN EXPRESSION
113:  MISSING OR INVALID COMMON BLOCK NAME
114:  ERROR IN BLANK COMMON DECLARATION
115:  MISSING "/" AFTER COMMON BLOCK NAME @
116:  EXTRANEOUS "," AFTER "/"
117:  ERROR IN DECLARATION OF COMMON BLOCK @
118:  MISSING OR INVALID SUBROUTINE NAME
119:  INVALID ITEM IN ARGUMENT LIST
120:  MISSING ")" AFTER PARAMETER LIST
121:  MISSING "/" AFTER VARIABLE LIST
```

```
122: MISSING OR INVALID EXPRESSION AFTER "=" IN IMPLIED DO
123: MISSING "," AFTER @ IN IMPLIED DO
124: MISSING EXPRESSION AFTER "," IN IMPLIED DO LIST
125: CLOSING ")" OF IMPLIED DO NOT FOUND
126: MISSING "(" BEGINNING EQUIVALENCE GROUP
127: CLOSING ")" OF EQUIVALENCE GROUP NOT FOUND
128: MISSING OR INVALID EQUIVALENCE GROUP AFTER ","
129: MISSING DATA-TYPE
130: MISSING "(" AFTER DATA-TYPE
131: MISSING OR INVALID CHARACTER IN LIST
132: MISSING ")" AFTER CHARACTER LIST
133: MISSING OR INVALID OPTION
134: MISSING VARIABLE BEFORE "=" IN ASSIGNMENT STATEMENT
135: MISSING "=" AFTER VARIABLE @
136: MISSING "," AFTER INITIAL VALUE SPECIFICATION
137: MISSING "(" AFTER "IF"
138: MISSING OR INVALID EXPRESSION AFTER "IF"
139: MISSING ")" AFTER EXPRESSION
140: OBJECT OF LOGICAL IF NOT A LEGAL FORTRAN STATEMENT
141: MISSING "," AFTER STATEMENT LABEL
142: SECOND STATEMENT LABEL MISSING OR INVALID
143: THIRD STATEMENT LABEL MISSING OR INVALID
144: EXTRANEOUS "," IN POSSIBLE ASSIGNMENT STATEMENT
145: MISSING OR INVALID LABEL AFTER "DO"
146: MISSING OR INVALID DO VARIABLE
147: MISSING "=" AFTER VARIABLE @ IN INITIAL VALUE SPECIFICATION
148: INITIAL EXPRESSION AFTER "=" MISSING OR INVALID
149: LIMIT EXPRESSION MISSING OR INVALID
150: INCREMENT EXPRESSION MISSING OR INVALID
151: MISSING "(" BEFORE STATEMENT LABEL LIST
152: INVALID GO TO STATEMENT
153: MISSING ")" AFTER STATEMENT LABEL LIST
154: MISSING OR INVALID EXPRESSION AFTER STATEMENT LABEL LIST
155: MISSING OR INVALID STATEMENT LABEL
156: MISSING OR INVALID SUBROUTINE NAME AFTER "CALL"
157: INVALID PARAMETER LIST
158: MISSING LABEL AFTER "$"
159: MISSING OR INVALID EXPRESSION AFTER "("
160: MISSING ")" AFTER EXPRESSION
161: MISSING "(" AFTER IDENTIFIER @
162: MISSING "TAPE" AFTER "INPUT" OR "OUTPUT"
163: MISSING OR INVALID FILE NUMBER
164: MISSING "," AFTER FORMAT OR FILE NUMBER
165: MISSING OR INVALID FORMAT
166: MISSING "(" AFTER "FREE" OR "BINARY"
167: MISSING ")" BEFORE I/O LIST
168: MISSING LABEL IN OPTION
169: MISSING EXPRESSION IN OPTION
170: MISSING OR INVALID I/O LIST ITEM
171: INVALID STATEMENT LABEL AFTER "ASSIGN"
172: MISSING "TO" AFTER STATEMENT LABEL IN "ASSIGN" STATEMENT
173: MISSING OR INVALID IDENTIFIER AFTER "TO"
174: MISSING OR INVALID FILE NUMBER
175: MISSING OR INVALID FILE NAME
176: SECOND FILE NAME MISSING OR INVALID
177: MISSING OR INVALID ARRAY NAME
178: MISSING "," AFTER ARRAY NAME
179: MISSING OR INVALID TASK NAME
180: INVALID EVENT LIST
181: MISSING OR INVALID EXPRESSION AFTER UNARY OPERATOR @
182: INVALID PARAMETER LIST FOR FUNCTION @
183: DO STATEMENT ILLEGAL AS OBJECT OF LOGICAL IF STATEMENT
201: ILLEGAL DUMMY ARGUMENT FOR STATEMENT FUNCTION
202: ONLY ONE OVERLAY NAME ALLOWED PER PROGRAM UNIT
203: ONLY ONE PROGRAM UNIT IDENTIFIER ALLOWED
204: CONFLICTING OR DUPLICATE DECLARATION OF VARIABLE @
```

```
205:  MISSING DIMENSION LIST FOR VARIABLE @
206:  ARRAY @ DIMENSIONED MORE THAN ONCE
207:  INVALID OPTION
208:  STATEMENT NUMBER @ IS NOT A FORMAT STATEMENT
209:  NEXT EXECUTABLE STATEMENT CANNOT BE REACHED
210:  STATEMENT NUMBER @ IS DEFINED MORE THAN ONCE
211:  ATTEMPT TO DIMENSION PARAMETER @
212:  DUPLICATE TYPE DECLARATION OF VARIABLE @
213:  FORMAT LABEL @ IS DEFINED MORE THAN ONCE
214:  STATEMENT LABEL @ HAS BEEN USED AS A FORMAT LABEL
215:  ILLEGAL CALL OF ARRAY, PARAMETER, OR STATEMENT FUNCTION @
216:  ILLEGAL CALL OF VARIABLE @
217:  DUPLICATE DUMMY ARGUMENT @ FOR STATEMENT FUNCTION
218:  ILLEGAL CALL OF OVERLAY @
219:  ARRAY DIMENSIONED WITH LOWER BOUND GREATER THAN UPPER BOUND
301:  ILLEGAL TYPE CONVERSION FOR EXPRESSION OR VARIABLE @
302:  TOO FEW ARGUMENTS PROVIDED FOR "FLD" OR "BYTE"
303:  TOO MANY ARGUMENTS PROVIDED FOR "FLD" OR "BYTE"
304:  WRONG NUMBER OF SUBSCRIPTS FOR ARRAY @
305:  SINGLE-SUBSCRIPT REFERENCE OF ARRAY @ ALLOWED ONLY IN EQUIVALENCE
307:  ILLEGAL ARGUMENT FOR INTRINSIC FUNCTION @
308:  WRONG NUMBER OF ARGUMENTS FOR INTRINSIC FUNCTION @
309:  CONSTANT OR STRING LITERAL APPEARS IN INPUT LIST
310:  NOT ENOUGH ARGUMENTS FOR STATEMENT FUNCTION
311:  TOO MANY ARGUMENTS FOR STATEMENT FUNCTION
312:  LOGICAL EXPRESSION NOT ALLOWED IN ARITHMETIC IF STATEMENT
313:  COMPLEX EXPRESSION NOT ALLOWED IN ARITHMETIC IF STATEMENT
314:  NEXT EXECUTABLE STATEMENT CANNOT BE REACHED
315:  LOGICAL EXPRESSION REQUIRED IN LOGICAL IF STATEMENT
316:  SUBROUTINE OR LABEL NAME @ USED ON LEFT OF "="
317:  ILLEGAL EXPRESSION
318:  ILLEGAL DATA TYPE FOR OPERATOR
318:  ILLEGAL REFERENCE ON LEFT-HAND-SIDE OF "="
319:  ILLEGAL "FLD" OR "BYTE" REFERENCE ON LEFT-HAND-SIDE OF "="
320:  ILLEGAL USE OF SUBPROGRAM OR ARRAY NAME @
321:  ILLEGAL EQUIVALENCING OF FUNCTION @
322:  ILLEGAL EQUIVALENCING OF STATEMENT FUNCTION @
323:  ILLEGAL USE OF PARAMETER @ AS ARRAY OR FUNCTION NAME
324:  RECURSIVE DEFINITION OF STATEMENT FUNCTION @
325:  ILLEGAL TYPE CONVERSION
326:  ILLEGAL DATA TYPE FOR OPERATOR
327:  RECURSIVE DEFINITION OF PARAMETER @
328:  ILLEGAL USE OF OVERLAY NAME @ AS FUNCTION NAME
401:  STATEMENT NUMBER @ IS NOT DEFINED AS AN EXECUTABLE STATEMENT
404:  DO LOOP TERMINUS, LABEL @,  CANNOT BE DO STATEMENT
405:  VARIABLE @ IS THE INDEX VARIABLE FOR AN OUTER LOOP
406:  DO LOOP TERMINUS, LABEL @, DOES NOT EXIST
407:  ILLEGAL ASSIGNMENT TO INDEX VARIABLE @
408:  DO LOOP ENDING AT STATEMENT @ IS NOT NESTED PROPERLY
410:  DO LOOP TERMINUS, LABEL @, PRECEEDS DO STATEMENT REFERENCING IT
411:  STATEMENT NUMBER ASSIGNED TO PARAMETER "@"
412:  VARIABLE @ HAS NEVER BEEN ASSIGNED A VALUE
501:  POSSIBLE UNDECLARED PARAMETER @ IN DIMENSION OR SUBSCRIPT
502:  ARRAY @ DIMENSIONED INCORRECTLY OR TOO LARGE
503:  CONFLICTING USE OF PARAMETER @
504:  CONFLICTING USE OF SUBPROGRAM NAME @
505:  CONFLICTING USE OF STATEMENT FUNCTION NAME @
506:  MULTIPLE DECLARATION OF VARIABLE @
507:  BACKWARDS EXTENSION OF COMMON BLOCK BY VARIABLE @
508:  VARIABLE @ IS ASSIGNED TO BOTH COMMON AND STATIC
509:  DUMMY ARGUMENT @ HAS BEEN ASSIGNED TO COMMON
510:  DUMMY ARGUMENT @ HAS BEEN EQUIVALENCED
511:  INCONSISTENT EQUIVALENCING OF VARIABLE @
512:  DUMMY ARGUMENT @ HAS BEEN DATA INITIALIZED
513:  DUMMY ARGUMENT @ HAS BEEN ASSIGNED TO STATIC
514:  MULTIPLE OCCURRENCES OF VARIABLE @ IN DUMMY ARGUMENT LIST
515:  BLOCK DATA SUBPROGRAM CONTAINS EXECUTABLE CODE
```

```
516:  INSUFFICIENT SPACE REMAINING FOR VARIABLE OR ARRAY @
517:  TOO MUCH SPACE REQUIRED BY DECLARED VARIABLES AND ARRAYS
701:  TOO FEW DATA ITEMS IN DATA INITIALIZATION LIST
702:  TOO MANY DATA ITEMS IN DATA INITIALIZATION LIST
703:  ILLEGAL ITEM @ IN DATA INITIALIZATION LIST
704:  INVALID REPEAT COUNT @ FOR DATA ITEM
705:  ILLEGAL EXPRESSION @ IN DIMENSION OR SUBSCRIPT
706:  VARIABLE @ IN SUBSCRIPT EXPRESSION IS UNDEFINED
707:  ILLEGAL ITERATION COUNT SPECIFIED FOR IMPLIED DO
708:  ILLEGAL ITEM AS INDEX OF IMPLIED DO
709:  INDEX VARIABLE @ DIMENSIONED OR NOT TYPE INTEGER
710:  VARIABLE @ IS THE INDEX FOR AN OUTER LOOP
711:  STRING CONSTANT @ TOO SHORT
712:  STRING CONSTANT @ TOO LONG
713:  ATTEMPT TO INITIALIZE LOGICAL VARIABLE WITH NON-LOGICAL CONSTANT
714:  ATTEMPT TO INITIALIZE NON-LOGICAL VARIABLE WITH LOGICAL CONSTANT
715:  TOO MUCH SPACE REQUIRED BY STACK VARIABLES AND TEMPORARIES
```

End of Appendix

Appendix C
Runtime Math Errors

The following is a list of runtime error messages that result when you use the mathematical routines described in Chapter 8 incorrectly. Variables and symbols used in the list have the following meanings:

n	Integer Argument
x	Real or Double Precision Argument
z	Complex or Double Precision Complex Argument
H	log 16**63 ($\approx$175)

Error Message	Primary Source Routine	Primary Cause	Secondary Source Routine*	Secondary Cause
ILLEGAL ARGUMENT TO SQRT	SQRT DSQRT	$x < 0$	--	--
ILLEGAL ARGUMENT TO EXP	EXP	$\|x\| \geq H$	SHCH POWR CEXP CSNCS CPOWR	$\|x\| \geq H$ $\|x_1 * \ln(x_2)\| \geq H$ $\|Real(z)\| \geq H$ $\|Real(z)\|$ or $\|Imag(z)\| \geq H$ $\|Real(z)\| \geq H$
	DEXP	$\|z\| \geq H$	DSHCH DPOWR DCEXP DCSNC DCPOWR	$\|x\| \geq H$ $\|x_1 * \ln(x_2)\| \geq H$ $\|Real(z)\| \geq H$ $\|Real(z)\|$ or $\|Imag(z)\| \geq H$ $\|Real(z)\| \geq H$
ILLEGAL ARGUMENT FOR LOG	LOG	$x \leq 0$	LOG10 CLOG CPOWR	$x \leq 0$ $\|z\| \approx 0$ $\|z_1\|$ or $z_2 \approx 0$
	DLOG	$x \leq 0$	DLOG10 DCLOG DCPOWR	$x \leq 0$ $\|z\| \approx 0$ $\|z_1\|$ or $z_2 \approx 0$
ILLEGAL EXPONENTATION	IPWRI DPWRI PWRI	$n = 0$ and $n \leq 0$	--	--
	POWR DPOWR	$x_1 = 0$ and $x_2 \leq 0$ $x_1 \leq 0$ and $x_2 \neq 0$	--	--
REQUEST FOR ATAN2 AT ORIGIN	ATAN2	$x_1 = x_2 = 0$	CLOG CPOWR	$\|z\| \approx 0$ $\|z_1\|$ or $\|z_2 \approx 0$
	DATN2	$x_1 = x_2 = 0$	DCLOG DCPOWR	$\|z\| \approx 0$ $\|z_1\|$ or $\|z_2\| \approx 0$
ILLEGAL ARGUMENT FOR ASIN OR ACOS	ASAC DASAC	$\|x\| > 1$	--	--
INTEGER OVERFLOW ON CONVERSION	INT	$\|x\| > 32767$	--	--

*NOTE: A secondary source routine is one that eventually calls a primary source routine.

End of Appendix

Appendix D
ASCII Character Set

To find the *octal* value of a character, locate the character, and combine the first two digits at the top of the character's column with the third digit in the far left column.

LEGEND:

Character code in decimal
EBCDIC equivalent hexadecimal code
Character

		10_
0	64	@
	7C	

OCTAL	00_			01_			02_			03_		
0	0	00	NUL	8	16	BS (BACK-SPACE)	16	10	DLE ↑P	24	18	CAN ↑X
1	1	01	SOH ↑A	9	05	HT (TAB)	17	11	DC1 ↑Q	25	19	EM ↑Y
2	2	02	STX ↑B	10	15	NL (NEW LINE)	18	12	DC2 ↑R	26	3F	SUB ↑Z
3	3	03	ETX ↑C	11	0B	VT (VERT. TAB)	19	13	DC3 ↑S	27	27	ESC (ESCAPE)
4	4	37	EOT ↑D	12	0C	FF (FORM FEED)	20	3C	DC4 ↑T	28	1C	FS ↑\
5	5	2D	ENQ ↑E	13	0D	RT (RETURN)	21	3D	NAK ↑U	29	1D	GS ↑]
6	6	2E	ACK ↑F	14	0E	SO ↑N	22	32	SYN ↑V	30	1E	RS ↑↑
7	7	2F	BEL ↑G	15	0F	SI ↑O	23	26	ETB ↑W	31	1F	US ↑←

OCTAL	04_			05_			06_			07_		
0	32	40	SPACE	40	4D	(	48	F0	0	56	F8	8
1	33	5A	!	41	5D	)	49	F1	1	57	F9	9
2	34	7F	'' (QUOTE)	42	5C	*	50	F2	2	58	7A	:
3	35	7B	#	43	4E	+	51	F3	3	59	5E	;
4	36	5B	$	44	6B	, (COMMA)	52	F4	4	60	4C	<
5	37	6C	%	45	60	-	53	F5	5	61	7E	=
6	38	50	&	46	4B	. (PERIOD)	54	F6	6	62	6E	>
7	39	7D	' (APOS)	47	61	/	55	F7	7	63	6F	?

OCTAL	10_			11_			12_			13_		
0	64	7C	@	72	C8	H	80	D7	P	88	E7	X
1	65	C1	A	73	C9	I	81	D8	Q	89	E8	Y
2	66	C2	B	74	D1	J	82	D9	R	90	E9	Z
3	67	C3	C	75	D2	K	83	E2	S	91	8D	[
4	68	C4	D	76	D3	L	84	E3	T	92	E0	\
5	69	C5	E	77	D4	M	85	E4	U	93	9D	]
6	70	C6	F	78	D5	N	86	E5	V	94	5F	↑ or ^
7	71	C7	G	79	D6	O	87	E6	W	95	6D	← or _

OCTAL	14_			15_			16_			17_		
0	96	79	` (GRAVE)	104	88	h	112	97	p	120	A7	x
1	97	81	a	105	89	i	113	98	q	121	A8	y
2	98	82	b	106	91	j	114	99	r	122	A9	z
3	99	83	c	107	92	k	115	A2	s	123	C0	{
4	100	84	d	108	93	l	116	A3	t	124	4F	|
5	101	85	e	109	94	m	117	A4	u	125	D0	}
6	102	86	f	110	95	n	118	A5	v	126	A1	~ (TILDE)
7	103	87	g	111	96	o	119	A6	w	127	07	DEL (RUBOUT)

SD-00217 Character code in octal at top and left of charts.

↑ means CONTROL

End of Appendix

Index

Within this index, *f* or *ff* after a page number means *and the following page* or *pages.* In addition, primary page references for each topic are listed first.

□ 1-4, 6-19ff, D-1
" 1-4, 2-6, D-1
$ 1-4, 8-3, 8-6, D-1
' 1-4, 2-6, D-1
( ) 1-4, 2-13, 2-15, D-1
* 1-4, 2-10f, 2-15, D-1
+ 1-4, 2-10f, 2-15, 6-19f, D-1
, 1-4, 6-19, D-1
- 1-4, 2-10f, 2-15, 6-19, D-1
. 1-4, D-1
/ 1-4, 2-10f, 2-13, 2-15, 6-19, 6-21, D-1
: 1-4, 7-2, D-1
< 1-4, 2-6, D-1
> 1-4, 2-6, D-1
= 1-4, 2-13, 3-1ff, D-1
** 2-13, 2-15, D-1

A format code 6-12
ABS library function 8-6, 8-8
ACCEPT statement 5-9
ACOS library function 8-6f
addition 2-10f, 2-13
actual arguments
  function reference 8-3f
  function subprogram 8-3f
  statement function 8-1f
  task 9-3f
actual array declarator 7-2
adjustable dimensions 7-2
AIMAG library function 8-8
AINT library function 8-8
ALOG library function 8-9
ALOG10 library function 8-9
alphanumeric data 6-12ff, 6-22
alternate return 8-3, 8-6
AMAX0 library function 8-9
AMAX1 library function 8-9
AMIN0 library function 8-9
AMIN1 library function 8-9
AMOD library function 8-9
.AND. 2-10, 2-13ff
ANDing of bits 8-11
angle brackets 1-4, 2-6, D-1
ANINT library function 8-8
ANTICIPATE statement 9-7
apostrophe 1-4, 2-6, D-1
argument
  external subprogram 7-9
  function and subroutine subprogram 8-3f
  library function 8-6f
  task 9-3f
arithmetic assignment statement 3-1f
arithmetic expressions
  definition 2-10
  in statement functions 8-1f
  naming 7-8
  rules for evaluation 2-11ff
arithmetic IF statement 4-3
array
  data typing 7-3
  declarators 7-2
  definition 2-9
  dimensioning 7-1ff
  EQUIVALENCEing 7-6f
  in COMMON storage 7-4f
  in a FORMAT statement 5-1f
  in I/O lists 5-3
  in STATIC storage 7-6
  naming 2-8
  storage 2-9f
  subscripting 2-9
ASCII character set 1-4, D-1
ASIN library function 8-6f
ASSIGN statement 3-3
assigned GOTO statement 4-2
assignment
  arithmetic assignment statement 3-1
  ASSIGN statement 3-3
  conversion rules 3-2
  file 5-13
  logical assignment statement 3-3
  of variables 2-9
ATAN library function 8-6f
ATAN2 library function 8-6f
attributes, file 5-11ff
auxiliary I/O
  BACKSPACE statement 5-15
  CLOSE statement 5-14
  DELETE statement 5-14
  ENDFILE statement 5-14
  OPEN statement 5-11ff
  RENAME statement 5-15
  REWIND statement 5-15

B format code 6-6
BACKSPACE statement 5-15
base operand 2-11
binary
  I/O 6-6
  operator 2-10
bit manipulation 8-12f
blank
  ASCII character 1-4, D-1
  common storage 7-4f
  lines 1-4
  program translation of 1-4
block, common
  definition 7-4
  EQUIVALENCEing data in 7-6f
  extending the block 7-7
  data typing 2-8
BLOCK DATA subprogram 8-14
Boolean operators 2-10, 8-11
bounds, array 2-9, 7-2
BYTE function
  definition 8-14
  in an assignment statement 3-1
  in a statement function 8-2
byte manipulation 8-14

C (comment indicator) 1-4
CABS library function 8-8
CALL statement 8-5
card reader 5-13
carriage control
  formatted I/O 6-19f
  multiple-record formatting 6-21f
  suppression of carriage return (Z) 6-19f
  tabulation 1-4, 6-16
carriage return
  ASCII character 2-6
  in free-formatted I/O 5-8f
  suppression of 6-19f
CCOS library function 8-7
CCOSH library function 8-7
CEIL library function 8-8
CEXP library function 8-8
character
  FORTRAN set 1-4, D-1
  position 1-3f
CLOG library function 8-9
CLOSE statement 5-14
CMPLX library function 8-6, 8-8
colon
  ASCII character 1-4, D-1
  in array declarators 7-2
column positioning control 6-16
comma
  ASCII character 1-4
  in complex numbers 2-4
  in format specification 6-19
comment line 1-4
COMMON
  array dimensioning in 7-4f
  associating variables in 2-9
  blank 7-4f
  data typing 2-8
  EQUIVALENCEing data in 7-6f
  extending a block in 7-7
  named 7-4f
  statement 7-4f
compile-time options 7-12
compiler error and warning messages B-1ff
COMPILER statement 7-12
complex data
  accessing imaginary parts 7-6
  assigning constants to 7-11
  constants 2-4
  conversion 2-11f, 3-2
  in a FUNCTION statement 8-3
  in an I/O list 6-21
  in a Type-statement 7-3
  storage 2-1
components of FORTRAN 5 2-1ff
  arrays 2-9f
  constants 2-2ff
  data types 2-1
  expressions 2-10ff
  function references 2-15
  symbolic names 2-8
  variables 2-9
computed GOTO statement 4-2
CONJG library function 8-6, 8-8
console, I/O 5-13
constant dimensions 7-2
constants
  complex 2-4
  double precision 2-3, 7-12
  double precision complex 2-5
  Hollerith 2-6ff
  integer 2-2
  list in DATA statement 7-10f
  logical 2-6
  naming 7-8
  octal 2-2
  real 2-2f
  string 2-6ff
contiguous files 5-11
continuation line 1-3
CONTINUE statement
  definition 4-6
  in a DO loop 4-5
control block, task 9-2
control characters 2-6f

control statements
  arithmetic IF 4-3
  assigned GOTO 4-2
  computed GOTO 4-2
  CONTINUE 4-6
  DO 4-4f
  END 4-7
  IF 4-3
  GOTO 4-1f
  logical IF 4-3
  PAUSE 4-6
  STOP 4-7
  unconditional GOTO 4-1
control variable 4-4
conversion, I/O
  alphanumeric 6-12ff
  binary 6-6
  double precision 6-9
  exponential 6-8, 6-10
  generalized 6-10
  hexadecimal 6-5
  integer 6-3, 6-10
  logical 6-11, 6-10
  octal 6-4
  real 6-7, 6-10
COS library function 8-6f
COSH library function 8-6f
CSIN library function 8-7
CSINH library function 8-7
CSQRT library function 8-9
CTAN library function 8-7
CTANH library function 8-7
CXDCX library function 8-8

D format code 6-9
D in double precision constants 2-3
DABS library function 8-8
DACOS library function 8-7
DASIN library function 8-7
data
  alphanumeric 6-12ff
  binary 6-6
  complex 2-4
  double precision 2-3
  double precision complex 2-5
  formatting of 6-1ff
  hexadecimal 6-5
  Hollerith 2-6ff, 3-1, 6-15, 7-11
  initialization 7-1ff
  integer 2-2
  logical 2-6
  octal 2-2, 6-4
  real 2-2f
  string 2-6ff, 3-1, 6-12ff, 7-11
  transfers 5-2f, 5-5ff
  typing of 2-1ff
DATA statement 7-10f
data types 2-1ff
DATAN library function 8-7
DATAN2 library function 8-7
DBLE library function 8-8
DCABS library function 8-8
DCCOS library function 8-7
DCCSH library function 8-7
DCEIL library function 8-8
DCEXP library function 8-8
DCLOG library function 8-9
DCMPLX library function 8-8
DCONJG library function 8-8
DCOS library function 8-7
DCOSH library function 8-7
DCSIN library function 8-7
DCSNH library function 8-7
DCSQRT library function 8-9
DCTAN library function 8-7
DCTNH library function 8-7
DCXCX library function 8-8
DDIM library function 8-8
decimal point
  ASCII character 1-4, D-1
  in real constants 2-2f
declarator, array
  actual 7-2
  dummy 7-2
DECODE statement 5-7
DELETE statement 5-14
descriptor, field 6-1ff
DEXP library function 8-8
DFLOAT library function 8-6, 8-8
DFLOOR library function 8-8
DIM library function 8-6, 8-8
DIMAG library function 8-8
dimension
  adjustable 7-2
  array 2-9
  constant 7-2
  in a COMMON statement 7-4f
  in a Type-statement 7-3
DIMENSION statement 7-1f
DINT library function 8-8
direct file access 5-2
division 2-10f, 2-13
DLOG library function 8-9
DLOG10 library function 8-9
DMAX1 library function 8-9
DMIN1 library function 8-9
DMOD library function 8-9
DO-implied list 5-4
DO loop
  extended range 4-5
  nested 4-4f
  range of 4-4
  restrictions 4-5
  termination 4-4
DO statement 4-4f
dollar sign
  alternate return 8-3ff
  ASCII character 1-4, D-1

double precision data
  constant 2-3
  conversion 2-11f
  in a FORMAT statement 6-9
  in a Type-statement 7-3
  scale factoring 6-17f
  storage 2-1
DOUBLE PRECISION option 7-12
double precision complex data
  constant 2-5
  conversion 2-11f
  in a FORMAT statement 6-9, 6-21
  in a Type-statement 7-3
  scale factoring 6-17f
  storage 2-1
DREAL library function 8-9
DSIGN library function 8-9
DSIN library function 8-7
DSINH library function 8-7
DSQRT library function 8-9
DTAN library function 8-7
DTANH library function 8-7
dummy arguments
  in array dimensioning 7-1f
  in a FUNCTION statement 8-3
  in function and subroutine subprograms 8-3f
  in multitasking 9-3f
  in a statement function 8-1f
  in a SUBROUTINE statement 8-4

E format code 6-8
E in real constants 2-2f
element, array 2-9f
editing field descriptors
  T format code 6-16
  X format code 6-16
ENCODE statement 5-6
end
  of function subprogram 8-2
  of program unit 1-3
  of subroutine subprogram 8-4
END statement 4-7
ENDFILE statement 5-14
.EQ. 2-14f
EQUIVALENCE statement 7-6f
error messages, compiler B-1ff
event 9-3
excess 64 representation 2-3
exclusive OR 8-11
EXECUTABLE statements
  ACCEPT 5-9
  ANTICIPATE 9-7
  ASSIGN 3-3
  assignment, arithmetic 3-1
  assignment, logical 3-3
  BACKSPACE 5-15
  CALL 8-5
  CLOSE 5-14
  CONTINUE 4-6
  DECODE 5-7
  definition of 1-3
  DELETE 5-14
  DO 4-4f
  ENCODE 5-6
  ENDFILE 5-14
  GOTO, assigned 4-2
  GOTO, computed 4-2
  GOTO, unconditional 4-1
  IF, arithmetic 4-3
  IF, logical 4-3
  INCLUDE 1-4f
  KILL 9-5
  OPEN 5-11ff
  PAUSE 4-6
  PRINT 5-7
  PUNCH 5-7
  READ 5-5, 5-7
  READ BINARY 5-10
  READ FREE 5-8
  READ INPUT TAPE 5-5
  READ TAPE 5-10
  RENAME 5-15
  RETURN 8-5f
  REWIND 5-15
  STOP 4-7
  SUSPEND 9-6
  TASK 9-3f
  TYPE 5-9
  WAIT 9-5
  WAKEUP 9-6
  WRITE 5-6
  WRITE BINARY 5-10
  WRITE FREE 5-8
  WRITE OUTPUT TAPE 5-6
  WRITE TAPE 5-10
EXP library function 8-6, 8-8
explicit data typing 7-3

exponentiation
  arithmetic operator 2-10
  in a complex constant 2-4
  in a double precision constant 2-3
  in a double precision complex constant 2-5
  in an expression 2-12
  in a FORMAT statement 6-8ff
  in a real constant 2-2f
  operator precedence 2-13
expression
  arithmetic 2-10ff
  assigning names to 2-8, 7-8
  data typing 2-11f
  definition 2-10
  in an assignment statement 3-1ff
  logical 2-14f
  relational 2-14f
extended DO range 4-5
extending a COMMON block 7-7
EXTERNAL statement 7-9

F format code 6-7
factor, scale 6-17f
.FALSE. 2-6
field descriptor
  alphanumeric (A,R) 6-12f
  binary (B) 6-6
  column positioning (X) 6-16
  double precision (D) 6-9
  editing (X,T) 6-16
  explicit exponent (E) 6-8
  generalized (G) 6-10
  hexadecimal (Y) 6-5
  Hollerith (H) 6-15
  integer (I) 6-3
  logical (L) 6-11
  numeric (I,Y,O,B,F,E,D,G) 6-3ff
  octal (O) 6-4
  real (F) 6-7
  string (" ",' ',S) 6-14f
  tabulation (T) 6-16
field separators 6-19
field width (w) 6-1ff
file
  access 5-2
  assignment 5-13
  backspacing 5-15
  closing 5-14
  contiguous 5-11
  creation 5-11
  deletion 5-14
  opening 5-11ff
  organization 5-3
  preconnections 5-13
  prepared for printing 5-11, 6-19f
  random 5-11
  renaming 5-15
  rewinding 5-15
  sequential 5-11
FLD function
  definition 8-13
  in an assignment statement 3-1
  in a statement function 8-2
FLOAT library function 8-6, 8-8
FLOOR library function 8-8
form feed 2-6, 6-19
format specification 6-1
formatted I/O
  DECODE statement 5-7
  definition 5-1f
  ENCODE statement 5-6
  FORMAT statement 6-1ff
  PRINT statement 5-7
  PUNCH statement 5-7
  READ statement 5-5, 5-7
  summary of rules 6-22
  WRITE statement 5-6
formatting
  carriage control in 6-19f
  complex data 6-21
  data transfers 5-2
  file organization 5-3
  format specification 5-1, 6-1
  group repeat specifications 6-18
  identifiers 5-1
  I/O statement association 6-21
  multiple records 6-21f
  scale factoring 6-17f
  summary of rules 6-22
free-formatted I/O
  ACCEPT statement 5-9
  definition 5-2ff
  READ FREE statement 5-8
  TYPE statement 5-9
  WRITE FREE statement 5-8
FREE option 7-12
function, intrinsic 8-6ff, 2-8
function, generic 8-6ff, 2-8
function, library 8-6ff
function reference 8-1, 8-3f
function subprogram
  arguments 8-3
  data typing 7-3
  declared external 7-9
  definition 8-2
  naming 2-8
  RETURN statement 8-5f
FUNCTION statement 8-3

G format code 6-10
.GE. 2-14f
generalized field descriptor 6-10
generic function
  definition 8-6ff
  disabling generic selection 7-9
  symbolic name 2-8

GOTO statements
  assigned GOTO 4-2
  computed GOTO 4-2
  unconditional GOTO 4-1
.GT. 2-14f

H format code 6-15
hexadecimal format code 6-5
Hollerith constant
  angle brackets in 2-6ff
  association with I/O list elements 6-21
  definition 2-6ff
  in an assignment statement 3-1
  in a DATA statement 7-10f
  in expressions 2-8
  in a FORMAT statement 6-15
  storage 2-7f

I format code 6-3
IABS library function 8-8
IAND function 8-11
ICLR routine 8-12
identification number, task 9-2
identifier, format 5-1f
IDIM library function 8-8
IDINT library function 8-8
IEOR function 8-11
IF statement
  arithmetic IF 4-3
  logical IF 4-3
IFIX library function 8-8
IMAG library function 8-6, 8-8
imaginary number
  in complex constants 2-4
  in double precision complex constants 2-5
IMPLICIT statement 7-3
inactive chain 9-2
INCLUDE statement 1-5
incremental plotter 5-13
initialization, data 7-1ff, 7-10f
input, see I/O
INT library function 8-6, 8-8
integer data
  constant 2-2
  conversion 3-2
  in a FORMAT statement 6-3, 6-10
  in a Type-statement 7-3
  logical word operations 8-11
  single bit operations 8-12
  storage 2-1
interaction, programmer/program 5-9
intrinsic function
  symbolic name 2-8
  definition 8-6

I/O
  auxiliary statements 5-11ff
  conversational 5-9
  conversion 6-1ff
  data transfers 5-2f
  file access 5-2
  file organization 5-3
  format identifiers 5-1f
  format specification 5-1f, 6-1
  formatted statements 5-5ff
  free-formatted statements 5-8f
  lists 5-3f, 6-21
  records 5-2
  unformatted statements 5-10
  unit numbers 5-1
I/O statements
  ACCEPT 5-9
  BACKSPACE 5-15
  CLOSE 5-14
  DECODE 5-7
  DELETE 5-14
  ENCODE 5-6
  ENDFILE 5-14
  OPEN 5-11ff
  PRINT 5-7
  PUNCH 5-7
  READ 5-5, 5-7
  READ BINARY 5-10
  READ FREE 5-8
  READ INPUT TAPE 5-5
  READ TAPE 5-10
  RENAME 5-15
  REWIND 5-15
  TYPE 5-9
  WRITE 5-6
  WRITE BINARY 5-10
  WRITE FREE 5-8
  WRITE OUTPUT TAPE 5-6
  WRITE TAPE 5-10
IOR function 8-11
ISET routine 8-12
ISHFT function 8-11
ISIGN library function 8-9
ITEST function 8-12
IXOR function 8-11

K following octal constant 2-2
keyboard, console 5-13
KILL statement 9-5

L format code 6-11
label, statement 1-4
.LE. 2-14f

library functions
ABS 8-6, 8-8
ACOS 8-6f
AIMAG 8-8
AINT 8-6, 8-8
ALOG 8-9
ALOG10 8-9
AMAX0 8-9
AMAX1 8-9
AMIN0 8-9
AMIN1 8-9
AMOD 8-9
ANINT 8-8
ASIN 8-6f
ATAN 8-6f
ATAN2 8-6f
CABS 8-8
CCOS 8-7
CCOSH 8-7
CEIL 8-8
CEXP 8-8
CLOG 8-9
CMPLX 8-6, 8-8
CONJG 8-6, 8-8
COS 8-6f
COSH 8-6f
CSIN 8-7
CSINH 8-7
CSQRT 8-9
CTAN 8-7
CTANH 8-7
CXDCX 8-8
DABS 8-8
DACOS 8-7
DASIN 8-7
DATAN 8-7
DATAN2 8-7
DBLE 8-8
DCABS 8-8
DCCOS 8-7
DCCSH 8-7
DCEIL 8-8
DCEXP 8-8
DCLOG 8-9
DCMPLX 8-8
DCONJG 8-8
DCOS 8-7
DCOSH 8-7
DCSIN 8-7
DCSNH 8-7
DCSQRT 8-9
DCTAN 8-7
DCTNH 8-7
DCXCX 8-8
DDIM 8-8
DEXP 8-8
DFLOAT 8-6, 8-8
DFLOOR 8-8
DIM 8-6, 8-8
DIMAG 8-8
DINT 8-8
DLOG 8-9
DLOG10 8-9
DMAX1 8-9
DMIN1 8-9
DMOD 8-9
DNINT 8-8
DREAL 8-9
DSIGN 8-9
DSIN 8-7
DSINH 8-7
DSQRT 8-9
DTAN 8-7
DTANH 8-7
EXP 8-6, 8-8
FLOAT 8-6, 8-8
FLOOR 8-8
IABS 8-8
IDIM 8-8
IDINT 8-8
IFIX 8-8
IMAG 8-6, 8-8
INT 8-6, 8-8
ISIGN 8-9
LOG 8-6, 8-9
LOG10 8-6, 8-9
MAX 8-6, 8-9
MAX0 8-9
MAX1 8-9
MIN 8-6, 8-9
MIN0 8-9
MIN1 8-9
MOD 8-6, 8-9
REAL 8-6, 8-9
SIGN 8-6, 8-9
SIN 8-6f
SINH 8-6f
SNGL 8-9
SQRT 8-6, 8-9
TAN 8-6f
TANH 8-6f
line
blank 1-4
comment 1-4
continuation 1-3
definition 1-3
initial 1-3
optionally compiled 1-4
termination of 1-3f
line feed 2-6, 6-19
line printer 5-13
line-oriented file 5-3
list, do-implied 5-4
list, I/O 5-3f, 6-21
LOG library function 8-6, 8-9
LOG10 library function 8-6, 8-9
logical assignment statement 3-3

logical data
  constant 2-6
  expressions 2-14f
  in an assignment statement 3-3
  in a FORMAT statement 6-11
  in a Type-statement 7-3
  operators 2-14
  storage 2-1
  truth table 2-14
logical field descriptor 6-11
logical IF statement 4-3
logical word operations
  IAND 8-11
  IEOR 8-11
  IOR 8-11
  ISHFT 8-11
  IXOR 8-11
  NOT 8-11
loop, DO 4-4f
lower bound, array 2-9f
.LT. 2-14f

main program unit 1-3
mathematical expressions, see arithmetic expressions
MAX library function 8-6, 8-9
MAX0 library function 8-9
MAX1 library function 8-9
MIN library function 8-6, 8-9
MIN0 library function 8-9
MIN1 library function 8-9
mixed-mode expression evaluation 2-11f
MOD library function 8-6, 8-9
multiple-record formatting 6-23f
multiplication 2-10f
multitask programming 9-1ff

name rule 2-8
name, symbolic 2-8
named common storage
  data-initializing elements in 7-11
  definition 7-4
  in a BLOCK DATA subprogram 8-14
.NE. 2-14f
nested
  DO loops 4-4f
  DO-implied lists 5-4
  parentheses in formatting 6-21ff
new-line character
  ASCII notation 2-6
  in free-formatted I/O 5-8f
  suppression of 6-19f
nonexecutable statements
  BLOCK DATA 8-14
  COMMON 7-4f
  COMPILER 7-12
  COMPLEX 7-3
  DATA 7-10f
  definition of 1-3
  DIMENSION 7-1f
  DOUBLE PRECISION 7-3
  DOUBLE PRECISION COMPLEX 7-3
  END 4-7
  EQUIVALENCE 7-6f
  EXTERNAL 7-9
  FORMAT 6-1ff
  FUNCTION 8-3
  IMPLICIT 7-3
  INTEGER 7-3
  LOGICAL 7-3
  OVERLAY 7-13
  PARAMETER 7-8
  REAL 7-3
  STATIC 7-6
  SUBROUTINE 8-4
  Type- 7-3
normal return 8-5
NOSTACK option 7-12
NOT function 8-11
.NOT.
  as arithmetic operator 2-10
  as logical operator 2-14
  operator precedence 2-13, 2-15
null byte
  ASCII character 2-6
  in string storage 2-7f
  record padding 5-11f
numeric field descriptors
  B format code 6-6
  D format code 6-9
  E format code 6-8
  F format code 6-7
  G format code 6-10
  I format code 6-3
  O format code 6-4
  Y format code 6-5

O format code 6-4
octal
  ASCII codes 2-6f
  constant 2-2
  format code 6-4
OPEN statement 5-11ff

operands
  arithmetic 2-10f
  base 2-12
  data types 2-11
  exponent 2-12
  logical 2-14
  relational 2-14
operators
  arithmetic 2-10
  binary 2-10
  boolean 2-10
  logical 2-14
  precedence 2-13
  relational 2-14
  unary 2-10
options list, I/O 5-4
.OR.
  as arithmetic operator 2-10
  as logical operator 2-14
  operator precedence 2-13, 2-15
ORing of bits 8-11
output, see I/O
overlay
  data typing 2-8
  declared external 7-9
  naming 7-13
OVERLAY statement 7-13

P attribute 5-11
P scale factor indicator 6-17f
paper tape punch 5-13
paper tape reader 5-13
PARAMETER statement 7-8
parentheses
  ASCII characters 1-4, D-1
  in an arithmetic expression 2-13
  in a format specification 6-19, 6-21f
  in an I/O list 5-4
PAUSE statement 4-6
plotter, incremental 5-13
plus sign
  as an arithmetic operator 2-10ff
  ASCII character 1-4
  in an ACCEPT statement 5-9
  in carriage control 6-19f
precision, data
  complex 2-4
  double precision 2-3
  double precision complex 2-5
  integer 2-2
  real 2-2
preconnections, file 5-13
print-oriented file 5-3
PRINT statement 5-7
printer
  console 5-13
  line 5-13
printing control characters 6-19f
priority, task 9-1f
program
  ANSI variations and extensions 1-1ff
  changing sequential flow 4-1ff
  character set 1-4
  definition 1-3
  elements 1-3f
  INCLUDE statement 1-4f
  main 1-3
  procedure 1-3
  suspending execution 4-6
  tasking 9-1ff
  termination 1-3, 4-7
  unit 1-3
punch
  paper tape 5-13
  TTY 5-13
PUNCH statement 5-7

quotation mark 1-4, 2-6, D-1

R format code 6-13
R with a string constant 2-6ff
random file 5-11
range of a DO 4-4
READ statement
  effect of FREE option on 7-12
  formatted 5-5
  simple formatted 5-7
READ BINARY statement 5-10
READ FREE statement 5-8
READ INPUT TAPE statement 5-5
READ TAPE statement 5-10
reader
  card 5-13
  paper tape 5-13
  TTY tape 5-13
ready task 9-2
real data
  constant 2-2f
  conversion 2-11f
  in a FORMAT statement 6-7ff
  in a Type-statement 7-3
  scale factoring 6-17f
  storage 2-1
REAL library function 8-6, 8-9
record, I/O 5-2
record-oriented file 5-3
reference
  array element 2-9f
  common variable 7-4f
  function 2-15, 8-2f
  overlay 7-13
  subroutine 8-5
relational expression 2-14f
RENAME statement 5-15
repeat specification
  definition 6-18
  in multiple-record formatting 6-21f

replacement operator (=)
  ASCII character 1-4
  in an arithmetic expression 2-10ff
  in an assignment statement 3-1ff
  operator precedence 2-13
return
  alternate 8-3, 8-5f
  carriage 2-6, 5-8f, 6-19f
  from function 8-3, 8-5
  from subroutine 8-4f
  normal 8-5
RETURN statement 8-5f
REWIND statement 5-15
right-justified string data 2-8
runtime math errors C-1f

S format code 6-14
/S switch 2-9
scale factoring 6-17f
semicolon 1-4
sequential file access 5-2
SIGN library function 8-6, 8-9
simple list, I/O 5-3
SIN library function 8-6f
single bit operations
  ICLR 8-12
  ISET 8-12
  ITEST 8-12
SINH library function 8-6f
slash
  ASCII character 1-4
  in an arithmetic expression 2-10ff
  in a FORMAT statement 6-19f, 6-21
  operator precedence 2-13
SNGL library function 8-9
specification, format 6-1, 5-1
specification statements
  COMMON 7-4f
  COMPILER 7-12
  COMPLEX 7-3
  DATA 7-10f
  DIMENSION 7-1f
  DOUBLE PRECISION 7-3
  DOUBLE PRECISION COMPLEX 7-3
  EQUIVALENCE 7-6f
  EXTERNAL 7-9
  IMPLICIT 7-3
  INTEGER 7-3
  LOGICAL 7-3
  OVERLAY 7-13
  PARAMETER 7-8
  REAL 7-3
  STATIC 7-6
  Type- 7-3
SQRT library function 8-6, 8-9
stack
  multitasking allocation 9-4
  static storage 7-6
statement
  assignment 3-1ff
  BLOCK DATA 8-14
  control 4-1ff
  definition 1-3f
  FORMAT 6-1ff
  function 8-1ff
  INCLUDE 1-5
  I/O 5-1ff
  label 1-4
  lines 1-3f
  multitasking 9-1ff
  specification 7-1ff
  subroutine 8-4ff
STATIC
  option 7-12
  statement 7-6
STOP statement 4-7
storage
  arrays 2-9f
  common block 7-4f
  EQUIVALENCEing data in 7-6f
  free-formatted ASCII data 5-8
  Hollerith constants 2-7f
  static 7-6, 7-12
  string constants 2-7f
stream file access 5-2
stream-oriented file 5-3
string
  angle brackets in 2-6ff
  association with I/O list elements 6-21
  definition 2-6ff
  in an assignment statement 3-1
  in a DATA statement 7-10f
  in expressions 2-8
  in a FORMAT statement 6-15
  storage 2-7f
string field descriptors
  A format code 6-12
  Hollerith constant 6-15
  R format code 6-13
  S format code 6-14
  string constant 6-15
subprogram
  associating variables in 2-9
  BLOCK DATA 8-14
  declared external 7-9
  function 8-2ff, 8-5f
  overlay 7-13
  static variables in 7-6
  subroutine 8-4ff
  termination 8-5f
subroutine
  arguments 8-3f
  CALLing a 8-5
  data typing 2-8
  declared external 7-9
  definition 8-4
  RETURNing from 8-5f

SUBROUTINE statement 8-4
subscript, array
  definition 2-9
  in an I/O list 5-3f
  progression 2-9f
subtraction 2-10f, 2-13
SUSPEND statement 9-6
suspended task 9-2
symbolic name
  as a parameter 7-8
  data typing 2-8, 7-3
  in an EXTERNAL statement 7-9

T format code 6-16
tabulation
  ASCII character 2-6
  in formatted I/O 6-16
TAN library function 8-6f
TANH library function 8-6f
tape
  punch 5-13
  reader 5-13
task
  ANTICIPATE a 9-7
  argument passing 9-3f
  control block 9-2
  data typing 2-8
  definition 9-1
  event 9-3
  identification number 9-2
  KILLing a 9-5
  multitask programming 9-1ff
  priority 9-1f
  stack allocation for 9-4
  states 9-2
  SUSPENDing a 9-6
  WAITing for a 9-5
  WAKEUP a 9-6
TASK statement 9-3f
TCB 9-2
termination
  function 8-3, 8-5f
  program 1-3
  statement 1-3f
  subroutine 8-4ff
  task 9-5
.TRUE. 2-6
truth operations 2-14
truth table, logical 2-14
TTY
  punch 5-13
  tape reader 5-13
TYPE statement 5-9
Type-statement 7-3
typeless constants 2-6ff, see also Hollerith and string constants

unary operator 2-10
unconditional GOTO statement 4-1
unformatted I/O statements
  READ BINARY 5-10
  READ TAPE 5-10
  WRITE BINARY 5-10
  WRITE TAPE 5-10
unit
  block data subprogram 8-14
  definition 1-3
  function subprogram 8-2ff
  main program 1-3
  number, I/O 5-1, 5-13
  subroutine subprogram 8-4ff
upper bound, array 2-9

variable
  control variable 4-4
  data typing 2-9, 7-3
  definition 2-9
  DOUBLE PRECISION option on 7-12
  EQUIVALENCEing 7-6f
  in common storage 7-4f
  in static storage 7-6, 7-12
  initializing values 7-10f

WAIT statement 9-5
WAKEUP statement 9-6
warning messages, compiler B-1ff
WRITE statement
  formatted 5-6
  FREE option with 7-12
WRITE BINARY statement 5-10
WRITE FREE statement 5-8
WRITE OUTPUT TAPE statement 5-6
WRITE TAPE statement 5-10

X compile option 1-4
X format code 6-16
/X switch 1-4
.XOR.
  as arithmetic operator 2-10
  as logical operator 2-14
  operator precedence 2-13

Y format code 6-5

Z format code 6-19f
zero, real 4-3

CUT ALONG DOTTED LINE

DataGeneral

Software Documentation Remarks Form

How Do You Like This Manual?

Title ______________________________ No. ____________

We wrote the book for you, and naturally we had to make certain assumptions about who you are and how you would use it. Your comments will help us correct our assumptions and improve our manuals. Please take a few minutes to respond.

If you have any comments on the software itself, please contact your Data General representative. If you wish to order manuals, consult the Publications Catalog (012-330).

Who Are You?

- ☐ EDP Manager
- ☐ Senior System Analyst
- ☐ Analyst/Programmer
- ☐ Operator
- ☐ Other ____________________

What programming language(s) do you use? ____________

How Do You Use This Manual?

(List in order: 1 = Primary use)

____ Introduction to the product
____ Reference
____ Tutorial Text
____ Operating Guide
____ ____________________

Do You Like The Manual?

Yes	Somewhat	No	
☐	☐	☐	Is the manual easy to read?
☐	☐	☐	Is it easy to understand?
☐	☐	☐	Is the topic order easy to follow?
☐	☐	☐	Is the technical information accurate?
☐	☐	☐	Can you easily find what you want?
☐	☐	☐	Do the illustrations help you?
☐	☐	☐	Does the manual tell you everything you need to know?

Comments?

(Please note page number and paragraph where applicable.)

From:

Name ____________ Title ____________ Company ____________

Address ______________________ Date ____________

SD-00742

FOLD DOWN FIRST FOLD DOWN

FIRST
CLASS
PERMIT
No. 26
Southboro
Mass. 01772

BUSINESS REPLY MAIL

No Postage Necessary if Mailed in the United States

Postage will be paid by:

Data General Corporation

Southboro, Massachusetts 01772

ATTENTION: Software Documentation

FOLD UP SECOND FOLD UP

SD-00742A

CUT ALONG DOTTED LINE

Data General

users group Installation Membership Form

Name ______________________ Position ______________________ Date __________

Company, Organization or School ____

Address ______________________ City ______________________ State ________ Zip __________

Telephone: Area Code __________ No. ______________ Ext. ______________

1. Account Category

☐ OEM
☐ End User
☐ System House
☐ Government

2. Hardware

	Qty. Installed	Qty. On Order
M/600		
C/350, C/330, C/300		
S/250, S/230, S/200		
S/130		
AP/130		
CS Series		
N3/D		
Other NOVA		
microNOVA		
Other __________		
(Specify) __________		
__________		

3. Software

☐ AOS ☐ RDOS
☐ DOS ☐ RTOS
☐ SOS ☐ Other

Specify __________

____

4. Languages

☐ Algol ☐ Assembler
☐ DG/L ☐ Interactive
☐ Cobol ☐ Fortran
☐ ECLIPSE Cobol ☐ RPG II
☐ Business BASIC ☐ PL/1
☐ BASIC ☐ Other

Specify __________

____

5. Mode of Operation

☐ Batch (Central)
☐ Batch (Via RJE)
☐ On-Line Interactive

6. Communications

☐ RSTCP ☐ CAM
☐ HASP ☐ 4025
☐ RJE80 ☐ Other
☐ SAM

Specify __________

____

7. Application Description

○ __________

____

____

____

____

____

8. Purchase

From whom was your machine(s) purchased?

☐ **Data General Corp.**
☐ Other
Specify __________

____

9. Users Group

Are you interested in joining a special interest or regional Data General Users Group?

○ __________

____

DG-05810

Data General

Data General Corporation, Westboro, Massachusetts 01581, (617) 366-8911

FOLD FOLD

STAPLE STAPLE

FOLD FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 26 SOUTHBORO, MA. 01772

Postage will be paid by addressee:

DataGeneral

ATTN: Users Group Coordinator

Southboro, Massachusetts 01772