Price: $2.50

CONVERSATIONAL FORTRAN REFERENCE MANUAL

for

SDS 940 COMPUTERS

PRELIMINARY EDITION

90 15 79A

January 1969

SDS

SCIENTIFIC DATA SYSTEMS/701 South Aviation Boulevard/El Segundo, California 90245

Printed in U.S.A.

RELATED PUBLICATIONS

Title	Publication No.
SDS 940 Computer Reference Manual	90 06 40
SDS 940 Terminal User's Guide	90 11 18
SDS 940 Time-Sharing System Technical Manual	90 11 16
SDS 940 QED Reference Manual	90 11 12
SDS 940 FORTRAN II Reference Manual	90 11 10

CONTENTS

APPENDIXES

1. INTRODUCTION

This manual is intended as a reference/operations manual for the SDS 940 Conversational FORTRAN System and assumes the reader is familiar with the general principles of FORTRAN programming and with the 940 Executive System described in the SDS 940 Terminal User's Guide.

SDS Conversational FORTRAN (CF) allows a high degree of man-machine interaction since program compilation, debugging, modification, and execution are all controlled via commands issued by the user at a teletype console.

In addition, CF permits incremental compilation of source program statements; i.e., one or more program statements can be compiled independently. To modify a compiled statement, the user recompiles only that statement. To add text to a program, he issues a compile command which directs the system to insert new material at a specified location.

CF also provides a set of on-line debugging features. The user is allowed to set breakpoints, insert temporary statements that execute but do not become compiled as a permanent part of the program, and then to resume execution at any specified point.

Other features of CF include free-form entry of program statements and the ability to save and restore symbolic and object code. An editing capability (subset of QED) is also provided.

Typographic Conventions

For clarity, several conventions have been used throughout this manual. These are explained below:

1. Underscored copy in an example represents copy generated by the computer. Copy that is not underscored in an example must be typed by the user.

2. The following notations have been used to represent special keys on the teletype:

   (RET) represents the RETURN key.

   (ESC)† represents the ESCAPE key.

   (LF) represents the LINE FEED key.

3. Nonprinting control characters are represented by an alphabetic character and a superscript c (e.g., D^c). The user depresses the alphabetic key and the Control (CTRL) key simultaneously to obtain a nonprinting character.

Operating Procedures

The standard procedure for gaining access to an SDS 940 time-sharing computer center from a teletype terminal is described in the SDS 940 Terminal User's Guide. The publication also includes information concerning the Executive System and the calling of various subsystems available to the terminal user. The following paragraphs summarize the standard procedures as they apply to CF users.

Log-In

To gain access to the computer, the following operating sequence is observed:

1. If the FD-HD (Full Duplex-Half Duplex) switch is present, turn the switch to the FD position. When the teletype is not connected to the computer (a condition sometimes called the Local Mode), this switch must be in the HD position.
2. Press the ORIG (originate) key, which is located at the lower right corner of the console directly under the dial. This key is depressed to obtain a dial tone before dialing the computer.

---

†In some 940 time-sharing systems the ALT MODE key is used instead of the ESCAPE key. Where (ESC) appears in this manual, ALT MODE may be substituted.

3. Dial the computer center number. When the computer accepts your call, the ringing will change to a high-pitched tone. There will then appear on the teletype a request that the user log in:

   PLEASE LOG IN:

4. The user must then type his account number, password, name and project code (if he has one) in the following format:

   PLEASE LOG IN: number password;name;project code (RET)

   Only persons who know the account number, password, and name, may log in under that particular combination. The following examples all illustrate acceptable practice.

   PLEASE LOG IN: A1PASS;JONES;REPUB (RET)

   PLEASE LOG IN: B4WORD;BROWN;DEMO (RET)

   PLEASE LOG IN: C6PW;SMITH; (RET)

   The optional 1-12 character project code is provided for installations that have several programmers using the same account number. The project code is not checked for validity.

   If the user does not correctly type his account number, password, and name within a minute and a half, a message is transmitted instructing him to call the computer center for assistance. The computer will then disconnect the user, and the dial and log-in procedure will have to be repeated.

5. If the account number, password (nonprinting), and name are accepted by the computer it will print READY, the date, and the time on one line, and prints a dash on the next line.

   READY date, time

   - (dash)

   The dash indicates that the Executive is ready to accept a command.

6. Following the dash the user types FORTRAN, which produces the following:

   -FORTRAN

   + (plus sign)

   The FORTRAN command calls CF (the command may be abbreviated to the first three letters, FOR).

   CF responds with a + sign on the next lower line, indicating that it is awaiting a command.

Escape

The ESCAPE key can be used at any time to abort the current operation. Striking the ESCAPE key before terminating a CF command aborts the command. The system then types a plus (+).

During compilation, escape is treated in a slightly different manner. The first escape received during compilation merely causes the bell to ring and has no other effect on the system. If a second escape is received during the same call to the COMPILE command, the program is returned to the state which existed prior to the COMPILE command. The system then types a plus (+).

Exit and Continue

Depressing the (ESC) key several times in succession returns control to the Executive, which responds with a dash (-). If the user wants to return to CF without losing his program, and if he has not subsequently called another subsystem (e.g., BASIC, QED, CAL), he may type CONTINUE. The computer will type FORTRAN and return to it without any initialization. Meanwhile, nothing in core has been destroyed.

Log-Out

When the user wishes to be disconnected from the computer, he depresses (ESC) several times in succession to return to the Executive (which types a dash (-)) and then types:

LOGOUT (RET)

or

EXIT (RET)

The computer will respond with the amount of hook-up (line) time charged to the user's account since the previous log-in procedure was completed. The EXIT command does not update the user's file directory.

2. CONVERSATIONAL FORTRAN SYNTAX

An SDS 940 Conversational FORTRAN program is an ordered set of statements that describes a procedure to be followed by the computer and data to be processed by the program. Statements belong to one of two general classes:

- executable statements that perform computation, input/output operations, and program flow control.
- nonexecutable statements that provide information to the processor about storage assignments, data types, and program form and also provide information to the program during execution about input/output formats and data initialization.

Statements are usually entered on-line at a teletype console in a manner to be described in detail in the following chapter. The use and syntax of the various statements are explained in succeeding chapters.

Several conventions to be followed in writing programs differ from those used in card-oriented, batch processing systems and provide the flexibility needed in a time-sharing system.

Character Set

The following characters may be used to form source statements:

Alphabetic: A through Z

Numeric: 0 through 9

Special Characters: + - * ↑ / ( ) @ $ % & ? [] ' " . , : = < > and blank

The following control characters have special significance in CF:

(RET) Terminates CF commands and CF statements.

(LF) Allows statement to be continued to next line.

(ESC) Aborts the current operation. (See Chapter 1.)

; Terminates CF statements. May be used in place of (RET).

or A^C Deletes the last character typed and may be used repeatedly to delete a corresponding number of characters. Note that # will not delete terminating semicolons.

Q^C or ← Deletes the entire statement currently being entered.

W^C Deletes the last word that was typed. May be used repeatedly to delete a corresponding number of words.

Hollerith input and Hollerith format fields may contain any printing teletype character except the control characters discussed above.

Error Correction

Editing can occur at three distinct levels:

1. A symbolic line can be edited while it is being input by using the # ← A^C Q^C W^C characters.
2. A symbolic line can be edited by referencing it by sequence number and using the EDIT command (see Chapter 3).
3. Data which is input at execution time can be edited by using the A^C W^C and Q^C characters.

Parentheses

Parentheses are used in the normal fashion for indicating the hierarchy of arithmetic operations, subscript notation, and FORMAT statement delimiters. Brackets must be used for all function and subroutine calls. Either parentheses or brackets may be used in I/O statements.

Statements

Statements may be entered at the teletype in a free-form format. It is unnecessary to follow the usual convention of beginning statements in column 7 of a line. Statements may begin anywhere on the line, including column one. A statement is terminated by either a (RET) or a semicolon. If a statement is terminated by a semicolon, the next statement may be typed immediately on the same line. A statement may then be continued from one line to the next by depressing (LF). Except for certain alphanumeric strings, blanks in a statement are ignored and may be used to aid readability.

END Statement

A FORTRAN program must end with a statement consisting of the characters END. This statement indicates to the compiler that there are no more statements in the program; it has no effect upon execution. Since it is nonexecutable, the END statement should not be referenced by another statement.

Comments

If the first nonblank characters of a COMMENT statement are C:, the following statement up to a semicolon or (RET) is treated as a comment. Comments may appear anywhere in a program; they have no effect on execution.

Comments may be continued from line to line by depressing (LF).

FORTRAN Statement Numbers

A statement may begin with a statement number consisting of any number of decimal digits. These numbers permit cross-reference between statements in the program. Leading zeros and blanks are ignored.

The following examples are equivalent:

```
  22
0022
 2 2
```

Statement numbers are used for identification of statements and must therefore be unique. No execution sequence is implied by the magnitudes of the statement numbers. Nonreferenced statements need not be numbered.

FORTRAN statement numbers should never be confused with sequence numbers.

Sequence Numbers

When a source program is entered from the teletype or from a previously prepared file, CF assigns each statement a positive number in the range .001 through 999.999.

Once a program has been entered, these statement numbers may be used for text manipulation. For example, a user may modify or delete a statement by specifying its line number and taking the appropriate console action.

A single number refers to one particular program statement. Two numbers separated by a colon indicate a range of statements. The range 22:30 specifies the statements from 22 through 30.

The assignment of CF sequence numbers is discussed in Chapter 3 in the COMPILE command section.

3. PROGRAM COMPILATION AND EXECUTION

All communication from the programmer to the computer regarding text entry and compilation, program file manipulation, and program execution is conducted via the set of CF system commands described in this chapter.

Programs are normally entered on-line at the teletype console. After logging in, the user calls Conversational FORTRAN with the Executive command

-FORTRAN ⏎

Upon receipt of this command the system executive activates CF and prints a plus sign (+) to indicate readiness to receive commands. The Executive command FORTRAN may be abbreviated to FOR.

General Description

The manner in which programs are entered and compiled differs greatly from procedures used in batch processing environments. In the latter, the programmer typically prepares a complete program on a card or tape file, compiles this program file to obtain an object program, and then executes the object program. If modifications are necessary, the entire program must be recompiled.

In CF, the user begins by issuing a COMPILE command, followed by the source language statements (one or more) to be compiled. The FORTRAN statements may be entered one by one from the teletype keyboard or from a previously prepared file. The statements entered need not comprise a complete program. Once the user has entered his initial group of statements, he may add to the program, modify one or more statements, or delete code without recompiling the entire program. To add statements, he issues another COMPILE command and uses assigned statements numbers to indicate where the additional text is to be inserted. To modify existing statements, he merely recompiles the statements in question or edits them by using the EDIT command. Other CF commands perform total or partial deletion of program text.

The output from the compilation phase is a threaded list of "elements", each of which contains the encoded representation of a source language statement and certain directive information for use in structuring statements into the program.

Execution and program listing are controlled by appropriate CF commands. Special debugging features allow the programmer to interrupt execution, insert temporary, "one-time only" statements, and proceed from any point in the program. At the end of a session at the teletype the user may save his entire program on file. This file may then be read in at the next session and the user may resume where he left off.

CF Commands

All CF commands contain a one-word command identifier. Identifiers may be abbreviated to one, two, or three characters, depending on how many characters are required to distinguish a particular command from the others in the set.

Many commands may also include "statement designators" consisting of CF assigned sequence numbers that describe the part of the program affected by the command.

All CF commands are terminated by a Carriage Return. (⏎).

Statement Designators

Statement designators in CF commands consist of either a single CF sequence number or two CF numbers separated by a colon. They are used by the CF commands to indicate what part of the program is to be affected.

A single number refers to one particular program statement. For example:

25
39.60

Two numbers separated by a colon indicate a range of statements. For example:

22:30

This means: "all the statements from numbers 22 through 30".

The statement designator is separated from a CF command identifier by a comma or space. In the examples in this chapter, statement designator is abbreviated as "sd".

Following is a functional description of the available Conversational FORTRAN commands:

Command	Function
COMPILE	To compile one or more statements
LIST	To obtain a symbolic listing of a compiled program
DELETE	To delete one or more statements
KILL	To delete an entire program
EXECUTE	To execute a previously compiled program
WHY	To determine the cause of an execution error
BREAKPOINT	To temporarily halt execution of a program in order to obtain debugging information
PERFORM	To insert a statement into a program during execution. The statement will not be permanently compiled into the program.
PROCEED	To resume execution of a program after a breakpoint
NEXT	To step through a program
CLEAR	To clear one or more breakpoints
PRINT	To print any symbolic CF program in a user's directory
SAVE	To save a symbolic program
EDIT	To edit a statement

COMPILE

Form	Example
COMPILE, sd	COMPILE, 1:100
COMPILE, sd, file name	COMPILE, 100:500, /PAYROLL/

This command causes control to be passed to the Conversational FORTRAN compiler when the confirming (RET) is read. The first form above is used when the program is entered manually from the teletype or from paper tape. The second form is used when a program has been prepared in QED and output to a disc file, in which case the file as a whole is input to the compiler.

The compiler reads characters from the teletype (keyboard or tape), up to and including an exclamation point (!). Control then returns to the CF command mode. The exclamation point is optional on disc files. During compilation, program statements are automatically numbered within the range specified by the statement designator.

Consider the following example:

```
+COMPILE, 1:20 (RET)
1 ACCEPT [A] (RET)
B=SQRT [A] (RET)
DISPLAY [A, B] (RET)
END!
```

In this example, the source statements following the first ⏎ are compiled and numbered in equal steps ranging from one to a maximum of 20. (In general, the size of the steps assigned depends on the total number of statements compiled.) The source statements could have been delimited by semicolons rather than ⏎, as discussed in Chapter 2. The resulting listing of compiled statements would be:

```
+ LIST
  1.      1 ACCEPT [A]
  7.      B=SQRT [A]
 13.      DISPLAY [A,B]
 19.      END
```

The END statement must appear in the first group of statements compiled; subsequent insertions to the text should then be positioned prior to the END statement.

Note that if only the lower limit of the statement range is given and more than one statement is to be compiled, the number given may not be the first number actually assigned by CF. If both limits of the range are given, the lower limit will always be used as the first statement number, as in the above example.

The COMPILE statement is used to insert additional text and to modify previously compiled statements. For instance, to insert a statement after statement 13 in the previous example, one could issue the command:

```
COMPILE, 14 ⏎
GO TO 1!
```

To modify statement number 7 by replacing variable B by C, the user could type

```
COMPILE, 7 ⏎
C=SQRT [A] !
```

The original statement 7 would be replaced by the new version.

Compiler Diagnostics

Whenever a syntax error is detected in a FORTRAN statement being entered from the teletype, a warning bell rings and the statement in error is immediately printed out. An arrow (↑) is printed beneath the statement at the point beyond which compilation could not proceed. The user must take corrective action either by retyping the statement correctly or by skipping the statement and proceeding to the next one.

Example:

```
COMPILE, 1:500 ⏎
A=3.0 ⏎
DISPLAY A ⏎
DISPLAY A; ⏎
DISPLAY [A] ⏎
      ↑
.
.
END!
```

In this example an error occurred when the programmer omitted the brackets or parentheses required by the DISPLAY statement. The computer immediately printed the statement below it and an arrow to indicate the point where the statement failed to conform to an acceptable format. The user took corrective action by retyping the statement correctly. Since all statements with errors are discarded by the compiler, only the corrected version remains.

If the input to the compiler consists of a previously prepared disc file or paper tape, the computer will print each statement in which there is an error and then discard it, but it will not pause to await corrective action. When compilation is completed, the user may insert corrected statements with further COMPILE commands. Alternatively, he may re-read the file into QED to make his corrections, then recompile the entire file.

Syntax errors only are detected by the compiler; other types of errors are diagnosed during execution.

LIST

Form	Example
LIST	LIST
LIST, sd	LIST, 5:9
LIST, sd, file name	LIST, 1:100, /FILEA/

The first two forms of the LIST command provide a listing of program statements at the teletype; the first lists all existing statements, while the second lists only those statements specified by the statement designator. Statements are listed one per line. This command is typically used immediately after compilation to ascertain the numbers assigned to program statements.

The third form of the statement may be used to place the listing on a user file.

DELETE

Form	Example
DELETE, sd	DELETE, 3 DELETE, 40:60

The DELETE command is used to delete those program statements indicated by the statement designator. The second example above will delete statements 40 through 60.

KILL

Form	Example
KILL	KILL

The KILL command deletes all existing program statements.

EXECUTE

Form	Example
EXECUTE	EXECUTE

This command causes control to transfer to the execution mode. Execution begins with the first statement of the main program.

Execution Diagnostics

If an error occurs during execution, execution is terminated and the system prints the statement that caused the error. Control is then returned to the command mode. If the cause of the error is not apparent, the user may issue the WHY command to request specific information.

WHY

Form	Example
WHY	WHY

The WHY command is used after an execution error to request an explanation of the error. A diagnostic message will be printed at the teletype. Corrective action may then be taken and execution restarted.

BREAKPOINT

Form	Example
BREAKPOINT, sd	BREAKPOINT, 103 BREAKPOINT, 200:410

The BREAKPOINT command allows a user to temporarily halt execution at specified points in a program and return to the command mode. Used in conjunction with the PERFORM and PROCEED commands, it provides a powerful aid to debugging.

If the statement designator consists of a single CF number, a breakpoint will be set at the corresponding program statement. If the statement designator specifies a range, a breakpoint will be set at each statement within the range. When a breakpoint is reached, execution halts prior to execution of the breakpoint statement. The statement is printed out and control transferred to the command mode so that the user can either take debugging action or proceed.

PERFORM

Form	Example
PERFORM, fs	PERFORM, A=100 PERFORM, ACCEPT (A(I),I=1,10)

where fs is a FORTRAN statement.

The PERFORM command allows a user to execute a FORTRAN statement without having it permanently compiled into the program. The command may only be given during a halt in execution caused by a breakpoint, Escape, or normal program termination. Use of the PERFORM command requires that the EXECUTE command was previously given (i.e., it requires the existence of a program).

The statements used in conjunction with PERFORM may not contain a call to a function. Also, statements are restricted to replacement and input/output statements. The statement may be terminated by a (RET), semicolon, or exclamation point.

The PERFORM statement is used in conjunction with the BREAKPOINT command for debugging pusposes. For example, the user can use a DISPLAY statement (Chapter 8) at various breakpoints to examine the changing values of variables being manipulated by the program. If he discovers an error in the computation of a variable, he can perform a replacement statement which sets the variable to the expected value, and he can then proceed to check out the rest of the program before following up on the previous error. Since FORTRAN statements inserted with the PERFORM command do not become a permanent part of the program, the user is not required to delete them.

A statement executed with a PERFORM affects only the storage in effect for the current program or subprogram. Thus, to change the value of a variable in a subprogram, execution must have been halted while in the subprogram.

PROCEED

Form	Example
PROCEED	PROCEED

This command is used at a breakpoint to resume execution.

NEXT

Form	Example
NEXT	NEXT

The NEXT command is used at a breakpoint to resume execution and to set a new breakpoint at the next statement in the program. In other words, the previous breakpoint statement will be executed and another halt will occur.

CLEAR

Form	Example
CLEAR, sd	CLEAR, 100 CLEAR, 250:360.5

The CLEAR command clears the indicated statements of breakpoints.

PRINT

Form	Example
PRINT, file name	PRINT, /PAYROLL/

The PRINT command allows the user to print on the teletype any symbolic file in his file directory.

SAVE

Form	Example
SAVE, sd, file name	SAVE, 200:450, /FILE 1/ SAVE, 1:500

The SAVE command allows the user to save all or a portion of the symbolic code of his program on a specified file for future compilation or manipulation with QED. If the user wishes to output the program to paper tape, he may omit the file name and turn the teletype tape-punch unit on immediately after the confirming (RET) has been typed. The statement designator is required when a disc file is named, but it is optional for the teletype.

When the SAVE command specifies a disc file, the message OLD FILE or NEW FILE is printed. This feature protects the user from inadvertently writing on an old file which he wants to preserve. If the message OLD FILE is printed, the user may choose to abort the command with the (ESC) key and to assign a different file. A (RET) will confirm the file assignment and complete the operation.

When the user wishes to compile the saved program statements, he names the file in a COMPILE command.[t]

EDIT

Form	Example
EDIT, sn	EDIT, 5

The EDIT command prints the line addressed (sn) and allows it to be edited using the following commands.

A^C (Prints ↑)	Deletes preceding character.
C^C	Copies the next character from the line being edited.

[t]When a tape is generated by a SAVE, the readiness sign (+) which is output at completion of the operation may be punched on the tape. Then, when the tape is input to the compiler, two plus signs (the punched one and the compiled one) will print at the end of compilation. This extra plus sign may be ignored.

D^C	Finishes a line edit by copying and printing the remainder of the line.
E^C	Allows characters to be inserted into a line. The first E^C typed will print <. The user types his insertions and denotes the end of the insertion by typing another E^C. which prints >.
F^C	Finishes a line edit by copying but not printing the remainder of the line.
H^C	Copies a line up to but not including the (RET).
S^C (Prints %)	Deletes the next character from the line being edited.
$Z^C x$	Copies all characters through x from the line being edited to the new line.
M^C	Similar to the (RET) key. It ends the edit mode.

Sample Program

The following exhibit illustrates a hypothetical session at the teletype. Text printed by the computer is underlined; comments appear in parentheses. Spacing has been modified for the sake of clarity. Note that semicolons might have been used as statement delimiters, in which case the statements would be contiguous.

```
PLEASE LOG IN   111DEMO;SMITH (RET)
READY 11/29  10:30
-FORT (RET)
+COMPILE, 1:500 (RET)
C:SORT
DIMENSION A(50)
ACCEPT [N,(A(I),I = 1,N)]                (Programmer uses simplified I/O, Chapter 8)
DO10J#I = 1,N-1                          (# deletes previous character)
IF(A)← DO10J = I+1,N                     (← deletes current statement)
IF(A(I)-A(J))10, 10,5
5 TEMP = A(J)
A(J) - A(I)
A(J) - A(I);                             (Compiler detects syntax error)
A(J) = A(I)↑                             (User re-enters statement correctly)
A(I) = TEMP
10 CONTINUE
END!                                     (! terminates compiler input)
+ LIST (RET)
 10.  C:SORT
 50.  DIMENSION A(50) (RET)
 90.  ACCEPT  N, (A(I), I = 1,N) (RET)
130.  DO 10 I = 1,N-1 (RET)
170.  DO 10 J = I+1,N
210.  IF (A(I) - A(J)) 10, 10, 5 (RET)
250.  5 TEMP = A(J) (RET)
290.  A(J) = A(I) (RET)
330.  A(I) = TEMP (RET)
370.  10 CONTINUE
410.  END (RET)
```

```
+ COM, 390 ⏎
DISPLAY [(A(I), I = 1, N)]                    (User inserts additional text)
+ EXECUTE ⏎
5,5,3,1,2,4, ⏎                                (User inputs values to the ACCEPT statement)
1.   2.   3.   4.   5. ⏎                      (Computer prints results implicitly formatted, Chapter 8)
+ ESC ESC                                     (User returns to Executive by striking the ESC key twice)
- LOGOUT
```

4. DATA

A constant is a quantity whose value is explicitly stated. For example, the integer 5 is represented as "5"; the number π, to three decimal places, as "3.142". A variable is a numerical quantity which is referenced by a symbolic name rather than by its explicit appearance in a program statement. During execution of the program, a variable may take on many values rather than being restricted to one.

All data processed by a CF program can be classed into five groups: integer, real (single precision)[†], complex, logical, and Hollerith.

Limits on Values of Quantities

Both integer and real (or "floating point") data can be assigned any value in the approximate range 10^{-77} to 10^{76}. Both kinds of data are stored in floating point form, using two words (48 bits): a 38-bit mantissa, a 9-bit exponent, and a sign bit. (The exponent constitutes the last 9 bits of the second word.) Both integer and real data have an associated precision of 11+ significant digits. That is, numbers with 11 significant digits will be accurate, while numbers with 12 significant digits will be accurate for values up to $2^{38}-1$, i.e., 274,877,906,943. Numbers greater than this will lose accuracy in the least significant position.

Complex data are approximations of complex numbers, taking the form of an ordered pair of real data. The first of the two real data approximates the real part, while the second approximates the imaginary part of the complex number. The values each part may be assigned are identical to the set of values for floating point data.

Logical data can acquire only the values .TRUE. or .FALSE.

Hollerith data represent character string values. The set of values which each character in the string may assume are given in Chapter 2, in the discussion on the FORTRAN character set. A Hollerith datum is stored in two computer words in ASCII Code. Characters are stored left justified with trailing blanks.

Constants

Constants are data that do not vary in value. They are referenced by having their values named. Constants may be any type of data. For constants with positive values the plus character (+) need not be present.

Integer Constants

Integer constants are represented by strings of decimal digits optionally preceded by a sign character.

Examples:

392	+997263
-13	1234567

Octal Constants

Octal constants are represented by a string of octal digits preceded by the character O, and a character count. The constant may be optionally preceded by a sign character.

Examples:

3O372	6O+77726
3O-13	7O1234567

[†]Although in the mathematical sense real numbers include integers, in FORTRAN the term "real" is used to describe only numbers that may have a fractional part, as opposed to integers, which may be whole numbers only.

Real Constants

Real constants are represented by a string of digits containing a decimal point at either end of the string or between 2 digits. Plus sign characters are optional.

A real constant can be given a scale factor by appending an E, followed by an integer constant specifying the power of 10 by which the floating point constant is to be multiplied. Thus, +0.234E+03 has the meaning $0.234 \times 10^3 = 0.234 \times 1000 = 234.0$. The magnitude of the resulting number must be within the limits for real data. This method provides a convenient way to express large numbers.

The scale factor constant may be preceded by a plus or minus sign; if omitted, the sign is considered positive.

The following forms are all equivalent:

+0.567E+05	.567E5	5.67E+4	56700.0
567000E-01	.5670E05	56700.E0	56700E-00

Since any real constant may be represented in a variety of ways, the user can choose the form most convenient for his purpose.

Examples:

-394.6238763	5.0	-7.6E+5
-3946.238763E-5	0.5	.39653
+3847562910.	1E1	-1E-1

Complex Constants

Complex constants are expressed as an ordered pair of constants in the format:

(c_1, c_2)

where c_1 and c_2 may be integer or real constants. Parentheses and comma are required. Integer constants are converted to real-constant approximations of their values. The complex constant (c_1, c_2) is interpreted as meaning c_1+c_2i. The following complex constants have values as indicated:

(1.34,52.01)	=	1.34+52.01i
(98344,.34452E+02)	=	98344.0+34.452i
(-1.,-1000)	=	-1.0 - 1000.0i
(2.3,0)	=	2.3+0i
(0,4.5)	=	0+4.5i
(2.7E1,0.8)	=	27.0+0.8i

Neither part of a complex constant may exceed the value limits established for real data.

Logical Constants

Logical constants may assume either one of two forms:

.TRUE. .FALSE.

where these forms have the values "true" and "false".

Hollerith Constants

Hollerith constants are represented in the form

nHs

where

n is an unsigned integer constant of the set (0, 1, 2, 3, 4, 5, 6).

s is a string of characters whose length exactly corresponds to the value of n.

Each character in a Hollerith constant may be one of the set of characters discussed in Chapter 2.

Hollerith constants may be assigned to real variables only. Since Hollerith constants are stored 3 characters per 24-bit word, and real data use 2 words each (48 bits), a maximum of 6 characters is allowed in the Hollerith constant. If less than 6 are used, the characters are stored left justified with trailing blanks.

If a variable to which a Hollerith constant is assigned is to be output, an A format specification should be used (see Chapter 8). It is not possible to do arithmetic or logical comparisons with Hollerith constants (see Chapter 5).

Examples:

4HFOUR	3HYOU	2H$$	1H+
6HOH BOY	3HOH?	2HX=	1HH

Identifiers

Identifiers are symbolic names consisting of strings of letters and decimal digits. An identifier must begin with a letter. Identifiers are used to name variables as well as subprograms and subprogram arguments. CF identifiers may be of any length. Embedded blanks are ignored. Subscripted variables should not have identifiers that correspond to FORTRAN statement types such as ACCEPT, DISPLAY, OPEN, CLOSE, READ, WRITE and FORMAT.

Examples:

X A345Q	J3	QUANTITY	FIRST ONE
ELEVATION	I	L987564	DIFFERENTIAL

Variables

Variables are data whose values may vary during program execution and are referenced with an identifier. Variables may be any of the data types.

If a variable has not been explicitly assigned to a particular data type (Chapter 9), the following conventions are assumed:

- Variables whose identifiers begin with the letters I, J, K, L, M, or N are integer data.
- Variables whose identifiers begin with any other letter are real data.

Consequently, complex and logical variables must be explicitly declared as such. The values assigned to variables may not exceed the limits established for the applicable data types.

Scalar Variables

A scalar variable is a single datum entity; it is accessed via an identifier of the appropriate type.

Examples:

I1	EXPONENT	NAME	XXX8

Arrays and Array Variables

An array is an ordered set of data that may be referenced and altered in a program. The set as a whole is named by an array identifier according to the rules discussed above for variables. The elements of the array, called array

variables, are referenced by the array identifier followed by an expression, called a subscript, which describes the element's position within the array.

For example, A(4) refers to the fourth element in a set of elements called A. This would be a one-dimensional array, or vector. A two-dimensional array is thought of as arranged into columns and rows. An element in a two-dimensional array is referenced as A(I, J) where I refers to a row element and J refers to a column element. For example, consider the set of numbers:

111	222	333
444	555	666
777	888	999

If the entire set is called B, then the element 666 is referenced as B(2, 3). B is called a "3 by 3" array or matrix.

SUBSCRIPTS

Subscripts may assume the following form:

$(s_1, s_2, s_3, \ldots, s_n)$

where

s_i are any expressions of integer or real mode.

n is the value of the number of dimensions associated with the array.

The parentheses and commas are required. Real expressions used as subscripts are truncated to integer values.

Examples:

Array Name	Array Variable
MATRIX	MATRIX(3, 9)
CUBE	CUBE(J*4, P, 3.6)
A	A(Q/I+U-M)
J	J(7.5E+2)

Nested subscripts are permissible; that is, subscripts themselves may be subscripted. There is no limit on the level of nesting.

Examples:

```
ALPHA (I(J))
MATRIX (I(J(K)))
```

5. EXPRESSIONS

Expressions are strings of operands separated by operators. Operands may be constants, variables, or function references. An expression may contain subexpressions, that is, expressions enclosed in parentheses. Operators may be unary, i.e., they may operate on a single operand, or they may be binary, operating on pairs of operands.

Expressions may be classified as arithmetic, logical, or relational. All expressions yield a single, unique value when evaluated.

Arithmetic Expressions

An arithmetic expression is a sequence of constant, variable, or function references connected by arithmetic operators.

The arithmetic operators and their connotations are as follows:

Operator	Operation
** or ↑	Exponentiation
/	Division
*	Multiplication
-	Subtraction (binary) or negative sign (unary)
+	Addition (binary) or positive sign (unary)

Expressions may consist of a single basic element, i.e., a constant, variable, or function.

Examples:

```
3.1415
X(N)
SQRT [ALPHA]
```

Basic elements may be combined through use of the arithmetic operation symbols to form compound expressions.

Examples:

```
A+B
PI*RADIUS**2
SQRT [THETA* THETA]
```

Compound expressions may be enclosed in parentheses to form subexpressions.

Examples:

```
(A+B)/(C+D)
-((M-N)*(Z-Q(J)))
```

Precedence

The expression A+B/X could be evaluated as:

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

or as

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

To avoid the possibility of such ambiguities, various rules governing precedence of evaluation have been established. The evaluation hierarchy is as follows:

1. The innermost subexpression, followed by the next innermost subexpression, until all expressions have been evaluated.

2. The arithmetic operations have the following order of precedence:

Operation	Operator	Order
Exponentiation	** or ↑	1 (highest)
Multiplication and Division	* /	2
Addition and Subtraction	+ -	3

Several additional conventions are necessary:

1. At any one level of evaluation, operations of the same order of precedence are evaluated from left to right. Consequently, I/J/K/L is equivalent to ((I/J)/K)/L.

2. As in algebraic notation, parentheses are used to define evaluation sequences explicitly. Thus, $\frac{A+B}{C}$ is written as (A+B)/C.

3. The sequence "operator operator" is permissible if the expression can be evaluated when the second operator is interpreted as unary. Thus A*-B is interpreted as A*(-B).

As an illustration of the above rules of precedence, the expression

A*(B+C*(D-E/(F+G)-H)+P(3))

is evaluated in the following sequence, where the r_i are the various levels of evaluation:

$r_1 = F+G$

$r_2 = E/r_1$

$r_3 = D-r_2-H$

$r_4 = C*r_3$

$r_5 = B+r_4+P(3)$

$r_6 = A*r_5$

Mixed Expressions

Arithmetic expressions may contain references to data or functions of the integer, real, or complex types. References to data, subexpressions, or functions of the logical type are excluded from arithmetic expressions, except when they appear as function arguments. When arithmetic expressions contain references of more than one type, they are called mixed expressions. Mixed expressions are evaluated in the mode of the highest order of reference:

Complex	1 (highest precedence)
Real	2
Integer	3

The following rules also govern evaluation of mixed expressions:

1. Expressions appearing as subscripts or function arguments are evaluated separately in their own modes and have no effect on the mode of the larger expression in which they are contained.

2. Exponents may be integer or real.

3. Values of expressions, subexpressions, and terms are restricted to those limits associated with the mode of the expression.

4. Values of real and integer modes which appear in complex-mode expressions are assumed to have imaginary parts of zero value.

Relational Expressions

Relational expressions consist of arithmetic expressions separated by relational operators which cause the expressions to be compared. Evaluation results in one of the two logical values, "true" or "false".

In general, the form of a relational expression may be written:

$$e_1 r e_2$$

where

e_i are arithmetic expressions.

r is a relational operator.

The following table shows the relational operators and their meanings:

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

Relational expressions have the value "true" if and only if all comparisons in the expression are true. For example:

```
1 .LT. 6          true
0 .GT. 8          false
0 .LT. (2.**N)    always true
0 .LT. -(2**N)    always false
```

Only real and integer comparisons are allowed. If one expression is integer and the other real, the two expressions are first evaluated, each in its own mode; then the value of the integer expression is converted to real mode and a real comparison is made.

It is not possible to nest relational expressions, as in the case

```
(L(X. GT. 0.2345E6))
```

where (X .GT. 0.2345E6) is a relational subexpression rather than an arithmetic expression, as the definition of relational expression requires.

Logical Expressions

Logical expressions are expressions of the form:

$$e_1 c_1 e_2 c_2 e_3 c_3 \cdots e_n$$

where

e_i are logical elements.

c_i are the logical operators.

Evaluations of logical expressions result in one of the two values, "true" or "false".

Logical elements are defined as one of the following entities:

1. A logical variable or function reference.
2. A logical constant.
3. A relational expression.
4. Any of the above enclosed in parentheses.
5. A logical expression enclosed in parentheses.
6. Any of the above preceded by the unary logical operator . NOT.

There are four logical operators:

Operator	Type
. NOT.	unary
. AND.	binary
. OR.	binary
. EOR. (exclusive OR)	binary

Logical expressions are evaluated as follows (the letter "e" denoting a logical element):

. NOT. e	true only when e is false
e_1 . AND. e_2	true only when both e_1 and e_2 are true.
e_1 . OR. e_2	true when either or both e_1 and e_2 are true.
e_1 . EOR. e_2	true when either but not both e_1 and e_2 are true.

These rules are illustrated in the following table:

		Logical Operator			
		. NOT. e	. AND.	. OR.	. EOR.
e True		False			
e False		True			
e_1 False	e_2 False		False	False	False
e_1 True	e_2 False		False	False	False
e_1 False	e_2 False		False	True	True
e_1 True	e_2 True		True	True	True

Consider the following examples of logical expressions:

```
A. AND. B
```

A and B must be logical variables

```
A. GT. B. AND. I. EQ. 3
```

The value of the expression depends on the values of the relational expressions

```
A. LT. B. OR. C. GT. 5. AND. T
```

Evaluation Hierarchy

In a manner similar to that discussed for arithmetic espressions, parentheses are used to define explicitly evaluation sequences. Consequently,

```
A . AND. B . OR. Q(3) . NE. X
```

does not have the same meaning as:

```
A . AND. (B . OR. Q(3) . NE. X)
```

where (B. OR. Q3 . NE. X) may be called a logical subexpression.

The evaluation hierarchy for logical expressions is:

1. Arithmetic expressions.
2. Relational expressions. (The relational operators are all of equal precedence.)
3. The innermost logical subexpression, followed by the next innermost logical subexpression, etc.
4. The logical operations are in the following precedence:

Operator	Order
. NOT.	1 (highest)
. AND.	2
. OR.	3
. EOR.	4

Note: Two contiguous logical operators are permissible only when the second operator is . NOT. For example:

e_1 . AND. . OR. e_2 illegal

e_1 . AND. . NOT. e_2 legal

6. ASSIGNMENT STATEMENTS

The CF language is comprised of five types of statements:

Assignment Statements

Control Statements

Input/Output Statements

Declaration Statements

Subprogram Statements

Each type of statement performs a specific function. Assignment statements are discussed in this chapter; subsequent chapters are devoted to discussion of the other types of statements.

There are two types of assignment statements: replacement statements and statement number assignment statements.

Replacement Statement

A replacement statements specifies (1) an expression to be evaluated and (2) the variable, called the statement variable, to which the expression value is to be assigned.

Form	Example
v=e	A=B Q(I)=Z**2+N*(L-J) L=B. OR. . NOT. C. AND. R. NE. 23.93

where

v is a variable name

e is an expression

Note that the equal sign denotes replacement rather than equality. Thus Y=Y+1 is a valid statement meaning "add one to the value of Y and assign the resulting value to Y".

When the mode of the expression e is not the same as the variable type for v, the variable is assigned values as indicated in the following table.

Rules for Assignments of e to v		
If v Type is	and e Type is	Assignment Rule is
Integer	Integer	Assign
Integer	Real	Fix and Assign
Integer	Complex	Prohibited
Integer	Logical	Prohibited
Real	Integer	Float and Assign
Real	Real	Assign
Real	Complex	Prohibited
Real	Logical	Prohibited
Complex	Integer	Prohibited
Complex	Real	Prohibited
Complex	Complex	Assign
Complex	Logical	Prohibited
Logical	Integer	Prohibited
Logical	Real	Prohibited
Logical	Complex	Prohibited
Logical	Logical	Assign

where

assign means transmit the resulting value, without change, to the variable.

fix means truncate any fractional part of the result and transform that value to the form of an integer datum.

float means transform the value to the form of a real datum.

Statement Number

Statement number assignment is used to assign to a variable the location of a statement.

Form	Example
ASSIGN k TO v	ASSIGN 153 to LABEL ASSIGN 603 to FLAG1

where

k is a statement number

v is a scalar variable reference, of integer or real mode.

Once a statement number has been assigned to a variable, the variable must not be referenced except as a statement number. Thus, in the following sequence the value assigned to C will be indeterminate because the value of A is undefined:

```
ASSIGN 101 to A
C=A/B
```

Note that the statement M=5 cannot be substituted for ASSIGN 5 TO M, or vice versa.

The use of such assignment is discussed in Chapter 7, in the section on Assigned GO TO Statements.

7. CONTROL STATEMENTS

Each statement in a CF program is executed in the order of its appearance in the source program, unless this sequence is interrupted or modified by a control statement. If program control is to be transferred to a particular statement, that statement must be identified by a number (see Chapter 2).

In general, control statements may be used to:

- provide unconditional transfer of control to other statements in the program.
- test variables and provide conditional transfer of control to other statements in the program.
- execute a particular sequence of statements repeatedly a specified number of times.
- provide branching to and return from subprograms.

GO TO Statements

There are three forms of GO TO statements: unconditional, assigned, and computed.

Unconditional GO TO

The unconditional GO TO provides a means to unconditionally transfer control to another statement in the program.

Form	Example
GO TO k	GO TO 5 GO TO 800

where

k is a statement number of an executable statement

The result of executing this statement is that the next statement executed is the statement whose number is k.

Assigned GO TO

The Assigned GO TO statement transfers control to a statement referenced by a variable previously defined in an ASSIGN statement.

Form	Example
GO TO v GOTOv, $(k_1,k_2,k_3,\ldots k_n)$	GO TO G GO TO G, (117,56,101)

where

v is a variable appearing in a previously executed ASSIGN statement.

k_i are statement number references.

Control is transferred to the statement whose location has been assigned to the variable v.

If the second form is used, each number in the list must be defined in the program or subprogram in which the GO TO statement appears (i.e., must be the number of a program statement). This form serves no purpose other than to provide compatibility with other processors. The comma and parentheses characters must appear as shown.

For example, the statements

```
ASSIGN 5371 TO G
GO TO G
```

will cause transfer of control to the statement labeled 5371. The optional form would be

```
ASSIGN 5371 TO G
GO TO G, (117,56,101,5371)
```

Computed GO TO

The Computed GO TO statement allows transfer of control to one of a group of statements, the particular statement chosen depending on conditions at run time.

Form	Example
GO TO $(k_1,k_2,k_3,\ldots,k_n)$,e	GO TO (98,65,405,3), R GO TO (5,6,7), T**2-1

where

k_i are statement numbers

e is an expression of integer or real mode.

Control is transferred to the statement whose number is k_j, where j is the integer value of the expression e. The value of the expression must be greater than zero and less than or equal to n, that is, $0 < j \leq n$. Real mode expressions are evaluated and then truncated to integer value.

In the first example above, if the expression R has the value 3, control will be transferred to the statement labeled 405. If the expression (T**2-1) in the second statement has the value 1.56, control will be transferred to statement 5.

The comma preceding e is optional

IF Statements

IF statements are conditional transfer statements that allow the programmer to change the logical flow of a program on the basis of a test. There are two types of IF statement: arithmetic and logical.

Arithmetic IF Statements

Form	Example

IF (e) k_1,k_2,k_3	IF(G+B(I)) 76, 4, 3
	IF(X-Y) 100, 250, 3000
	IF (I) 1, 2, 3

where

e is an expression of integer or real mode.

k_i are statement numbers.

If the value of e is less than 0, transfer is to k_1; if the value of e is equal to 0, transfer is to k_2; if the value of e is greater than 0, transfer is to k_3.

A comma may optionally precede k_1.

Examples:

Statement	Expression Value	Transfer To
IF(I)1,2,3	47802	3
IF(C(J, 10)/4),23, 12, 8	-. 098433	23
IF(A+B(I))44, 33, 22	0. 0	33

Logical IF Statements

Form	Example

IF (e) s	IF (E. OR. D) GO TO 3135
	IF (A. LT. B. AND. C. LT. D) K=22
	IF (A. AND. G), IF(C. NE. K), ON=
	. TRUE.

where

e is a logical mode expression.

s is any executable statement.

The statement s is executed if the expression e has the value "true"; otherwise, the next executable statement following the logical IF statement is executed. The statement following the logical IF will be executed in any case after the statement s, unless the statement s causes a transfer to occur, as in the first example above.

Note that the entire statement IF (e) s, is treated as a single statement.

A comma may optionally precede the statement s.

DO Statement

The DO Statement is used to control repetitive execution of a group of statements.

Form	Example
DO k v=e_1,e_2,e_3	DO 10 I=1, 10 DO 12 J=2, 98, 2 DO 15 V=END, START, -. 05

where

k is a statement number.

v is a reference to a scalar variable of integer or real mode.

e_i are expressions of integer or real mode.

An optional comma may be placed between k and v.

The DO statement causes repeated execution of all statements within its range. The range of a DO extends from the first executable statement following the DO statement up to and including statement k.

The scalar variable v is called the index of the DO statement. It is used to identify the repetition currently being performed. The value of e_1 represents the initial value of the index, the value of e_2 the limiting or terminal value of the index, and e_3 the incrementing quantity. If e_3 is omitted, the increment is assumed to be one.

The initial execution is always performed, regardless of the values of the limit and increment. After each execution of the range, the increment value is added to the value of the indes, and the result is compared with the limit value. If the value of the index is not greater than the limit, the range is executed again, using the new value of the index. (In case the increment value is negative, another execution will be performed if the new value of the index is not less than the limit value.)

When the index value exceeds (or, if decrementing, is less than) the limit value, control passes to the statement immediately following statement k. Exit may also be effected by a transfer from within the range of the DO statement.

Consider this example:

```
DO 999, I=1,5,2,
```

The meaning is: execute all statements immediately following, up to and including statement number 999, first for I=1, next for I=3, and last for I=5. Then transfer control to the statement following statement number 999. Thus, the loop will be executed a total of 3 times.

The terminal statement of a DO range (k) may be any executable statement. However, the programmer should exercise care if the terminal statement is a transfer; the consequences can be determined by inspection. Incrementing and testing will not take place if k is a transfer statement.

If a transfer is made out of the range of a DO before all iterations have been completed, the value of v will be that value current during the iteration during which the transfer occurred.

The value of the variable v may be modified by any form of assignment statement within the range of the DO. It may also be modified by a subprogram called within the range of the DO.

A transfer into the range of a DO may only occur if there has been a prior transfer out of the range. In fact, the statements executed "outside" the range will then be considered part of the DO range.

A DO loop may include other DO loops, provided that the range of each "inside" or "nested" DO statement is contained completely within the range of an "outside" DO statement. In other words, the ranges of two DO statements may not partially intersect one another. Only total intersection or no intersection is allowed. There is no limit to the level of nesting. The same statement may be used as the terminal statement for any number of DO statements.

If the programmer wishes to avoid terminating a loop with a transfer statement, he may use the CONTINUE statement as a dummy end for the loop.

CONTINUE Statement

This statement is a dummy, a "do nothing" statement used primarily to serve as a target point for transfers, particularly as the last statement in a DO loop. At the end of the range of a DO, the CONTINUE statement means, in effect, "do nothing but proceed to modify and test the index".

For example, in the sequence:

```
DO  5  I=1, MAX
.
.
.
GO TO 5
.
.
.
X=SUM
.
.
.
5 CONTINUE
```

If the GO TO is intended to begin another execution of the DO loop, without performing the statement X=SUM, the CONTINUE statement provides the necessary target address.

PAUSE Statement

The PAUSE statement provides a means of temporarily halting program execution

Form	Example
PAUSE	PAUSE
PAUSE c	PAUSE 777

where c is any string of characters.

The word PAUSE will be displayed at the teletype, as will the string c if it is specified. The programmer can cause execution to continue by typing any character.

STOP Statement

The STOP statement causes termination of the program and return of control to the CF command mode.

Form	Example
STOP	STOP
STOP c	STOP 777

where c is any string of characters.

Subprogram Control

The CALL and RETURN statements, discussed below, provide transfer of control between subprograms and calling programs (see Chapter 10 for a general description of subprograms).

CALL Statement

The CALL statement causes a transfer of control to a subroutine-type subprogram.

Form	Example
CALL p	CALL DUMP
CALL p $[a_1, a_2 \cdots, a_n]$	CALL FACTOR [A+1, BETA]

where

p is the identifier of the subroutine.

a_i are arguments required by the subroutine.

If the subroutine being called does not require an argument list, the first form above is used.

Arguments in a CALL statement may be constants, scalar variable references, array element references, array identifiers, expressions, or subprogram references.

If a subprogram identifier is used as an argument, the identifier is not followed by an argument list, since this argument form is only used to provide the called subroutine with a subprogram reference. In this sense, the subprogram reference is merely a name and, as such, has no value associated with it.

For example, one might use

```
CALL CALC  A, B, SQRT
```

to call

```
SUBROUTINE CALC  X, Y, Z
:
C=Z X
:
END
```

At execution time, the dummy subprogram identifier Z will be replaced by SQRT.

The name of a subroutine has no bearing on the mode of its results.

RETURN Statement

The RETURN statement returns control from an external subprogram to the calling program. Thus, the last statement executed in a subprogram will be a RETURN. It need not be physically the last statement but instead may appear at any point in the subprogram where it is desired to terminate execution. Any number of RETURN statements can be used.

Form	Example
RETURN	RETURN

Within a function subprogram, the RETURN causes a return to the evaluation of the expression in which the function reference appeared. In a subroutine subprogram, RETURN causes transfer to the first executable statement following the CALL statement which passed control to the subroutine.

8. INPUT/OUTPUT STATEMENTS

Input and output statements provide the capability of communicating with devices external to the computer. Input statements enable a program to receive information from external sources for storage in memory, while output statements allow transmission of information from storage to external sources.

In the time-sharing environment, data files are normally created at the teletype (manually and from paper tape), maintained on a large disc, and printed out at the teletype. If the use of other devices such as magnetic tape, card reader, card punch, or printer is required, a special request should be made to the data center.

The number of statements in CF necessary to input and output data has been reduced to four. The ACCEPT and DISPLAY statements are used for format-free I/O to and from the teletype. The READ and WRITE statements are used for conventional formatted I/O to or from any user file. (In addition, the OPEN and CLOSE statements control availability of files during program execution.

Input/Output Lists

All input/output statements include a list which defines what data are to be processed by the statement. Input lists specify variables to which incoming data are to be assigned. Output lists specify expressions whose values are to be transmitted to an external device.

Simple List Items

Simple list items appear in the forms

$$e_1, e_2, e_3, \ldots, e_n$$

and

$$v_1, v_2, v_3, \ldots, v_n$$

where

e_i are expressions of any mode for output lists.

v_i are variable references of any type for input lists.

The comma characters must be present. The items in the list must be given in the same order in which their corresponding values actually exist on the input medium or will exist on the output medium.

On input, values of variables read early in a list may be used in subscript or control expressions for variables occuring later in the list. For example, the list

```
K, A(K+1)
```

may be used to read in a value for K and then to have that value used in the subscript of variable A.

Examples:

Input Lists	Output Lists
A	E
Q(25, L)	I(J(H), N)
RY, Y(U, B), XYZ	753820, T**5, B/3. +Y

Note: ACCEPT, DISPLAY, READ, WRITE, OPEN, and CLOSE statements, discussed below, may use either brackets or parentheses; i.e., all of the following are correct:

```
ACCEPT (A, B, C)
ACCEPT [A, B, C]
ACCEPT (A, B, C]
ACCEPT [A, B, C)
```

Free Format I/O

The ACCEPT and DISPLAY statements are designed to relieve CF users of the burden of providing formats when explicit format control is not required. The device accessed is always the teletype

ACCEPT Statement

The ACCEPT statement is used to read values from the teletype. When this statement is executed, the computer waits for data to be input by the programmer and then assigns these values to variables in a list.

Form	Example
ACCEPT [list]	ACCEPT [A, (B(I), I=1,3), CHECK]
ACCEPT (list)	ACCEPT (X,Y)

The type of the list variable determines the form in which conversion from external to internal form takes place. The rules governing storage are similar to those for assignment statements (see Chapter 6). For example, if the type of the list variable is integer and a real number is input, the number will be truncated to an integer prior to storing.

If the type of the list variable is complex, two numbers will be demanded. The first number read will be assigned to the real part, the second to the imaginary part. If the type of the list variable is logical, the first character read must be a T for true or F for false, and any remaining characters up to the delimiter (see below) will be discarded.

Values input to the ACCEPT must be separated from each other by a space, comma, or (RET). Values will be demanded until the variable list is satisfied.

The input values may be edited by using A^c, U^c, or Q^c. See "Editing of Input Strings" later in this chapter.

Successive delimiters with no values typed in between them will cause zero values to be assigned to the appropriate variables in the list.

In the example ACCEPT [A, (B(I), I=1, 3), CHECK], the user could input the following:

1 (RET)
-3, 10E6, -16.6 (RET)
T (RET)

Assuming that the variable CHECK was data typed previously as LOGICAL in a type statement, these values would be interpreted as 1., -3., 10×10^6, -16.6, and "true", respectively.

DISPLAY Statement

The DISPLAY statement is used to print data at the teletype. The data is transmitted as values of expression in a list.

Form	Example
DISPLAY [list]	DISPLAY [A, B+1]
DISPLAY (list)	DISPLAY ('PRICE=$", X)

The mode of a list expression determines the manner in which its value is converted from internal to external form. In general, values are output to maximum accuracy, with all leading spaces and trailing zeros suppressed. Values are separated by three spaces and are output until the list is satisfied.

A real value is output with or without an exponent depending on the magnitude of the number, and according to the same rules that govern G-type output (discussed under G-conversion, below).

The system automatically checks whether sufficient room is available on the current teletype line (72 positions) for data. When sufficient room does not exist, a (RET) is issued and the next value is output at the beginning of the next line. When the list has been satisfied, a (RET) is automatically issued.

Character strings may be output by enclosing the string within single and double primes. Thus, if X = 100, then

```
DISPLAY ['PRICES=$", X]
```

will produce the line

```
PRICE=$bbb100.
```

where b denotes blank.

Complex values will be output as two real variables separated by commas. For example,

```
1.23,5.65
```

If an expression is logical, the output will be a T for true or an F for false.

Formatted Input/Output

When the programmer wishes to have explicit control over the form in which data are transmitted, formatted input/output statements are used. In general, formatted input/output follows conventional FORTRAN rules.

All data files (other than the teletype, which may also be referenced as a file) must be explicitly opened and closed by the two statements discussed below.

OPEN Statement

The OPEN statement makes a data file available for input or output. A program may have 3 files, in addition to the teletype, opened at the same time. Any file to be opened must have been defined in the user's file directory prior to execution of the OPEN statement[†].

Form	Examples
OPEN [file number, file name, use, type]	[OPEN 2, /IN/, INPUT, BINARY]
OPEN (file number, file name, use, type)	OPEN (3, /FILE1/, OUTPUT)

A file number must be assigned between 0 and 4 inclusive. The numbers 2, 3, and 4 are used for disc files. The numbers 0 and 1 are reserved for teletype input and output, respectively; however, since the teletype is always open, an OPEN statement for the teletype is redundant. The file number may appear as an expression.

The file name is the name of the file as it appears in the user's file directory. Disc file names are enclosed in slashes. (See Executive Manual for details concerning creation of files.)

The use of the file is either INPUT or OUTPUT.

The type is either SYMBOLIC or BINARY. Type is optional; if not given, symbolic is assumed.

CLOSE Statement

As its name implies, the CLOSE statement closes the designated file, i.e., makes it unavailable for input or output until reopened.

[†]An output file may be defined in the user's directory by writing anything on it with the Executive command COPY. Data output to the file during program execution will write over the original contents.

Form	Example
CLOSE [file number] CLOSE (file number)	CLOSE [3] CLOSE (2)

where the definition of a file number is the same as for the OPEN statement.

When a file is closed and then reopened, the next I/O statement to reference it will access the beginning of the file.

All files are automatically closed upon completion of program execution.

READ Statement

The READ statement is used to input data from a file and store the data as values of the variables in a list.

Form	Example
READ [n,f]list	READ [3, 50] A, B, (C(I), I=1, 10)
READ (n, f) list	READ (4, 'I 10") N
READ [n] list	READ [4] X, Y, ZED
READ (n) list	READ (3) Q, R

where

n is a file number.

f is a FORMAT statement number or a string literal.

The file number n is assigned in the OPEN statement. Data is converted from external to internal form according to format f, which may be (1) the number of a FORMAT statement, or (2) a format expressed as a character string enclosed in single and double primes. The format reference is omitted for binary files, as in the second form given above.

A READ accesses the teletype if n=0. The ⏎ is ignored until the list is satisfied; thus the ⏎ may be used to continue from one line to the next when entering data at a teletype. Once the list has been satisfied, a ⏎ is interpreted as an "end of record" character.

WRITE Statement

The WRITE statement causes the values of expressions in a list to be written on a file.

Form	Example
WRITE [n, f] list	WRITE [3, 1] X, Y, Z
WRITE [n] list	WRITE [4, 'F10.2"] A, B
WRITE (n, f) list	WRITE (4,800) JACK
WRITE (n) list	WRITE (3) C+1,D

where

n is a file number.

f is a FORMAT statement number or a string literal.

The file number n is assigned in an OPEN statement. Data is converted from internal to external format according to format f, which may be (1) the number of a FORMAT statement or (2) a format expressed as a character string. The format reference is omitted for binary files, as in the second form given above.

A WRITE accesses the teletype if n = 1. A (RET) is automatically issued at the end of a line if the output string exceeds 72 characters.

FORMAT Statement

FORMAT statements specify the conversion to be performed on data being transmitted during a formatted input/output operation. In general, conversion performed during output is the reverse of conversion performed in an input operation.

Form
FORMAT $(f_1, f_2, f_3, \ldots, f_n)$

where the f_i are field specifications (described in the following pages).

Each FORMAT statement must be numbered so that references may be made to it by formatted input/output statements. In addition, an entire FORMAT (the parentheses and the items they enclose but not the word FORMAT) may be stored in an array variable; in this case the array itself is referenced by the input/output statement. (See "FORMATs Stored in Arrays" at the end of this chapter.)

Field Specifications

Field specifications describe the kind or type of conversion to be performed, specific data to be generated, scaling of data values, and editing to be executed. Each integer, real, double precision, or logical datum appearing in an input/output list is processed by a single field specification while complex data are operated on by two consecutive field specifications.

Field specifications may be any of the following forms:

rFw.d	rIw	rJc.d	rX
rEw.d	rOw	nHs	r/
rDw.d	rLw	s	Z
rGw.d	rAw	's"	

where the characters F, E, D, G, I, O, L, A, J, H, $, single prime (') and double prime ("), X, slash (/), and Z define the type of conversion, data generation, scaling, editing, and FORMAT control

r is an optional, unsigned decimal integer which indicates that the specification is to be repeated r times. Thus 3I6 is equivalent to 6, 6, 6.

c for the J specification, is an unsigned decimal integer and specifies the number of digits appearing before the decimal point.

w is an unsigned decimal integer which defines width in characters (including digits, decimal points, and algebraic signs) of the external representation of the data being processed.

d for F, E, D, J, and input G specification, is an unsigned decimal integer and specifies the number of fractional digits appearing in the magnitude portion of the external field.

n for the H specification, is an unsigned decimal integer which defines the number of characters being processed.

s for the H specification, is a string of the characters acceptable to the processor.

These field specifications are discussed in the following sections.

F Conversion

F conversion takes the form

rFw.d

Integer, real, or either part of complex data may be processed by this form of conversion. The value of d allows for the appropriate number of digits in the fractional portion of the field.

OUTPUT

Internal values are converted to real constants, truncated at d decimal places with an overall length of w. The field is right justified with as many leading blank characters as necessary. Negative values are preceded with a minus sign. Consequently, for the specification F11.4

273.4	is converted to	273.4000
7	is converted to	7.0000
-.003	is converted to	-.0030
-442.30416	is converted to	-442.3041

If a value requires more positions than are allowed by the magnitude of w, an asterisk (*) will be output, followed by the sign and as many significant digits as possible. In order to insure that such a loss of digits does not occur, the following relation must hold true:

$w \geq d + 2 + n$

where n is the number of digits to the left of the decimal point.

INPUT

Input strings may take any of the integer or real forms discussed under "Numeric Input String" later in this chapter. Each string will be a length w with d characters in the fractional portion of the value. If a decimal point character is present in the input string, the value of d is ignored, and the number of digits in the fractional portion of the value will be explicitly defined by that decimal point character.

If a field is short-terminated (see "Termination of Input Strings" later in this chapter), the input string is considered to be left justified, and is filled with trailing zeros. Consequently, for the specification F10.3

33	is converted to	3300000.
802142	is converted to	8021420.
.34562	is converted to	.34562
-7.001	is converted to	-7.001

E Conversion

E conversion takes the form:

rEw.d

Integer, real, or either part of complex data may be processed by this form of conversion.

OUTPUT

Internal values are converted to real constants of the form

.ddd...dE±ee

where ddd...d represents d digits, while ±ee is interpreted as a multiplier of the form

10±ee

Internal values are truncated to d digits, and negative values are preceded by a minus sign. The external field is right justified and preceded by the appropriate number of blank characters. The following are examples for the specification E14.8:

90.4450	is converted to	.90445000E+02
-435739015.	is converted to	-.43573901E+09
.000375	is converted to	.37500000E-03
-1	is converted to	-.10000000E+01
.2	is converted to	.20000000E+00
0.0	is converted to	.00000000E+00

The field width is counted from the right and includes the exponent digits, the sign (minus or blank), the letter E, the magnitude digits, the decimal point, and the sign of the value (minus or blank). If a width specification is of insufficient magnitude to allow expression of an entire value, an asterisk will be output, followed by the sign, decimal point, E character, sign of the multiplier, and as many significant digits as possible. To prevent a loss of this nature, it is necessary to insure that the relation

$W \geq d + 6$

is present in the field specification.

INPUT

Input strings may take any of the integer or real forms discussed under "Numeric Input Strings" later in this chapter.

Examples:

Value	Specification	Converted to
10.3456E03	E10.2	10345.6
-113409E2	E11.6	-11.340900
-409385E-03	E11.02	-4.09385
849935E-02	E10.5	.0849935

First, the decimal point is positioned according to the specification. The value of the exponent is then applied to determine the actual poisition of the decimal point. In the second example, -113409E2 with a specification of E11.6 is interpreted as -.113409E02, which, when evaluated (i.e., $-.113409 \times 10^2$), becomes -11.340900. A decimal point in the input string overrides the specification, as in the first example.

J Conversion

J Conversion takes the form

rJc.d

Conversion of this type is similar to E conversion, except that c specifies the number of digits before the decimal point.

Field width is defined by the relation

$w = d + c + 6$

OUTPUT

Internal values are truncated to d digits and negative values are preceded by a minus sign. The following are examples for the specification J3.4:

123.0	is converted to	123.0000E+00
9.64931	is converted to	964.9310E-02
-.001	is converted to	-100.0000E-05

INPUT

On input, conversion is identical to E-type conversion.

D Conversion

D conversion takes the form

rDw.d

Conversion of this type is similar to E conversion, with the exception that for output, the character D will be present instead of the character E. For example,

for E12.6,	-667.334	is converted to	-.667334E+03
for D12.6	-667.334	is converted to	-.667334D+03

G Conversion

G conversion takes the form

rGw.d

Integer, real, or either part of complex data may be processed by this form of conversion.

OUTPUT

The purpose of the G format for output is to express numbers in a form which is most natural; that is, they are expressed in the form which is normally used for values of the corresponding magnitude.

Internal values are converted to real constants. The form of the constants is dependent upon the magnitude of the data, and conversion is either E or F-type, as indicated below, where M represents the magnitude of the data:

For $10^{i-1} \leq M < 10^{i}$

conversion will be

Fn.m

when $0 \leq i < d$ (where $n = w - 4$ and $m = d - i$)

Otherwise, conversion will be

Ew.d

Values converted with the F specification are followed by four blank characters in the external character string.

The following are examples for the specification G9.2:

-1.773	is converted to	-1.7bbbb
.133	is converted to	.13bbbb
532.	is converted to	.53E+03

-.0947	is converted to	-.94E-01
-.0996	is converted to	-.99E-01

where ♭ represents a blank.

If the magnitude of width w is insufficient to allow representation of the data value, digits are lost, as in E and F conversions.

INPUT

On input, processing is identical to F conversion.

I Conversion

I conversion takes the form

rIw

Integer, real, or either part of complex data may be processed by this form of conversion. If the width specification w is of sufficient magnitude, real values are converted in full precision.

OUTPUT

Internal values are converted to integer constants. Real data are truncated to integer values; however, the integers may contain as many digits as are specified by w.

Negative values are preceded by a minus sign, and the field will be right justified and preceded by the appropriate number of blank characters. The specification 16 implies that

273.4	is converted to	273
7	is converted to	7
-.003	is converted to	0
-44205.965	is converted to	-44205

If the magnitude of data requires more positions than permitted by the value of the width w, an asterisk will be output, followed by the sign and as many significant digits as possible.

INPUT

On input, conversion is identical to F-type processing except that fractional portions of a value are lost through truncation.

O Conversion

O conversion takes the form

rOw

OUTPUT

Internal binary word values, with no regard to data type, are converted to their octal equivalences. In order to fully represent each data type, real and integer data require 16 digits for the value of the width w.

Note that real data include either part of complex data and Hollerith information. Logical data cannot be output with an O conversion.

Example:

Data Values	Internal Binary	rOw	External Octal
1	0100000000000000000000000000000000000001	O16	2000000000000001

Example (cont.):

Data Values	Internal Binary	rOw	External Octal
5.0	01010000000000000000000 00000000000000000000011	O16	2400000000000003
HOL	00101000001011110010110O	O8	12027454

Whenever the magnitude of w is insufficient for the complete expression of a value, digits will be lost from the least significant portion of the field.

If w is of a magnitude greater than that necessary to express the octal representation of the data, the field in the external string will be right justified and preceded by the appropriate number of zero characters.

INPUT

External fields processed by O conversion may contain only strings of octal digits and blank characters. If a field contains other than one of the above, an error occurs.

Conversion begins with the first character in the string, including blanks. Blank characters are treated as if they were zero characters. Thus

␣3566716

is equivalent to

03500710

for the specification O8, where ␣ represents a blank. Also, fields that contain nothing but blank characters are assumed to have the value zero.

Fields which contain more significant digits than required by the corresponding list itmes lose digits from the least significant portion of the field. For instance, if the list item is integer, and the input specification use is O16, then

12345670076543211234567O

is converted to

1234567007654321

L Conversion

L conversion takes the form

rLw

Only logical data may be processed with this form of conversion; any other data type causes an error to occur.

OUTPUT

Logical values are converted to either TRUE or FALSE for the values "true" and "false", respectively. If the field width will not contain the full word, either a T or an F character is output. The T and F characters are preceded by w-1 blank characters. For example, using the specification L4,

.TRUE.	is converted to	TRUE
.FALSE.	is converted to	␣␣␣F

where ␣ represents the character blank.

INPUT

The first nonblank character in the input string must be either a T or an F character; any other character appearing as the first nonblank character causes an error to occur. The occurrence of a T or an F character causes the corresponding list item to be assigned the values "true" or "false", respectively.

Thus, the strings

TRUE

and

FALSE

are valid input strings. Characters falling between the T or F character and the righ-hand boundary of the external field are ignored. Fields consisting of only blank characters cause an error condition. A field cannot be terminated prematurely by a comma (see "Termination of Input Strings" later in this chapter).

A Conversion

A conversion takes the form

rAw

OUTPUT

Internal binary values are converted to character values at the rate of eight binary digits per character. The most significant digits are converted first, i.e., conversion is from left to right. Internal values are processed in the following manner:

Data Type	Internal Binary	rAw	External String
integer	001010010010111000110100	A3	INT
	000000000000000000000000	A2	IN
	(1222706400000000 octal)		
real	001100100010010100100001	A6	REAL [8
	001011000011101100011000	A11	␣␣␣␣␣REAL [8
	(1442244113035430 octal)		

When the magnitude of w does not provide for enough positions to express the data value completely (6 for real or integer) the external field is shortened from the right or least significant portion. When w has a value greater than necessary, the external character string is preceded by the appropriate number of blank characters.

This type of conversion is normally used to output Hollerith information which has been placed in storage.

INPUT

Hollerith input may be stored in real variables only. When the value of w is less than 6, the list item is filled with the w characters in the most significant positions, and the remainder of the positions are filled with blank characters. Consequently, if the field specification is A4,

UVWX is converted to UVWX␣␣

where ␣ represents the character blank.

When the width w is larger than 6, an error condition occurs.

A general rule for this type of conversion is that internal values are considered to be left justified, while external fields are considered to be right justified.

H Conversion

H conversion takes the form

nHs

OUTPUT

The n characters in the string s are transmitted to the external medium. For instance:

Specification	External String
1HE	E
8H␣␣VALUE:	␣␣VALUE:
5H$3.95	$3.95
9HX(2,5)␣=␣	X(2,5)␣=␣

where ␣ represents the character blank.

Care should be taken that the character string s contains exactly n characters, so that the desired external field will be created, and so that characters from other field specifications are not used as part of the string.

INPUT

The n characters in the string s are replaced by the next n characters from the input record. This replacement occurs as shown in the following examples:

Specification	Input String	Resultant Specification
3H123	ABC	3HABC
10HNOW␣IS␣THE	␣TIME␣FOR␣	10H␣TIME␣FOR␣
5HTRUE␣	FALSE	5HFALSE
6H␣␣␣␣␣␣	RANDOM	6HRANDOM

where ␣ represents the character blank. This feature can be used to change titles, dates, column headings, etc., which are to appear on an output record generated by the H specification.

If n is not present or is equal to zero, an error condition occurs.

$ Conversion

$ Conversion takes the form

s

The string s may contain any character other than a dollar sign character ($) and the control characters given in Chapter 2.

OUTPUT

The string s is transmitted to the external device in a manner similar to that for H conversion. Thus,

$DOLLAR SIGN$

is output as the string

DOLLAR SIGN

INPUT

The characters appearing between the dollar sign characters are replaced by the same number of characters taken sequentially from the input string. Therefore, for the input string

MATRIX

and the specification

$VECTOR$

the resultant specification is

$MATRIX$

Dollar sign characters may not appear in the input string.

' – '' Conversion

This type of conversion takes the form

's"

It is identical to $ conversion, with the exception that dollar sign characters may be present in the string s. The prime and double prime characters may be used within the string, but they must occur in pairs as they denote strings within strings.

Examples:

'TWAS BRILLIG AND THE SLITHY TOVES..."
'WHAT,'ME WORRY?'"

Blank characters in FORMAT statements are significant only in H $ and '-" specifications.

X Specifications

X specifications take the form

iX

These specifications cause no conversion to occur. Instead, they cause i positions of an external field to be "skipped".

OUTPUT

The next i positions of the output record will be blank characters. In other words, a field of i blank characters will be created. The specifications

$WXYZ$, 4X, 'IJKL"

cause the external string

WXYZ␣␣␣␣IJKL

to be generated, where ␣ represents the character blank.

INPUT

The next i characters from the input string are ignored. For example, with the specification

F5.3, 6X, I3

and the input string

76.42IGNORE597

the characters IGNORE will not be processed.

/ Specifications

Slash (/) specifications take the form

r/

Each slash specified causes another record to be processed. In the case of contiguous slash specifications (i.e., ////.../ or r/), since no conversion occurs between each of the slash specifications, records are ignored during input (scanned to a ⏎), and empty records are generated during output operations. The same condition can occur when a slash specification and either of the parentheses characters surrounding the field specifications are contigous.

OUTPUT

When a slash specification is encountered, the current record being processed is output and another record is begun. If no conversion has been performed when the slash is sensed, an empty record is created. (On the teletype, this would be a blank line.) The statements

```
WRITE [4, 10] A, B
10 FORMAT (F5.3,//,I 13)
```

are processed in the following manner:

1. A record is begun, and A is converted via the specification F5.3.

2. The first slash is encountered, the record containing the external representation of A is terminated, and another record is begun.

3. The second slash causes termination of the second record, and a third record is started. Since no conversion occurred between the terminations of the first and second records, the second record was empty.

4. The value of B is converted with the I13 specification, the closing right parenthesis character is encountered, and the third record is terminated.

If a third item C were added to the output list, as in

```
WRITE [4, 10] A, B, C
```

the following additional steps would occur:

5. A fourth record is begun, and C is converted using the specification F5.3.

6. The first slash is re-encountered, the fourth record is terminated, and a fifth record is begun.

7. Again, the second slash is processed; the fifth record, which is empty, is terminated; and the sixth record is started.

8. Since there are no more list items, the specification I13 is not processed, a termination occurs, and the final or sixth record, which is also empty, is output.

The original FORMAT statement could also have been written as

```
10 FORMAT (F5.3, 2/I13)
```

or

```
10 FORMAT (F5.3, 2/,I13)
```

with the identical effect.

The two statements

```
WRITE [3, 4] X

4 FORMAT (3/E6.4/)
```

cause the generation of three empty records, followed by a record containing the value of X, converted by the specification E6.4, followed by another empty record.

INPUT

The effect of slash specifications during input is similar to the effect for output, except that for input, records are ignored where empty records would be created during output. For example, the statements

```
WRITE [3, 4] X
4 FORMAT (3/E6.4/)
```

cause three records to be bypassed (i.e., three ⏎ to be read), a value from the fourth record to be converted with the specification E6.4 and assigned to X, and a fifth record to be bypassed. This means that, as with the last example for output, records created with a FORMAT statement containing slash specifications can be input by use of the identical FORMAT statement.

Z Conversion

This specification takes the form

```
Z
```

OUTPUT

On output the Z specification causes the suppression of the terminal normally issued upon termination of output. This feature is useful when outputting requests for input from the teletype. For example,

```
WRITE [1, 10]
10 FORMAT ($A=$, Z)
```

will cause

```
A=
```

to be printed at the teletype with the carriage positioned after the last character typed. If the next statement executed is an ACCEPT, the user's input will be typed on the same line.

INPUT

This specification is ignored on input.

Repetition Of Field Specifications

Within a FORMAT statement, any number of field specifications may be repeated by enclosing them within parentheses, preceded by a repeat count, in the following form:

$$r(f_1, f_2, f_3, \ldots, f_n)$$

where r is the repeat count and the f_i are specifications. Thus the statement

```
3 FORMAT (3(A4, F4.2, 3X), 3I)
```

is equivalent to

```
3 FORMAT (A4, F4.2, 3X, A4, F4.2, 3X, A4, F4.2, 3X, 3I)
```

The repetition count may be any number up to $2^{24}-1$.

During input/output processing, each repetitive specification is exhausted in turn, as in each singular specification.

Additional examples:

```
  34 FORMAT (4X, 2(A8, I1, 7G9.3), I4, 3(L5))
1125 FORMAT (/, A4, F9.7, 5(E14.8,2/), E14.8)
   8 FORMAT (7(I8, 2(3X, F12.9), F12.9), A16)
```

In the last example above, repetitions have been nested. Nesting of this type is permissible to a depth of ten levels.

Numeric Input Strings

Input strings processed by F, E, D, J, G, and I conversions may take any of the following forms:

±n

±n.m

±n±e

±n.m±e

±nE±e

±n.mE±e

where n, m, and e are strings of decimal digits or blank characters; plus sign characters are optional except prior to e when the character E is not present; and the decimal point and E characters must be present in that form. The character D may be substituted for the E character with no change in meaning or values.

Blank characters in the strings n, m, and e are treated as zero characters, as are n, m, and e if they are empty strings.

When conversion is via an I specification, fractional portions of a value are lost through truncation.

In all cases, conversion begins with the first nonblank character in the field, and blank characters falling between the E (or D) character and the exponent field ±e are ignored.

Termination of Input Strings

Normally, a READ statement inputs the exact number of characters in the field specification. However, the occurrence of a comma within a numerical input field causes termination of the field. Consider the following example:

```
 10.     READ(0,1000) I,A,B
 40.     WRITE(1,1000) I,A,B
 70.     1000 FORMAT(I5,F10.5,E12.4)
100.     END!
+ EXECUTE
  3,10.2,.32E+5, (RET)
    3  10.20000     .3200E+05
```

An error will result if a (RET) is input before the input list has been satisfied or if any field was not terminated properly. Consider the following program:

```
 10.     READ(0,2000) I,K,J
 50.     2000FORMAT(315)
 90.     END!
+ EXECUTE
3,7,5 (RET)
IO OR DATA ST. ERR.
 10.     READ(0,2000) I,K,J
```

Note that the last field did not have a width of I5 and was not terminated by a comma.

Editing of Input Strings

The following control characters allow an input string to be edited:

A^c delete the previous character.

W^c delete the previous word; i.e., delete all previous characters until the first blank is encountered.

Q^c delete the entire line.

A line feed may be used to continue to the next line if the editing requires an additional line.

FORMAT and List Interfacing

Formatted input/output operations are controlled by the FORMAT requested by each READ or WRITE statement. Each time a formatted READ or WRITE statement is executed, control is passed to the FORMAT processor, which operates in the following manner:

1. When control is initially received, the processor prepares to read a new record or line, or construction of a new output record or line is begun.

2. Subsequent records are started only after a slash specification has been processed (and the preceding record has been terminated) or the final right parenthesis of the FORMAT has been sensed, or the maximum number of characters for a teletype line has been output.

3. During an input operation, processing of an input record is terminated whenever a slash specification or the final right parenthesis of the FORMAT is sensed, or when the FORMAT processor requests an item from the list and no list items remain to be processed. Construction of an output record terminates, and the record is written on occurrence of the same conditions.

4. Every time a conversion specification (i.e., F, E, J, D, G, I, O, L, or A specification) is to be processed, the FORMAT processor requests a list item. If one or more items remain in the list, the processor performs the appropriate conversion and proceeds with the next field specification. (If conversion is not possible because of a conflict between a specification and a data type, an error occurs.) If the next field specification is one which does not require a list item (i.e., H, $, '-", Z, X, or /), it is processed whether or not another list item exists. When there are no list items still to be processed the current record is terminated and control is passed to the statement following the READ or WRITE statement which initiated the input/output operation.

5. When the final right parenthesis of a FORMAT statement is encountered by the FORMAT processor, a test is made to determine if all list items have been processed. When the list has been exhausted, the current record is terminated, and control is passed to the statement following the READ or WRITE which initiated the input/output operation. However, if another list item is present, an additional record is begun, and the FORMAT is re-scanned. The re-scan takes place as follows:

   a. When the FORMAT statement contains one or more groups of specifications enclosed with repetition-type parenthesization, the re-scan is started with the group whose right parenthesis character was the last one encountered prior to the final right parenthesis of the FORMAT statement.

   b. If no such group exists, the entire FORMAT is re-scanned.

6. Each list item to be converted is processed by one specification or one iteration of a repeated specification, with the exception of complex data, which are processed by two such specifications.

7. Each READ or WRITE statement containing a nonempty list must refer to a FORMAT statement which contains at least one conversion specification (see Step 4 above). If this condition is not met, the FORMAT statement will be processed, but an error will occur.

Formats Stored in Arrays (Not Implemented at the Time of Publication)

A FORMAT, including the beginning left parenthesis character, the final right parenthesis character, and the specifications enclosed therein (but not the word FORMAT) may be stored in an array variable. The FORMAT must be stored as a Hollerith constant (i.e., a string of characters) by use of either an input statement or an assignment statement.

READ or WRITE statements which refer to a FORMAT stored in an array must reference only the identifier of the array, with no subscription. For example

```
WRITE [4,R] E,F,G
```

refers to a FORMAT stored in an array R.

If the variable Z is a real array, and the string to be stored is (F8.5,4HNODE,I3), two methods may be used:

1. The string may be read in at execution time. For example

```
      READ [M,90] (Z(I),I=1,3)
90    FORMAT (3A6)
```

2. Assignment statements may be used to achieve the same effect. For example

```
Z(1) = 6H(F8.5,
Z(2) = 6H4HNODE
Z(3) = 6H,I3)
```

Care must be taken when storing into an array a FORMAT containing specifications of the nHs, s, and 's" forms. In these cases, all characters in the string s, including blank characters, are significant. For example, if an A4 format had been used to read in the string in the example above, the following results would have occurred:

Element	Storage after READ
Z(1)	(F8.␣␣
Z(2)	5,4H␣␣
Z(3)	NODE␣␣
Z(4)	,I3)␣␣

which is not the desired result, since it is equivalent to the FORMAT

```
(F8.5,4H␣␣NODE,I3)
```

where ␣ represents the character blank.

Even though a FORMAT may be quite short, it must be stored in an array rather than a scalar variable.

Using the teletype as the input file, this feature may be used to good advantage during on-line checkout. Programs may be tested with minimal formats, but once a program is operational, the output may be dressed up to any desired level. Or, during checkout, a part of the output may be suppressed altogether with F0.0, I0, or E0.0 specifications. On the other hand, it is much easier to design, test, and modify complex formats while actually observing program output on-line.

9. DECLARATION STATEMENTS

Declaration statements are used to define the data type of a variable or function subprogram, the dimensions of an array variable, the initial values of variable data, and to provide other similar information to the processor.†

Classification of Identifiers

Identifiers may be defined as being in any one of the following categories:

- scalar identifiers
- array identifiers
- subprogram identifiers

The category in which an identifier is placed, and the data type (if any) associated with it are dependent upon the context in which the identifier is initially defined. This definition is a declaration, explicit or implicit, of the way in which the identifier is to be categorized throughout the remainder of the program.

Implicit Declarations

Unless specifically declared to be in a particular category or type, identifiers which appear in executable or DATA statements are implicitly classified according to the following set of rules:

1. Any identifier appearing in a CALL statement as the called subprogram is a subprogram identifier. For example,

   CALL ERR or CALL NIX[R,V]

2. An identifier (other than defined in paragraph 1) which is followed by an argument list enclosed in brackets, such as A[T,ALPHA,B+C] is

   a. A statement function definition if it appears in the manner discussed under "Statement Functions" in Chapter 10.

   b. A function subprogram reference if it appears in an expression. This does not apply to identifiers appearing to the left of a replacement operator (=).

   c. An error if it appears to the left of a replacement operator in any statement other than a statement function definition.

3. An identifier which is not followed by an expression list enclosed in parentheses is defined as a scalar variable.

4. When applicable, the data type associated with an identifier is integer if the identifier begins with the letter I, J, K, L, M, or N. If the identifier begins with any other letter, its type is real.

5. An identifier which appears in a non-executable statement (other than a DATA statement), but never in an executable or DATA statement, is implicitly classified after all statements have been processed. Classification is in accordance with the previous set of rules and depends upon the classification defined by the non-executable statement in which the identifier appears.

Explicit Declarations

All classifications of identifiers other than those discussed in the previous section require explicit definition. Explicit definitions and declarations include:

- array declarations
- type statement declarations
- subprogram definitions

---

†The storage allocation statements COMMON and EQUIVALENCE have not yet been implemented.

Array Declarations

Array declarations explicitly define an identifier as the name of an array, and have the form:

$v(d_1, d_2, d_3, \ldots, d_n)$

where

- v is the identifier.
- n is the number of dimensions associated with the array.
- d_i define the range of the corresponding dimensions.

Each d_i may take the forms:

r_u

or

$r_o:r_u$

where

- r_u is an integer which defines the upper bound of the dimension range.
- r_o is an integer which defines the lower bound of the dimension range.

In the first form, the lower bound is assumed to be 1, and the upper bound must be positive. For example,

```
ARRAY(10)
```

defines ARRAY to be a one-dimensional array, with a range which has 1 as its lower bound and 10 as its upper, for a maximum of 10 elements.

In the second form, both the upper and lower bounds may be positive, negative, or zero valued as long as the value of the upper bound is greater than or equal to the value of the lower bound.

Examples:

```
ARRAY (4:9, 15, 0:1, -20:20)

CUBE (-10:-1, 5, 32)

PLANE (-999:0, 1:450)

LINE (140)

X(1)
```

In the first example given above, ARRAY is defined as a 4-dimensional array. The first dimension has a range of 4 to 9, the second of 1 to 15, and so on.

Array Storage

Although an array may have several dimensions, it is placed in storage as a linear string. This string contains the array elements in sequence (from low address storage toward high address storage) such that the leftmost dimension varies with the highest frequency, the next leftmost dimension varies with the next highest frequency, and so forth. Thus a two dimensional array would be stored "column-wise", i.e., with the row subscripts varying most frequently.

The following figures contain pictorial examples of array storage.

Array A(3, 3, 3)		Array B(-3:1, 0:4)	
Item	Element	Item	Element
1	A(1, 1, 1)	1	B(-3, 0)
2	A(2, 1, 1)	2	B(-2, 0)
3	A(3, 1, 1)	3	B(-1, 0)
4	A(1, 2, 1)	4	B(0, 0)
5	A(2, 2, 1)	5	B(1, 0)
6	A(3, 2, 1)	6	B(-3, 1)
7	A(1, 3, 1)	7	B(-2, 1)
8	A(2, 3, 1)	8	B(-1, 1)
9	A(3, 3, 1)	9	B(0, 1)
10	A(1, 1, 2)	10	B(1, 1)
11	A(2, 1, 2)	11	B(-3, 2)
12	A(3, 1, 2)	12	B(-2, 2)
13	A(1, 2, 2)	13	B(-1, 2)
14	A(2, 2, 2)	14	B(0, 2)
15	A(3, 2, 2)	15	B(1, 2)
16	A(1, 3, 2)	16	B(-3, 3)
17	A(2, 3, 2)	17	B(-2, 3)
18	A(3, 3, 2)	18	B(-1, 3)
19	A(1, 1, 3)	19	B(0, 3)
20	A(2, 1, 3)	20	B(1, 3)
21	A(3, 1, 3)	21	B(-3, 4)
22	A(1, 2, 3)	22	B(-2, 4)
23	A(2, 2, 3)	23	B(-1, 4)
24	A(3, 2, 3)	24	B(0, 4)
25	A(1, 3, 3)	25	B(1, 4)
26	A(2, 3, 3)		
27	A(3, 3, 3)		

References to Array Elements

References to array elements must contain the number of subscripts which correspond to the number of dimensions declared for the array. References which contain an incorrect number of subscripts are treated as errors.

Furthermore, the value of each subscript should be within the range of the corresponding dimension as specified in the array declaration. Otherwise the references will be treated as errors.

DIMENSION Statement

DIMENSION statements are nonexecutable statements used to define the dimensions of an array. Every array variable appearing in a source program must represent an element of an array declared in a DIMENSION statement. Any number of arrays may be dimensioned in a single DIMENSION statement.

Form	Example
DIMENSION $s_1, s_2, \ldots, s_n$	DIMENSION D(45, -50:50, 4), Y(5000), WHTAX(0:70) DIMENSION F(2, 3, 4, 5, 6), G(3)

where the s_i are array declarations.

Array declarations have been discussed in detail earlier in this chapter.

DATA Statement

DATA statements are used to initialize variables to declared values. If a DATA statement is unlabeled, the initialization occurs during loading of an executable program and prior to execution of the program. If the DATA statement has a statement label, it is treated as a normal program statement and is executed when reached in the course of program execution.

DATA statements have the general form:

DATA $d_1 d_2 d_3 \ldots d_n$

The d_i take the following form:

$k/c_1, c_2, c_3, \ldots, c_n/$

where

k is a list which is similar to an input list (see Chapter 8).

c_j are either constants or repeated groups of constants.

The purpose of the statement is to cause the variables in the list k to be assigned the values of the corresponding constants in c_j.

The following rules distinguish the list k from input lists:

1. No variables may be used in subscript expressions unless they are the control variable in an implied DO loop (i.e., v in $v = e_1, e_2, e_3$), or they appear earlier in the DATA statement since otherwise they have no values during initialization.†
2. The expressions in an implied DO loop may contain variables only under the same condition as described in Item One.†
3. All implied DO loops must be enclosed in parentheses.

The c_j, which are either constants or repeated groups of constants, may take any of the forms:

c

r*c

$r(c_1, c_2, c_3, \ldots, c_n)$

where

c may be a constant of any type.

r is an unsigned decimal integer whose value is the number of repetitions of each group. In the third form, r is optional, and if not present it has an assumed value of 1.

Example:

```
DATA X, (Y(I), I=1, 5), Z/32.5, 5*0.0, -7/R, Q/1.5E3, .TRUE./
```

has the same effect as the statements

```
X=32.5
DO 1 I=1, 5
1 Y(I)=0.0
Z=-7
R=1.5E3
Q=.TRUE.
```

except that the DATA statement is effective prior to execution of the program, since it is unlabeled. Note that the expression 5*0.0 in the above example does not mean 5 times 0, but rather five zeros.

If the data type of a constant is not the same as the data type of the variable to which it is assigned, conversion occurs according to the rules in Chapter 6.

†Applies only to unlabeled DATA statements

The list k must specify at least as many items as are specified by the list of constants. If the list k specifies more items than the list of constants, the list of constants is repeated until all the items in the list k have been assigned values. For example, if there are 3 items in the list k, the DATA statement

```
DATA A, B, C/3 * 1.0/
```

is equivalent to

```
DATA A, B, C/1.0/
```

and

```
DATA A, B, C/1.0, 2.0/
```

is equivalent to

```
A = 1.0
B = 2.0
C = 1.0
```

Variables of complex data type which appear in the list k require two constants per datum for initialization. The first of the two constants initializes the real part, the second the imaginary part of the complex datum. Two constants used for this purpose may be written as:

(C_1, c_2)

which is a repeated group of two constants with an implied repeat count of one (i.e., the same as $1(c_1, c_2)$). Consequently

```
COMPLEX T
REAL R, S, U
DATA R, S, T, U/5, -48.3, (34, 8), 111/
```

are equivalent to

```
COMPLEX T
REAL R, S, U
R = 5
S = -48.3
T = (34, 8)
U = 111
```

Type Statements

Type Statements are used to explicitly define the data type of a variable or function subprogram.

Form	Example
$d_t v_1, v_2, \ldots, v_n$	INTEGER R, F, I, E(-5:10, 15) REAL M, KING(55, 55) LOGICAL SEQ, BOOLE(5, 5, 5, 5)

where

d_t is a data type.

v_i are identifiers of variables or function subprograms, or they are array declarations.

The possible data types are:

```
INTEGER
REAL
COMPLEX
LOGICAL
```

Type statements may appear anywhere in a program.

10. SUBPROGRAMS[t]

A subprogram is one or more lines of code executed when called upon by name by another program. The purpose of a subprogram is to make it more convenient to perform frequently occurring operations.

There are two general categories of subprograms:

1. A function subprogram is called implicitly by using its name in an expression, and it returns a single result through its identifier.
2. A subroutine subprogram is called explicitly by a CALL statement, and may return more than one value through arguments.

Each of these categories will now be discussed in detail.

Function Subprograms

Function subprograms are programmed procedures which are often used to provide solutions to mathematical functions and which are used in a manner similar to that of normal mathematical notation. For example, there is a library cosine function whose identifier is COS, thus allowing

```
y=cos x
```

to be written as

```
Y=COS X
```

The appearance of the identifier COS constitutes a call to the standard library subprogram COS, which is available to all FORTRAN users. Control transfers to the function, which, when executed, returns a value to the function reference in the calling program. The calling program can then use this value as it would any other.

Thus, function references may be used in the same manner as variable references in any expression. For example,

```
X=(B+SQRT B**2-4*A*C )/2*A
```

where

SQRT is the identifier of the square root function

B**2-4*A*C is the calling argument list.

There are three types of function subprograms:

Library Functions
Statement Functions
FUNCTION Subprograms[t]

Library Functions

Library ("intrinsic") functions are subprograms which evaluate commonly used mathematical functions. They are contained in the CF library. These functions have inherent data type classifications, as given in the table below. In the table, C signifies a complex value; I, an integer value; L, a logical value; and R a real value. N means number.

Library Function Names	Type of Function	Number of Arguments	Type of Arguments	Definition of Function
ABS	R	1	I, R	Absolute value.
AIMAG	R	1	I, R, C	The imaginary part of the argument (zero if not complex) expressed as a real value.

[t]Where the word "function" is capitalized in this text, the reference is to the specific type of function subprogram which begins with a FUNCTION statement (as discussed later in this chapter).

Library Function Names	Type of Function	Number of Arguments	Type of Arguments	Definition of Function
AINT	R	1	I, R	The integer part of the argument expressed as a real value.
ALOG	R	1	I, R	Natural logarithm (base e).
ALOG10	R	1	I, R	Common logarithm (base 10).
AMAX1	R	N 1	I, R	Maximum value. All arguments are converted to and compared as real values.
AMAX0	R	N 1	I, R	Maximum value. All arguments are converted to and compared as integer values.
AMIN1	R	N 1	I, R	Minimum value. All arguments are converted to and compared as real values.
AMIN0	R	N 1	I, R	Minimum value. All arguments are converted to and compared as integer values.
AMOD	R	2	I, R	Arg_1 (mod arg_2). Evaluated as $arg_1 - arg_2 * AINT[arg_1/arg_2]$; i.e., the sign is the same as arg_1.
ATAN ATAN2	R	1, 2	I, R	Arctangent of argument. If two arguments, quadrant allocated between -pi and +pi.
CABS	R	1	I, R, C	Complex absolute value; i.e., modulus.
CATAN	C	1	I, R, C	Complex arctangent.
CCOS	C	1	I, R, C	Complex cosine.
CCOSH	C	1	I, R, C	Complex hyperbolic cosine.
CEXP	C	1	I, R, C	Complex exponential. (e**arg)
CINT	C	1	I, R, C	Complex number formed by the integer values of the real and imaginary parts of argument.
CLOG	C	1	I, R, C	Complex natural logarithm (base e). Allocated between -pi and +pi.
CLOG10	C	1	I, R, C	Complex common logarithm (base 10). Allocated between -pi and +pi.
CMPLX	C	2	I, R	Complex number where real part $=arg_1$, imaginary part $=arg_2$; i.e., converts two real numbers to a complex number.
CONJG	C	1	I, R, C	Complex conjugate. (Has no effect if argument is not complex.)
COS	R	1	I, R	Cosine.
COSH	R	1	I, R	Hyperbolic cosine.
CSIN	C	1	I, R, C	Complex sine.
CSINH	C	1	I, R, C	Complex hyperbolic sine.
CSQRT	C	1	I, R, C	Complex square root. Allocated between -pi/2 and +pi/2; i.e., the real part is positive.
EXP	R	1	I, R	Exponential. (e**arg)
FLOAT SNGL	R	1	I, R	Argument converted to a real value.
IABS	I	1	I, R	Integer absolute value.
IDIM	I	2	I, R	Positive difference; i.e., $arg_1 - MIN[arg_1, arg_2]$.
IFIX	I	1	I, R	Argument converted to an integer value.

Library Function Names	Type of Function	Number of Arguments	Type of Arguments	Definition of Function
ISIGN	I	2	I, R	Magnitude of arg_1 with sign of arg_2. If arg_2 is zero, the sign is positive.
MAX MAX0	I	N 1	I, R	Maximum value. All arguments are converted to and compared as integer values.
MAX1	I	N 1	I, R	Maximum value. All arguments are converted to and compared as real values.
MIN MIN0	I	N 1	I, R	Minimum value. All arguments are converted to and compared as integer values.
MIN1	I	N 1	I, R	Minimum value. All arguments are converted to and compared as real values.
MOD	I	2	I, R	Arg_1(mod arg_2). Evaluated as $arg_1-arg_2*FIX[arg_1/arg_2]$, i.e., the sign is the same as arg_1.
REAL	R	1	I, R	Argument converted to a real value. Same as FLOAT and SNGL except accepts complex arguments and returns the real part.
SIGN	R	2	I, R	Magnitude of arg_1 with sign of arg_2. If arg_2 is zero, the sign is positive.
SIN	R	1	I, R	Sine.
SINH	R	1	I, R	Hyperbolic sine.
SNGL	R	1	I, R	See FLOAT.
SQRT	R	1	I, R	Square root.
TANH	R	1	I, R	Hyperbolic tangent.

Statement Functions

Statement functions are function subprograms which can be defined in a single expression within the calling program. The definition holds only in the program or subprogram containing it.

Form	Example
$f[a_1, a_2, a_3, \ldots, a_n]=e$	NUMBER[K] K*(K+1)/2 EI[THETA] =CMPLX[COS[THETA]] QTH[OM] = NAME[OM] +ADDR[OM] SWITCH CK[A,B,C]=FLAG[A] .AND.FLAG[B].AND.FLAG[C]

where

f is the function identifier.

a_i are dummy function arguments.[t]

e is an expression.

[t]Dummy arguments serve as placeholders for the actual arguments provided by the calling program at execution time. Since the rules governing dummies apply to all subprograms, they are discussed separately later in this chapter.

Once a statement function has been defined, the appearance of its name in an expression suffices to call the function during the evaluation of the expression at run time. The function name is accompanied by the actual arguments to be used in evaluating the expression. For example, if the function were defined as

```
F [X] =Z*X**2+B*X+C
```

it might be referenced in the program statement as

```
RESULT = F [Y]
```

The current value of Y would replace the dummy argument X in evaluation of the function. The value of the expression

```
A*Y**2+B*Y+C
```

would be returned to the calling program where it would be assigned to RESULT.

A statement function must have at least one argument. The expression e must be of a mode which may be assigned to data of the type declared (implicitly or explicitly) for the function f (see Rules for Assignments of e to v, Chapter 6). References in the expression are unrestricted with the exception that the identifier of the function f itself may not appear; however, any other statement function may be referenced.

Since each a_i is merely a dummy and as such does not actually exist in storage, the identifiers used to represent the a_i may be the same as any other identifier, except those referenced within the expression e, without conflict.

If it is implicitly typed, a statement function is considered integer type if its identifier begins with I, J, K, L, M, or N; otherwise it is considered real type. If a statement function is to be typed explicitly, its identifier must appear in a Type statement prior to the definition of the function.

As stated previously, a statement function may be referenced only within the program or subprogram in which it is defined. Statement function definitions must precede all executable statements in the program or subprogram in which they appear.

FUNCTION Subprogram

Functions which cannot be defined in a single statement may be defined external to the calling program as FUNCTION programs. Like statement functions, the FUNCTION subprogram computes a value and returns that value through the function identifer.

The FUNCTION subprogram must begin with a function statement:

Form	Example
FUNCTION f$[a_1,a_2,a_3,\ldots,a_n]$	FUNCTION DIFFEQ R, S, N FUNCTION IOU W, X, Y, Z1 FUNCTION ROUND OMEGA

where

f is the function identifier.

a_i are dummy arguments

Each FUNCTION subprogram must have at least one argument. Values may be assigned to arguments within the subprogram without restrictions.

A FUNCTION subprogram must contain at least one RETURN statement to transfer control back to the calling program. A RETURN is the last statement executed in the function (see Chapter 7).

Within the subprogram, the identifier of the FUNCTION is treated as though it were a scalar variable, and it must be assigned a value during each execution of the subprogram. The value returned for a FUNCTION is the last one assigned to its identifier prior to the execution of a RETURN.

A FUNCTION subprogram typed implicitly is considered to be integer type if its identifier begins with the letters I, J, K, L, M, or N: otherwise it is considered to be real.

The following example of a FUNCTION subprogram finds the product of two 1-dimentional arrays with 3 elements each:

```
      FUNCTION DOT  V1, V2
      DOT=0
      DO 2 K=1, 3
2     DOT=DOT + V1 (K) * V2(K)
      RETURN
      END
```

If this function is called by the statement

```
PROD=DOT  A, B
```

the dummies V1 and V2 will be replaced by A and B at execution time. The value of the function is the single quantity DOT, which will be returned to the calling program and assigned to the variable PROD. Note that while A and B must be dimensioned in the calling program, V1 and V2 need not be dimensioned in the function (see Dummy Arguments below).

Subroutine Subprograms

Subroutine subprograms, like function subprograms, are self-contained programmed procedures. However, unlike functions, subroutines do not have values associated with their identifiers, and they may not be referenced in an expression. Instead, subroutines return values to the calling program by assigning values to arguments and are accessed by CALL statements (see Chapter 7).

Subroutine subprograms always begin with a SUBROUTINE statement:

Form	Example
SUBROUTINE p $a_1,a_2,a_3,\ldots,a_n$	SUBROUTINE CHECK SUBROUTINE VI ROMAN SUBROUTINE OUTPUT ARRAYS A, FMT, I, J

where

p is the subroutine identifier.

a_i are subroutine dummy arguments.

If no arguments are to be passed to the subroutine by the calling program or subprogram, the list of a_i and the comma and bracket characters would not be present; otherwise they are required.

A subroutine must contain at least one RETURN statement to transfer control back to the calling program; it should be the last statement to be executed during a run.

The following is an example of a subroutine subprogram which finds the cross product of two 1-dimensional arrays with 3 elements each:

```
SUBROUTINE CROSS  A, B, C
C(1) = A(2) * B(3) - A(3) * B(2)
C(2) = A(1) * B(1) - A(1) * B(3)
C(3) = A(1) * B(2) - A(2) * B(1)
RETURN
END
```

The call to this subroutine might be:

```
CALL CROSS  X, Y, Z
```

in which case the current values for array elements in X and Y would be used during execution of the subprogram, and the results would be assigned to elements in Z. Note that although X, Y, and Z should be dimensioned in the calling program, A, B, and C need not be dimensioned in the subroutine (see Dummy Arguments below).

Since the name of a subroutine plays no part in the result, it has no bearing on the mode of the result.

Dummy Arguments

Dummy arguments provide a means of passing information between a subprogram and the program which called it. Both function and subroutine subprograms may have dummy arguments, but a subroutine need not have any, while a function must have at least one. Dummies are merely "formal" arguments, used to indicate the number and sequence of subprogram parameters. They do not actually exist in that no storage areas are used for them.[†] They serve as place-holders for the calling arguments.

The calling arguments themselves may be scalar variables, array elements, array names, expressions, or subprogram identifiers. However, the dummies corresponding to these are always written as unsubscripted identifiers. A dummy argument need not conform to the data type of the corresponding calling argument. It should not be defined in a type statement within the subprogram. It need not be dimensioned in the subprogram unless, for example, the programmer wishes to modify the dimensionality of the calling array or the ranges of its dimensions. However, except when array declarations are modified, declarations in effect for calling arguments at execution time are those which prevail during execution of the subprogram. When a dummy corresponds to a variable in the calling argument list, a reference to the dummy is actually a reference to the calling argument variable. Not only will the dummy initially have the value which the calling argument was assigned at the time of the call, but any value subsequently assigned to the dummy will actually be assigned to the calling variable, thus effectively returning a result through the argument list. For effectively returning a result through the argument list. For example, if the calling statement for a function subprogram is

```
Y=X**NI+SQRTDS   Z, Q
```

and the function called is

```
FUNCTION SQRTDS   A, B
C=AMAX1  A, B
B=AMIN1  A, B
A=C
SQRTDS=SQRT   A**2-B**2
RETURN
END
```

then the values of Z and Q will be reversed whenever the initial value of Q is greater than that of Z.

When a dummy corresponds to an expression other than a single variable, the expression serves to initialize the value of the dummy. In this case storage is actually reserved for the dummy, whose value may be modified within the subprogram. For example, if the constant 3 in the calling list corresponds to the dummy J in the subprogram list, J will be initialized to 3 and may be modified in the subprogram. However, no result is returned through J and the value of the constant in the calling program is not affected.

If a dummy array is dimensioned, it may not contain more elements that declared for the corresponding calling array. In subprograms, a range specified in an array declaration may be indicated by a variable, providing the variable is itself a dummy:

[†]More precisely, storage areas are associated with dummies but contain pointers back to the storage areas for the calling arguments.

Example:

```
DIMENSION  A(100)          SUBROUTINE  X  [S, C]
.                          DIMENSION  S  [C, C]
.                          .
.                          .
B = 10                     .
CALL X [A, B]              S(2, 3) = 156
.                          .
.                          .
.                          .
END                        RETURN
                           END
```

Note: Dummy variable values are lost after return to the calling program. Thus the user should not repeatedly reference a subprogram expecting all values to be intact.

APPENDIX A. SUMMARY

Commands

	Command Abbreviations
COMPILE, sd	CO
COMPILE, sd, file name	CO
LIST	LI
LIST, sd	LI
LIST, sd, file name	LI
DELETE, sd	DE
KILL	K
EXECUTE	EK
WHY	W
BREAKPOINT, sd	B
PERFORM, fs	PE
PROCEED	PRO
NEXT	N
EDIT, sd	ED
CLEAR, sd	CL
PRINT, file name	PRI
SAVE, sd, file name	S

FORTRAN Statements

v = e (replacement)

ASSIGN k TO v

GO TO k
GO TO v
GO TO v, $(k_1, k_2, k_3, \ldots, k_n$
GO TO $(k_1, k_2, k_3, \ldots, k_n)$

IF (e) k_1, k_2, k_3

IF (e) fs

DO k v=e_1, e_2, e_3

CONTINUE

PAUSE
PAUSE c

STOP
STOP c

CALL p
CALL p$[a_1, a_2, a_3, \ldots, a_n]$

---

†Abbreviations used in this Summary are as follows, unless otherwise indicated:

sd statement designator	e expression	n file number (except where used as subscript
fs FORTRAN statement	c character string	f format reference
v variable	p subroutine name	s array declaration
k statement label	a argument	d_t data type

RETURN

ACCEPT [list]

DISPLAY [list]

OPEN [file number, file name, use, type]

CLOSE [file number]

READ [n, f] list
READ [n] list

WRITE [n, f] list
WRITE [n] list

FORMAT $(f_1, f_2, f_3, \ldots, f_n)$ — Where the f_i are field specifications.

DIMENSION $s_1, s_2, s_3, \ldots s_n$

DATA $d_1 d_2 d_3, \ldots, d_n$ — Where the d_i are lists of variables and constants (see text).

$d_t v_1, v_2, \ldots, v_n$ — Type Statement – where the v_i are identifiers of variables or function subprograms, or they are array declarations.

$f\,[a_1, a_2, a_3, \ldots, a_n] = e$ — Function Definition – where f is a function identifier.

FUNCTION $f\,[a_1, a_2, a_3, \ldots, a_n]$ — Where f is a function identifier.

SUBROUTINE $p\,[a_1, a_2, a_3, \ldots, a_n]$

APPENDIX B. SDS 940 INTERNAL ASCII AND TELETYPE CODES

Internal ASCII	TTY	Internal ASCII	TTY
00		40	@
01	!	41	A
02	"	42	B
03	#	43	C
04	$	44	D
05	%	45	E
06	&	46	F
07	'	47	G
10	(	50	H
11	)	51	I
12	*	52	J
13	+	53	K
14	,	54	L
15	-	55	M
16	.	56	N
17	/	57	O
20	0	60	P
21	1	61	Q
22	2	62	R
23	3	63	S
24	4	64	T
25	5	65	U
26	6	66	V
27	7	67	W
30	8	70	X
31	9	71	Y
32	:	72	Z
33	;	73	[
34	<	74	\
35	=	75	]
36	>	76	↑
37	?	77	←

INDEX

F

G

H

I

J

K

L

M

N

O