HONEYWELL

MULTICS FORTRAN MANUAL

SOFTWARE

LEVEL 68

MULTICS FORTRAN MANUAL

SUBJECT

Additions and Changes to Multics FORTRAN

SPECIAL INSTRUCTIONS

This manual supersedes AT58-02, dated December 1979, and its Addendum A, dated August 1980. Throughout the manual, change bars in the margins indicate technical additions and changes; asterisks denote deletions.

SOFTWARE SUPPORTED

Multics Software Release 9.1

ORDER NUMBER

AT58-03

December 1981

Honeywell

PREFACE

This document describes the FORTRAN language to be used on the Multics system. It is intended as a reference rather than a users' manual. The user is assumed to be familiar with some algebraic language. For further information about how to use Multics FORTRAN, or for a FORTRAN-oriented introduction to the Multics system, see the FORTRAN Users' Guide, Order No. CC70.

Warning

Multics FORTRAN is an implementation of FORTRAN IV as specified in American Standard FORTRAN, X3.9-1966, with extensions, and features of FORTRAN 77 as specified in American National Standard Programming Language FORTRAN, X3.9-1978.

Some but not all of the features of FORTRAN 77 are available only if the program is compiled with the ansi77 option in effect. Only those features that are incompatible with the ansi66 interpretation are under control of the ansi77 option. Almost all of the FORTRAN 77 standard features have now been implemented. See Appendix B for a list of the incompatible differences between ansi66 and ansi77.

It is possible to write FORTRAN programs that are, according to one of the standards, invalid, and for these programs to compile and execute in Multics without reported error. It should be clearly understood that any program in violation of the standard is not a valid program, and the results are undefined. As such there is no guarantee that a program dependent on constructs and values expressly stated to be undefined will produce correct or even consistent results now or in the future. In this manual such constraints are identified by "must," "must not," "cannot," "invalid," "undefined," or "in error."

File No.: 1L23

AT58-03

Notation

The notation of the metalanguage used in this manual is derived from that of the American National Standard Programming Language FORTRAN, X3.9-1978. The following conventions apply:

1. Letters, subscripted letters, and words indicate generalized items that must be replaced by particular items in actual statements.

2. Brackets, [], indicate optional items. A sequence of repeated letters, of which some are bracketed (as a[,a]) indicates distinct items that must be replaced by distinct items in actual statements.

3. An ellipsis,..., indicates that the preceding optional items may appear zero or more times in succession.

4. Where blanks appear they are for enhanced readability, except where noted otherwise.

Significant Changes in AT58-03B

Implementation of Large Arrays and Very Large Arrays.

Archive components can now be compiled.

Character constants may span multiple records in list-directed input.

Single record created from list-directed output.

Appendix A comparison of FORTRAN features extensively revised.

Only one feature of FORTRAN 77 is not yet added to Multics:

variable-expression array bounds

For purposes of clarity and ease of use, the MPM set has been reorganized. The six former MPM manuals, the Tools manual, and the RCP Users' Guide have been consolidated into a new set of three manuals.

Multics Programmer's Reference Manual (AG91)
contains all the reference material from the former eight manuals.

Multics Commands and Active Functions (AG92)
contains all the commands and active functions from the former eight manuals.

Multics Subroutines and Input/Output Modules (AG93)
contains all the subroutines and I/O modules from the former eight manuals.

The following manuals are obsolete:

Name	Order No.
MPM Peripheral Input/Output	AX49
MPM Subsystem Writers' Guide	AK92
Programming Tools	AZ03
MPM Communications I/O	CC92
Resource Control Users'Guide	CT38

References to these manuals still exist on pages not published with this addendum. When this manual is revised, the references in the text to the old manuals will be changed to reflect the new organization.

CONTENTS

ILLUSTRATIONS

Page

TABLES

SECTION 1

BASIC ELEMENTS OF A FORTRAN PROGRAM

A FORTRAN program consists of one or more distinct units, each of which can be compiled separately. The term program unit is applied to any sequence of FORTRAN statements and comment lines that is terminated by an end statement. This can be a main program or one of the three defined types of subprogram: subroutines, functions, and block data subprograms. Each executable program must have one and only one main program, and can have any number of subroutine, function, or block data subprograms. It is a feature of the Multics implementation that a subroutine taking no arguments is directly executable from command level.

The execution of an executable program begins with the main program. Associated program units are invoked by the call statement or by function references in expressions, which are resolved to the subprograms compiled with the referencing program unit or, if this fails, located using a linking mechanism specific to the particular environment of the run (information on environment-related topics is in the FORTRAN Users' Guide).

An executable program terminates either when a stop statement is executed or when control reaches the end statement of a main program. (See the FORTRAN Users' Guide for an explanation of the stop statement and the termination of an executable program.)

COMPILER FUNCTION

Input to a compilation is the ASCII text representation of one or more program units contained in a source segment or archive component. A main program must be the first program unit in the source segment in which it resides.

The output of compilation is an object segment, containing the binary representation of the source text. Entry points are defined within this segment for the beginning of each program unit and for additional entry points that may have been defined within the subprograms. All of these entry points are valid targets of call statements and function references when the program is executed, whether or not the referencing program unit is compiled with the unit it calls. Different program units in a single source segment are compiled in a single object segment. Program units may also be compiled independently: the fact that a program unit is not in the same object segment as the referencing program does not alter the validity of the call statement or function reference.

STATEMENTS

A program unit is composed of statements that describe either some portion of an algorithm or some aspect of the data to be processed. Comment lines can appear anywhere within a program unit. Conventions for designating comment and statement lines are described under "Program Structure" below. Not all the

statements described below are required in any given program unit, but when such statements appear, the prescribed statement order must be followed.

Order of Statements

Statements in a FORTRAN program unit can be executable or nonexecutable (the latter descriptive of data items or input/output formats). Format, declarative, and data statements can appear anywhere after an implicit statement -- if there is one -- and before the end statement. The %global and %option statements must be the first statements in any program unit in which they appear. Statement function definition statements must precede executable statements. Declarative statements for a particular variable must precede the first executable statement or the first namelist statement referencing that variable. A program unit can contain at most one implicit statement, and a subprogram can contain no more than one subroutine, one function, or one block data statement. The order of statements in a subprogram is the same as in a main program. A subprogram must begin with a subroutine, function, or block data statement. By contrast, the program statement in a main program is optional. If it is used, it must be the first statement of the main program. The syntax of this statement is:

```
program program_name
```

where program_name is the name given to the main program. It must conform to the requirements of symbolic names, described in Section 3. The actual order statements must follow is given in Figure 1-1 below.

When the program statement is used, the object segment produced will have as its internal entry point name the name specified in the program statement. If no program statement is used, the object segment will have the name "main_". Note that the dynamic linker treats the name "main_" specially (see the FORTRAN Users' Guide), and this special treatment is not given to names supplied with the program statement.

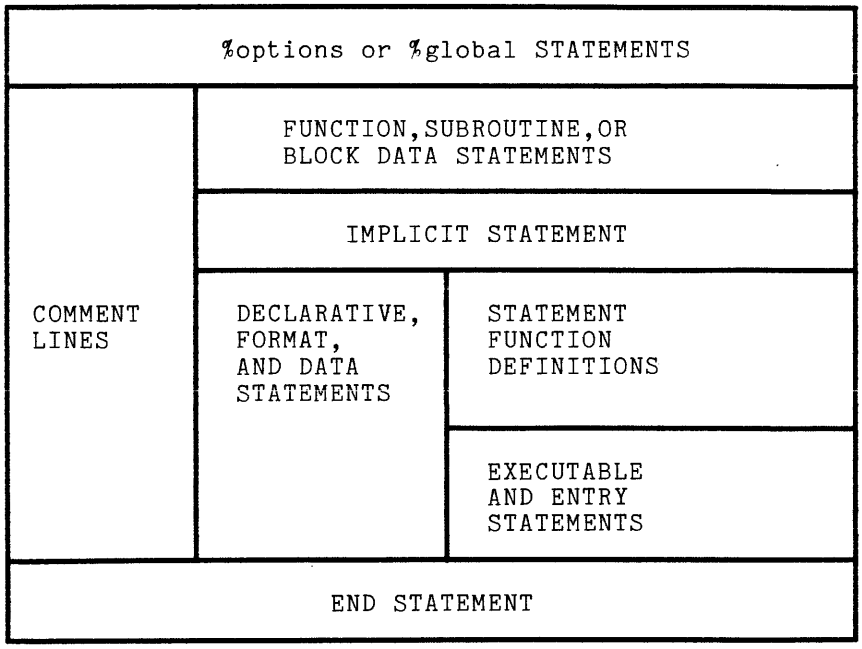


Figure 1-1. Order of Statements

A block data program is used to initialize variables in common blocks. The first statement says "block data," followed by, in order, common statements for the block or blocks being initialized, any declarative statements for the variables within the common blocks, data statements initializing those variables that have to be initialized and finally an end statement.

Statement Labels

In FORTRAN an unsigned integer of from 1 to 5 digits, containing no blanks, and one of which must be nonzero, is used to label a format statement or an executable statement, making it possible to reference that statement. The label appears at the beginning of the first line of a statement, and need not be separated from it by a delimiter. If line numbers are used (see "Line Numbers" below), the label follows the line number and must be separated from it by at least one blank character.

A statement label is not a line number.

Example:

```
5  a=b+c
   .
   .
   .
   go to 5
```

PROGRAM STRUCTURE

A FORTRAN program unit begins with the first statement in the source segment. A statement begins with an initial line and may be continued on one or more continuation lines. The text of a continued statement can be broken at any point, since continuation text is appended directly to text on the preceding line for compilation purposes. (In card-image format, the text on the preceding line is always at most 72 characters in length. See "Input Formats" below.)

A program unit ends with an end statement. This is an initial line whose nonblank characters are the letters "end". Any continuation lines following this initial line are in error.

INPUT FORMATS

Two formats can be used to enter source program text: free and card-image. In free format, the compiler interprets the line character by character. In card-image format, the compiler interprets the line according to distinct fields corresponding to card columns.

The compiler ignores blank characters except within a character-string constant.

Example:

```
a = b + c
```

is equivalent to:

```
a=b+c
```

The compiler treats the ASCII horizontal tab character (HT) as a single blank character (except within a character-string constant).

Comments can be included and are indicated using the conventions described below. For example, in both input formats, a line that begins with an uppercase or lowercase c is ignored by the compiler and can be used to contain comments.

Free Format

The compiler interprets free-format input text character by character according to the conventions given below. A line that is neither a comment nor a continuation line is an initial line. A blank line is permitted and is treated as an initial line.

COMMENTS

A line whose first character is an uppercase or lowercase c is a comment line, and is ignored except for listing purposes.

A line whose first nonblank character is an asterisk (*) or exclamation mark (!) is a comment line, and is ignored except for listing purposes.

An exclamation mark, except within a character-string constant, is a comment indicator at any point, and subsequent text on the containing line is a comment.

CONTINUATION LINES

A line whose first nonblank character is an ampersand (&) is a continuation line, and subsequent text is concatenated to the text on the preceding line.

BLANK LINES

Blank lines are treated differently by the ansi66 and ansi77 options. The ansi77 option treats them as comment lines and thus ignores them. The ansi66 option treats them as initial lines. Hence when a blank line precedes a continuation line, the latter is treated as a continuation of the blank line.

SEMICOLON

The semicolon (;) can function as a separator to allow more than one statement to appear on a line. Statements beginning after a semicolon on the same line as the semicolon cannot have labels. A semicolon cannot appear on an end line.

Line numbers, a free-format input option, are sequence numbers that do not constitute part of a program unit and appear at the beginning of each line. When line numbers appear, the first character on the line must be numeric. A line number is an unsigned integer constant of five or fewer digits. The highest line number allowable is 16383. The line number is terminated by the first nonnumeric character on a line, including the blank character. Embedded blanks are, therefore, not permitted. Line numbers must be unique and must be in strictly ascending order. Line numbers appear on program listings and are used to identify the erroneous statement for error messages during compilation.

Card-Image Format

Card-image format lines are restricted to 80 characters. The compiler prints a warning message if a source program line exceeds this length. A completely blank line is valid and is treated as an initial line in ansi66 and as a continuation line in ansi77. A line can contain less than 80 characters, in which case the missing character positions are treated as blanks.

The interpretation of a line is based on the contents of the fields shown below:

character position	field	interpretation
1-5	label	If the first character is a c, C, or *, then this line is a comment line. If this line is a continuation line (not a comment line, in particular), this field is ignored. If this line is an initial line, this field must be all blank or contain 1-5 numeric characters; the numeric characters are concatenated to form the statement label.
6	continuation	If this line is a comment line, this field is ignored. If this field is blank or zero, this is an initial line. If this field is not blank or zero, this is a continuation line.
7-72	statement text	This field contains text of a statement; if the line is less than 72 characters long, it is extended to 72 characters by inserting the appropriate number of blank characters.
73-80	identification	This field is ignored for all lines.

The %include Statement

Syntax:

```
%include partial_segment_name
```

where partial_segment_name is the entryname of an include file without the ".incl.fortran" suffix. A partial segment name can consist of anything but space characters.

An include statement must appear alone on an initial line. The compiler ignores anything on the line after the entryname of the include file.

The compiler processes the %include statement as follows: it appends the ".incl.fortran" suffix to the entryname and locates the include file using the translator search list. The program is then compiled as if the %include statement were a comment, and as if the include file itself were inserted between the %include statement and the next statement in the source. The total length of the name of the include file, after the addition of the suffix, must be no more than 32 characters.

The %include statement is not a statement in FORTRAN as defined either by the ANSI Standard or by Multics FORTRAN. Consequently both case and blanks have significance, regardless of the compiler options selected (i.e., free-form or card-image). The include file must be compatible with the input format of the source program in which the %include statement appears.

An included segment may itself contain a %include statement; the maximum depth of nesting is 32 levels. Recursion of nested include files is not permitted.

There is, due to a system-wide restriction, a limit of 256 segments per compilation, or the main source segment and 255 %include statements. In order to ensure the proper working of the debugging facilities, multiple uses of a single include file in different %include statements count as separate segments.

There is no restriction on the relationship between include file boundaries and subprogram boundaries. A single subroutine may use more than one %include statement, and a single include file may contain more than one subprogram. An include file may contain the last part of one subprogram and the beginning of another.

Language Options

Multics FORTRAN is being brought into conformance with the 1977 ANSI standard for FORTRAN (FORTRAN 77). As this process is carried out, certain incompatible changes to the language must be introduced. To reduce the impact of these changes, two options are available for controlling the interpretation of constructs whose meanings are different under FORTRAN 77.

Under the ansi66 option, the "old" interpretation of incompatible constructs is used. The interpretation corresponds to the 1966 ANSI standard for FORTRAN with many extensions specific to Multics FORTRAN.

Under the ansi77 option, the new interpretation of incompatible constructs is used. The interpretation corresponds to the rules of FORTRAN 77.

The majority of language constructs, including many features of FORTRAN 77 and extensions specific to Multics FORTRAN have identical interpretations under these two options. Only where FORTRAN 77 requires a different interpretation of some construct already present in the ansi66 Multics FORTRAN language do these options matter.

Appendix B contains a concise list of constructs which differ between the ansi66 and ansi77 options. This list is not yet complete, and as Multics FORTRAN approaches full conformance to FORTRAN 77 additional incompatible changes will be placed under control of the ansi77 option.

These two options may be specified by control arguments on the command line (-ansi66, -ansi77) and as keywords in %options and %global statements (ansi66, ansi77). Only one of the two may be in effect for any one program unit; ansi66 is presently the default.

FORTRAN'S Compilation Options

The following tabs lists all of the options available with the FORTRAN compiler and indicates how each can be specified.

(AGO) ansi66	(A) long	(A) non_relocatable
(AGO) ansi77	(A) long_profile	(A) optimize
(AGO) auto	(A) map	(A) profile
(AG) auto_zero	(AG) no_auto_zero	(A) relocatable
(A) brief	(A) no_check	(AGO) round
(A) brief_table	(AGO) no_check_multiply	(A) safe_optimize
(AGO) card	(AGO) no_fold	(A) severityN
(A) check	(AG) no_large_array	(AGO) static
(AGO) check_multiply	(A) no_line_numbers	(AGO) stringrange
(AGO) default_full	(A) no_map	(AGO) subscriptrange
(AGO) default_safe	(A) no_optimize	(A) table
(AGO) fold	(AGO) no_stringrange	(A) time
(AGO) free	(AGO) no_subscriptrange	(A) time_ot
(A) full_optimize	(A) no_table	(AGO) truncate
(AG) large_array	(A) no_version	(A) version
(A) line_numbers	(AG) no_very_large_array	(AG) very_large_array
(A) list	(AG) no_vla_parm	(AG) vla_parm

Key: A = control argument, G = %global, O = %options

The %global Statement

The %global statement makes it possible to specify in the program itself the options with which a source segment must be compiled, eliminating the need to specify certain control arguments at compilation time. The %global statement has the form:

%global <options>;

The options are specified by the keyword descriptors given below. Multiple keywords may be given with the statement, separated by commas. The list is terminated by a semicolon.

%global statements can be overridden by the fortran command's control arguments. A warning message is printed when a %global statement is overridden. Following is a list of keywords, with brief explanations, for the %global statement:

- ansi66
  specifies that the program is to be interpreted according to the 1966 FORTRAN standard (or Multics specific extensions to it) in situations where the 1977 standard is incompatible.

- ansi77
  specifies that the program is to be interpreted according to the 1977 FORTRAN standard in situations where it is incompatible with the 1966 standard or Multics specific extensions to it.

- card
  specifies that the source segment is in card-image format and that uppercase letters are to be interpreted as lowercase outside of character string constants. Conflicts with free.

- check_multiply
  checks single-precision overflows in integer multiplications. Conflicts with no_check_multiply. This is the default unless optimization is requested.

- no_check_multiply
  inhibits checking for single-precision overflows in integer multiplications.

- free
  specifies that the program is in free-form format and that uppercase and lowercase characters are distinct. Conflicts with card.

- default_full
  sets the default optimization to "full_optimize." Conflicts with default_safe. (This is assumed if no default option is specified.)

- default_safe
  sets the default optimization to "safe_optimize." Conflicts with default_full.

- fold
  specifies that uppercase letters are to be interpreted as lowercase when they are not part of a character string constant. Conflicts with no_fold.

- no_fold
  specifies that uppercase letters are not to be mapped into lowercase form. Conflicts with fold.

- stringrange
  produces additional code to allow substring range checking to be performed at run time. Conflicts with nostrg. Ignored if -optimize or -safe_optimize is given on the command line.

- no_stringrange
inhibits the production of code to allow substring range checking. Conflicts with strg.

- subscriptrange
produces additional code to allow subscriptrange to be checked at run time. Conflicts with nosubrg. Ignored if -optimize or -safe_optimize is given on the command line.

- no_subscriptrange
inhibits the production of code to allow subscriptrange checking (i.e., by the -subscriptrange control argument). Conflicts with subrg.

- static
all local variables are allocated static storage so their values will be retained between invocations. This option has the effect of a generalized save statement. (See Section 6.) Conflicts with auto. Ignored in any program unit in which a save or automatic statement appears.

- auto
all local variables are allocated automatic storage in the stack frame. Conflicts with static. Ignored in any program unit in which a save or automatic statement appears. Note that since common variables are not local variables, this keyword has no meaning for block data subprograms.

- round
specifies that intermediate and final results of real and double-precision calculations are to be rounded before they are stored. Conflicts with truncate. This is the default.

- truncate
specifies that intermediate and final results of real and double_precision calculations are to be truncated before they are stored. Conflicts with round.

- auto_zero
automatic storage for the program must be initialized to zero when allocated. Conflicts with no_auto_zero. This is the default.

- no_auto_zero
automatic storage for the program need not be initialized to zero when allocated. Conflicts with auto_zero.

- large_array
specifies that the compiler is to take all arrays in static and automatic and collect them for Large Array processing. This permits a very large number of arrays which may each be up to a full segment in length. Without this option, the aggregate stack frame size is limited to 62000 words, and the combined linkage section is limited to 128K. (Note the aggregate size of static for binding is 16K.) Conflicts with no_large_array and very_large_array.

- no_large_array
specifies that large array support is not needed. Conflicts with large_array.

- very_large_array
specifies that the size of individual arrays may exceed a segment in length. The present limit of individual arrays under this option is 2**24 words. A large number of such arrays may exist, up to the limit of segment numbers for the process. This automatically sets the mode of array sub-program parameters to VLA, which is a superset of normal hardware addressing. The vla option also sets the la option and provides Large Arrays and Very Large Arrays. Conflicts with no_very_large_array and large_array.

- no_very_large_array
  specifies that very large array is not needed. Conflicts with very_large_arrays.

- vla_parm
  specifies that the size of parameters passed may exceed a segment in length and that vla addressing must be used for parameters. It also permits the declaration of arrays which exceed a segment in length, without placing smaller arrays into Large Array storage. This is useful for the compilation of packages because it descreases startup overhead. Conflicts with no_vla_parm.

- no_vla_parm
  specifies no very large array parameters. Conflicts with vla_parm.

The %global statement, when it is used, must precede all other statements in the source segment, since it applies to all the program units in any one compilation. It cannot have a statement label, cannot be continued, and must appear alone on a line. More than one %global statement may be used. If conflicting options are specified in multiple %global statements, only the last specified has any effect. Control arguments specified with the fortran command are overridden by any conflicting keywords in the %global statement. Options duplicated by a control argument and a %global keyword will take effect as if there were no duplication.

The %options Statement

The %options statement makes it possible to specify in the program the options with which a particular program unit is to be compiled, thereby allowing program units that require different options to be compiled together in one compiler run. The %options statement has the following form:

```
%options <options>;
```

The keywords are the same as for the %global statement, described above, except that auto_zero, no_auto_zero, large_array, no_large_array, very_large_array, no_very_large_array, vla_parm and no_vla_parm cannot be given.

The %options statement affects only the immediately following program unit, and overrides standard defaults, options specified in a %global statement, and control arguments specified when the compiler is invoked.

The %options statement cannot have a statement label, cannot be continued, and must appear alone on a line. More than one %options statement may be used in the same program unit (main program or subprogram). The %options statement must precede all other statements in the program unit, including subroutine or function statements, if any, except the %global statement at the beginning of the segment, if %global is used.

SECTION 2

DATA MODES

Multics FORTRAN supports six data types, called modes: integer, real, double precision, complex, logical, and character. All six modes can be written as constants or literals within the text of a program unit, and can be assigned as the values of variables or returned as the values of functions.

DATA MODES

Constants

A constant is a fixed, invariant quantity. Multics FORTRAN provides four main kinds of constants: arithmetic, logical, character-string, and octal.

Arithmetic constants are integer, real, double precision, or complex numbers; logical constants are .true. or .false.; character constants are alphabetic and/or numeric characters; and octal constants are base eight numbers.

The form of the constant determines both its value and mode. The parameter statement makes it possible to give a constant a symbolic name (see "Named Constants" below). Blanks in a constant have no significance, except in a character-string constant.

An unsigned constant is a constant without a leading sign. A signed constant is a constant with a leading plus or minus sign. An optionally signed constant is either signed or unsigned.

Integer Mode

INTERNAL REPRESENTATION

An integer value occupies one 36-bit word of storage, in the form of a (2's complement) binary integer.

RANGE

Integer values range from -34,359,738,368 to +34,359,738,367.

CONSTANTS

An integer constant is an optionally signed number without a decimal point. An integer constant can have up to 11 decimal digits. Any 10-digit decimal integer lies within the permitted range of integer values.

Examples:

Valid	Invalid
+5	3,417 (comma not allowed)
100	456.1 (decimal point not allowed)
-25	106143614218 (out of range -2^{35} to $2^{35}-1$)
2314132567	

Real Mode

INTERNAL REPRESENTATION

A real value occupies one 36-bit word of storage, in the form of a binary single-precision floating-point number.

The internal representation has two parts: a mantissa or fractional part, and an exponent. (These are represented in the following diagram by the EXP and MANTISSA fields.)

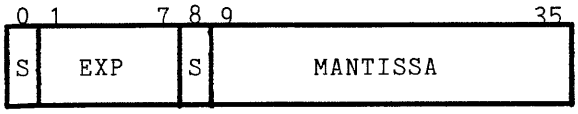


Figure 2-1. Format of Real Data

The value of such a real number is:

MANTISSA * 2**EXP

Bit 0 is the sign of the exponent.
Bit 8 is the sign of the number.

RANGE

The largest magnitude (absolute value) (ignoring sign) that can be represented in Multics FORTRAN is $1.7014118360\ 10^{-38}$ that results in a larger magnitude will cause the overflow condition to be signalled as the result of a hardware fault.

The smallest nonzero magnitude that can be represented in Multics FORTRAN is $1.4693679385278 * 10^{-38}$. Any arithmetic operation that results in a smaller nonzero magnitude will cause the underflow condition to be signalled as the result of a hardware fault.

CONSTANTS

A real constant is an optionally signed decimal number of up to eight digits that contains either a decimal point, an exponent, or both. In the absence of a decimal point, an exponent must appear, and the decimal point is assumed to be immediately to the right of the digit preceding the exponent. An exponent is written as a lowercase letter e followed immediately by an optionally signed integer constant, adhering to the rules given above for integer constants.

The value of a real constant containing an exponent is the number multiplied by a power of 10, where the power of 10 is given in the exponent field. Hence, "2000." is represented as "2.0e3". A decimal number containing a decimal point but no exponent is considered to be a real (single-precision) constant, unless there are more than eight digits in the number, in which case the number is assumed to be a double-precision constant. If the number of digits is greater than eight, a warning is printed out at compile time telling the user that a double-precision constant has been created.

Examples:

Valid	Invalid
1.0	1 (no decimal point or exponent)
1.5	2e-40 (exponent is too small)
-25.3	3d0 (invalid single-precision exponentiation indicator, 3)
7.0e3	2-e40 (this is really an expression)
1.e0	
+5.21e-5	
-.3333e-10	
5.e-15	
.0	
0.e-4	

Double-Precision Mode

INTERNAL REPRESENTATION

A double-precision value occupies a 72-bit double word of storage on an even boundary, in the form of a binary double-precision floating-point number. The internal representation of a double precision value is the same as that of a real-value, except that the mantissa has a second 36-bit word of storage, providing for increased precision.

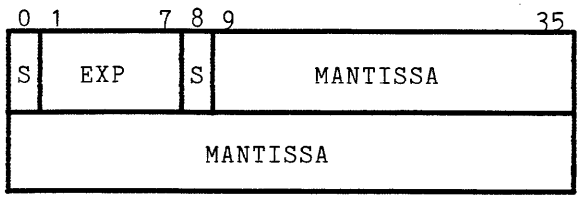


Figure 2-2. Format of Double-Precision Data

RANGE

The range of double-precision values is the same as for real values.

CONSTANTS

Double-precision constants are identical to real constants, except that either

- the character d is used as the exponentiation indicator instead of the character e, or
- the number of significant digits in the constant is greater than 8 (it may be as many as 19).

In the absence of a decimal point, an exponent must appear, and the decimal point is assumed to be immediately to the right of the rightmost digit before the exponent. If the exponent is omitted, the decimal point must appear and the number must contain at least nine decimal digits. A warning message is printed, indicating that the mode of the constant is being interpreted as double precision. To suppress the warning message, always include a double precision exponent in double precision constants.

Examples:

Valid	Invalid
25d5	1 (no exponent or decimal point)
-1.5d-03	467. (not enough digits)
1234.567890	1d+40 (exponent too large)

Complex Mode

INTERNAL REPRESENTATION

A complex value occupies two 36-bit words of storage on an even-word boundary, in the form of an ordered pair of real values, the first word containing the real part and the second word the imaginary part.

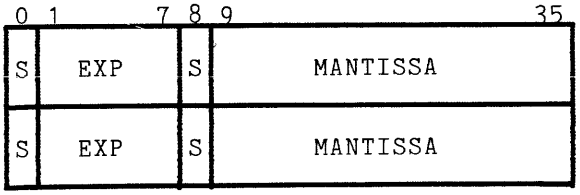


Figure 2-3. Format of Complex Data

RANGE

The range of values for each element of the ordered pair is the same as for real values.

CONSTANTS

A complex constant is a pair of integer or real constants, or one of each, separated by a comma and enclosed in parentheses. Double precision constants are not acceptable as either the real or the imaginary part of a complex constant.

Examples:

Valid	Invalid
(1.5, .3)	(1d3,123456789) (wrong exponentiation indicator)
(-5., .0)	(1e3,1234567890) (too many digits in external part)
(25.3, -5.2e7)	
(1, 5.2)	
(3.7, 0)	
(14.e20, 23.54321)	
(1, 2.0e5)	

Logical Mode

INTERNAL REPRESENTATION

Logical data occupies one 36-bit word of storage. The value true is represented by the octal value 400000000000. The value false is represented by the octal value 000000000000.

RANGE

Logical data may have the values .true. or .false.

CONSTANTS

Logical values are represented in the source program by the constants .true. and .false.

Examples:

Valid	Invalid
.true.	true (periods required)
.false.	

Character Mode

Character data represents an ordered sequence of ASCII characters stored in from 1 to 64 words. Each word, except the last, contains four characters; the last word may contain from one to four characters. The maximum number of characters allowed in a single character string is 256.

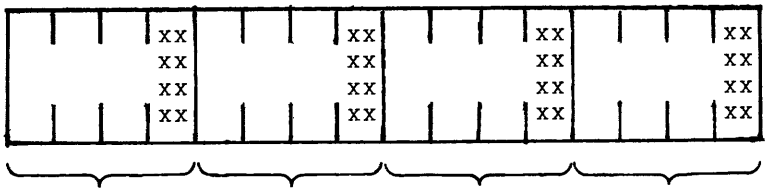
In the ansi66 implementation, all character variables and array elements are stored as aligned character strings, that is, starting on a word boundary in the computer memory. For the purposes of storage layout, array elements are padded to a multiple of 4 characters. These pad characters are not part of any character data but exists only to align the array elements at a word boundary.

In the ansi77 implementation, character variables may be stored as unaligned character strings, that is, each array element immediately follows the preceding element with no intervening padding and thus may begin at character positions that are not word boundaries.

For example, in the ansi66 implementation an array declared as

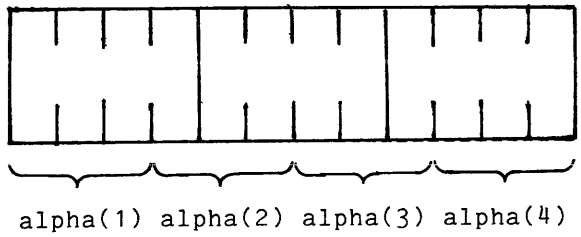
```
character*3 alpha (4)
```

is stored as:



Here, each of the 4 words stores an array element of 3 characters. The last byte of each word is unused so that the next element can begin at a word boundary.

In the ansi77 implementation this array is stored as:



Scalar character variables which appear in common blocks or equivalence groups may begin off word boundaries if the ansi77 option is in effect.

CONSTANTS

Character data can be represented in the text of a program by character-string constants. A character-string constant can be expressed in either of the two forms described below.

One form of character-string constant consists of a sequence of one or more contiguous ASCII characters enclosed either in quotation marks or apostrophes. When quotation marks are used to delimit it, any quotation marks in the actual string must be doubled. Similarly, any apostrophes embedded in a character string delimited by apostrophes must be doubled.

Examples:

```
"This is a character-string constant"
'This is a "character-string" constant'
"This string contains "" and '."
```

The second form used to express a character-string constant begins with an unsigned integer constant indicating the length of the string. This is followed immediately by the letter h followed by a string of ASCII characters whose length is equal to the value of the integer constant. There are no restrictions on the characters that can appear as part of the string.

Examples:

```
35hThis is a character-string constant
29hThis string contains " and '.
```

The FORTRAN 77 Standard does not include this second form. Multics, however, at present supports Hollerith character-string constants under both the ansi66 and ansi77 options.

A character-string constant can appear in the following contexts: on the right-hand side of an assignment statement; as an initial value in a data statement; as an argument of a call statement or a function reference; as a format specification in an input/output statement; as an operand of a relational operator; as an operand of the concatenation operator; and in a parameter statement.

OCTAL CONSTANTS

An octal constant is a numeric string preceded by a lowercase letter o. The numeric string can consist of from 1 to 24 octal (base eight) digits. Octal constants may be written only as initial values in a data statement.

When an octal constant is stored as the initial value of a variable, the bit value represented by the constant is right justified within the storage area allocated for the variable. Integer and real variables contain 12 octal digits; double precision and complex variables contain 24. If fewer digits are expressed, the constant will be padded on the left with zeroes; if more are expressed, excess left-hand digits will be truncated. See the various data modes in this section for the particulars.

Examples:

```
o7777
o54177
o1
```

A named constant can appear in most of the same contexts where a constant of the same mode can appear. An integer named constant cannot be used as either part of a complex constant, although a real named constant is allowed. A named constant cannot specify a statement label, appear within a format specification, or be used to where the syntax would be ambiguous. For example:

```
parameter (len=3)
character*len a
character b*len
```

The declaration of a is in error; the declaration of b is allowed. See the parameter statement for a description of how to declare named constants.

SECTION 3

EXPRESSIONS

In FORTRAN, the term expression applies to any language construct that represents a value, including arithmetic, character, relational, and logical expressions. The simplest form of expression is a constant (described in Section 2), a variable reference, array element reference, or function reference, described below. More complicated expressions combine one or more operands with operators and parentheses.

NAMES

The programmer can construct symbolic names to identify or represent the following program entities:

main program
variables
subprogram entries
arrays
statement functions
common blocks
namelists
named constants

Names are constructed using any alphabetic or numeric characters and the characters dollar sign ($) and underscore (_). The first character must be alphabetic. Only one $ is permitted in a name. An alphabetic character is any one of the lowercase or uppercase letters of the alphabet. A numeric character is any one of the decimal digits 0-9.

Normally, the Multics system makes a distinction between uppercase and lowercase (e.g., A10 is not the same name as a10). If the user selects the -fold control argument in compiling, however, all uppercase letters are converted to lowercase (except within character-string constants). The -fold control argument is implied by card-image input (i.e., -card control argument).

Programmer-constructed names for variables, arrays, and namelists can be from 1 to 256 characters long. The first nine characters of a variable name cannot be "parameter."

The names "main_" and "symbol_table" are used internally by the Multics system and cannot be used as entry point names.

Names used to identify generic functions, built-in functions, or keywords are defined by the language. All such names consist of lowercase alphabetic and/or numeric characters only. The programmer is free to use these names for his own purposes, but such multiple use is generally considered to be a poor programming practice. In general in the Multics system, because of dynamic linking, name duplications should be avoided. (See the FORTRAN Users' Guide for fuller discussion of these and related issues.)

VARIABLES

A FORTRAN variable is an entity that has both a symbolic name and a mode. A variable name must conform to the rules given under "Names" above. The mode of a variable corresponds to the mode of the datum stored in it. The first letter of the symbolic name implicitly defines the mode of the variable. The default implicit mode associated with the letters i, j, k, l, m, and n is integer; that associated with all the other letters in the alphabet is real. The implicit mode associated with a letter can be modified by the implicit statement. The mode of a specified variable can be changed from the implicit mode associated with its name to some other mode by declaring it in an explicit mode statement. An explicit mode statement overrides the implicit statement. Section 6 provides detailed information on the declaration of variable modes.

At any point in the execution of a FORTRAN program, a variable is either defined or undefined. Before a variable has been assigned a value, its contents are undefined, and the variable may not be referenced other than to assign it a value.

ARRAYS

An array is an n-dimensional ordered sequence of values (all of the same mode) that is given a name conforming to the rules given under "Names" above. The values are called array elements.

The dimensions and bounds of an array are specified by an array declarator in a dimension statement, a common statement, an automatic statement, or a mode statement. An array declarator has the form:

```
a(b[,b]...)
```

where a is the array name and each b gives the maximum value for one dimension of the array. The maximum number of dimensions permitted is seven. All references to an array element must contain the number of subscripts provided in the declaration.

SUBSCRIPTS

A subscript is an integer quantity (an integer constant or constant expression), or a list of integer quantities separated by commas, that is associated with the array name to identify one element of the array. It may be a negative value. The number of quantities in any subscript must be the same as the number of dimensions of the corresponding array. A subscript is enclosed in parentheses and immediately follows the array name.

The number of elements contained in an array may be declared by specifying in an array declaration the number of dimensions and the extent of each dimension in the array.

The form of each b is:

$[b_1:]b_2$

where: b_1 is the lower dimension bound. It is optional.
b_2 is the upper dimension bound.

If the lower bound is omitted, the colon must also be omitted, and the value of the lower bound is assumed to be 1.

Each b is one of the following:

an integer constant

an integer expression involving constants, named constants defined in earlier parameter statements, and the operators +, -, *, /, and **.

a simple variable

If b is a variable, it must either be in common storage or be provided for with a dummy parameter, and then a also must be provided for with a dummy parameter of the subprogram in which the array is declared.

The value of a subscript may be negative, zero, or positive, but it must be greater than or equal to the lower bound (b_1) and less than or equal to the upper bound (b_2). An array subscript is normally limited to a range which addresses a single segment of storage or less, -262143 through 262143 for integers and single precision real numbers, -131071 through 131071 for double precision and complex numbers. It is further restricted for character arrays by the length of the array element character string.

When Very Large Array compilation is used, numeric arrays can exceed a segment in length. In this case, the subscript limits for numeric data types are increased to address up to 2**24 words, -16,777,215 through 16,777,215 for single precision and integers, -8,388,607 through 8,388,607 for double precision and complex data types.

Assumed-Size Arrays

An array may be declared with an asterisk used as the upper bound of the last dimension. These arrays are called assumed-size arrays. In a subprogram, an assumed-size array, like any other array, must be provided for with a dummy parameter.

The omitted upper bound is assumed to be greater than or equal to the corresponding lower bound, but its exact value depends on the actual argument used when the array is referenced. Assumed-size arrays cannot be used in contexts where the value of the last upper bound is required. Those contexts, which treat the entire array as a unit, are:

the input/output list of any input/output statement

the argument list of a reference to an external subprogram declared with the (descriptors) attribute

the storage specification of an encode or decode statement

the format specification of a formatted input/output statement.

Note that assumed-size arrays may be passed to subprograms which do not require Multics argument descriptors.

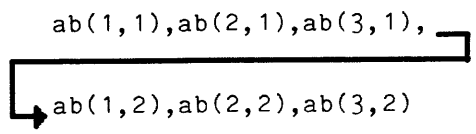
Subscript range checking is always disabled for subscripts in the last dimension of an assumed-size array.

STORAGE ARRAYS

The elements of an array are stored in contiguous storage in column-major order. If the elements are accessed in the order in which they are stored, the leftmost subscript varies most rapidly. Given the following dimension statement:

```
dimension ab(3,2)
```

storage is allocated as follows:



The FORTRAN language defines very few operations on arrays as a whole: they may be transmitted by input/output statements, they may be passed as arguments to subprograms, they may be used as format specifiers in any formatted input/output statement, and they may be used to identify storage in encode and decode statements.

In the Multics storage hierarchy, segments are limited to 255K (255*1024) words in length, which is the limit of vfile_ IO. Thus it is not possible to do IO of a Very Large Array, or any combination of normal arrays which exceeds this limit, with a single binary IO record. This means that it is not always possible to use an entire array as the list item for a binary IO read or write statement, since a Very Large Array will, by definition, exceed this limit.

ARRAY ELEMENTS

An individual value in an array, called an array element, is referred to by the name of the array followed by a parenthesized list of subscripts corresponding to the number of dimensions declared for the array, as shown below:

```
a(s[,s]...)
```

where a is the array name and each s is an arithmetic expression whose value is truncated to integer. There must be one s for each declared dimension of the array.

Examples:

```
a(5)    is a reference to the fifth element of the
        array a
b(i,j)  is a reference to the jth element in the ith row
        of the array b
c(k+3)  is a reference to the k+3 element of the array c
```

CHARACTER SUBSTRINGS

A character substring is character data that is a part of a character variable or array element. It is represented by a substring name and may appear in any place where a character array element or variable is permitted. The form of a substring name is:

```
variable_name(exp1:exp2)
```

or:

```
array_name(subscript1,...,subscriptn)(exp1:exp2)
```

where variable_name is the name of a simple character variable and array_name is the name of a character array, (subscript1,...,subscriptn) is the array subscript information, and exp1 and exp2 are both integer expressions. The expressions exp1 and exp2 are known as the substring expressions. The expression exp1 specifies the leftmost character position of the substring within the string, and exp2 specifies the rightmost position. If exp1 is omitted, a value of 1 is assumed. If exp2 is omitted, the length of the string upon which the substring is being defined is assumed. Thus a substring expression of the form string(:) identifies the entire character string. The length of the substring is equal to exp2-exp1+1.

The substring expressions may be any integer expressions. Array references and function references are permitted. Note that the evaluation of a function within a substring expression must not alter the value of any other expression within the substring reference.

The substring expressions are subject to the constraint:

```
1 <= exp1 <= exp2 <= length of string
```

Character substrings can be used only when the program is compiled under the ansi77 option.

FUNCTION REFERENCES

A function reference consists of a generic function name, a built-in function name, a statement function name, or a function subprogram entry name followed by a parenthesized argument list containing zero or more arguments. The semantics of a function reference are identical to those of the call statement described in Section 4 except that a function cannot alter the value of any other variable used in the same statement that contains the function reference, and statement labels cannot be passed as arguments. Reference to a zero argument function consists of the function name, a left parenthesis, and a right parenthesis. These parentheses must be included in both the actual and dummy argument lists of such functions; they cannot be omitted. For example, the statement:

```
area () = 3.14 * r **2
```

defines a statement function "area" with zero arguments. The statement:

```
integer function previous ()
```

begins a function subprogram of zero arguments.

Function references are evaluated within expressions at the point where their value is required and do not affect the order of operator evaluation. It should be noted that if multiple calls to the same function (using identical arguments) occur in a single statement, the compiler may generate code to call the function only once, and save the returned value to substitute it for the other calls.

MODE CONVERSION

Each FORTRAN operator requires that its operands be of specific modes. In certain cases an operand that does not conform to the required mode is converted to a temporary value that has the required mode. Mode conversion is also performed during the execution of an assignment statement if the mode of the left side of the assignment operator differs from the mode of the expression on the right side. The rules for specific mode conversion are as follows.

To a Mode of Higher Rank

INTEGER TO REAL

Any integer value can be converted to a real value although some values will lose precision.

The exponent of the real value is calculated in an appropriate fashion. The mantissa of the real value is 27 bits of the integer value, beginning with the first nonzero bit for nonnegative values, and beginning with the first zero bit otherwise. An integer value is represented exactly as long as its magnitude is less than 2^{27}. Otherwise, it is truncated. The leftmost bit of the integer that is not equal to the sign bit (0 or 1) is placed next to the sign bit of the mantissa and the exponent is adjusted accordingly (this operation is called normalization).

The round/truncate option is ignored.

INTEGER TO DOUBLE PRECISION

An integer value is represented exactly. The mantissa is normalized as explained above in "integer to real."

The round/truncate option is ignored.

INTEGER TO COMPLEX

The integer value becomes the complex value by conversion of the integer to a real value, which becomes the real part of the complex value. The imaginary part of the complex value is set to zero.

The round/truncate option is ignored.

REAL TO DOUBLE PRECISION

The exponent of the double precision value is the same as the exponent of the real value. The mantissa of the double precision value is the mantissa of the real value (27 bits) followed by 36 bits of zero.

Any real value can be converted to a double precision value with no loss of precision.

The round/truncate option is ignored.

REAL TO COMPLEX

The real value becomes the real part of the complex value. The imaginary part of the complex value is zero.

Any real value can be converted to a complex value with no loss of precision.

DOUBLE PRECISION TO COMPLEX

The exponent of the real part of the complex value is the same as the exponent of the double precision value. The mantissa of the complex value is determined by rounding the mantissa of the double precision value to 27 bits (if the round option is specified), or by truncating the mantissa of the double precision value to 27 bits (otherwise). The actual rounding or truncation takes place when the value is stored in main memory. The imaginary part of the complex value is zero.

Any double precision value can be converted to a complex value, although some values will lose precision.

To Mode of a Lower Rank

REAL TO INTEGER

If the round option is specified, and the real value is the result of evaluating an expression other than a variable or function reference, the real value is rounded to 27 bits of mantissa. Otherwise, the extra bits are truncated.

Real values whose absolute value is $\geq 2^{35}$ cannot be converted to integer. The result of an attempt to perform such conversions is undefined.

The real value is converted to an integer by shifting the mantissa until the exponent goes to zero. All fractional digits are truncated.

Any real value within the permitted range is converted with no loss of precision (except for the fractional part).

DOUBLE PRECISION TO REAL

The exponent of the real value is the same as the exponent of the double precision value. The mantissa of the real value is determined by rounding the mantissa of the double precision value to 27 bits (if the round option is specified), or by truncating the mantissa of the double precision value to 27 bits (otherwise). The actual rounding or truncation takes place when the value is stored in main memory.

Any double precision value can be converted to real, although some values will lose precision.

DOUBLE PRECISION TO INTEGER

If the round option is specified, and the double precision value is the result of evaluating an expression other than a variable or function reference, the double precision value is rounded to 63 bits of mantissa. Otherwise, the extra bits are truncated.

The double precision value is converted to integer by shifting the mantissa until the exponent goes to zero. All fractional digits are truncated.

Any double precision value within the permitted range is converted with no loss of precision (except for the fractional part).

COMPLEX TO DOUBLE PRECISION

The real part of the complex value is itself a real value, and is converted to the double precision value as described above under "Real to Double Precision." The imaginary part of the complex value is ignored.

The round/truncate option is ignored.

COMPLEX TO REAL

The real part of the complex value is assigned unchanged to the real value. The imaginary part of the complex value is ignored.

COMPLEX TO INTEGER

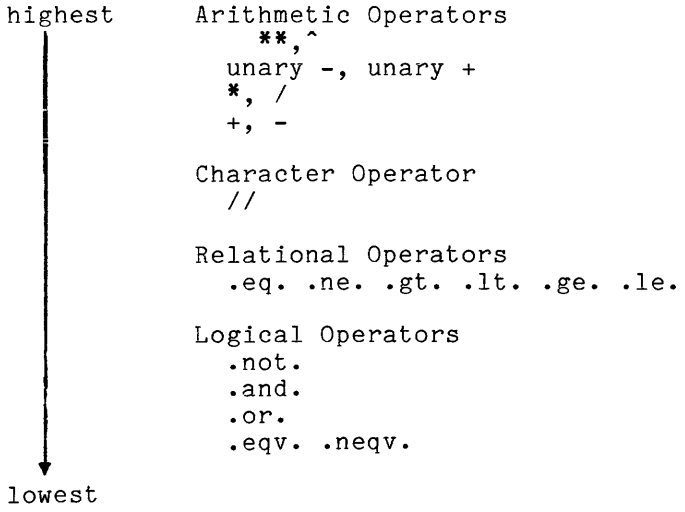
The real part of the complex value is converted to integer by shifting the mantissa until the exponent goes to zero. All fractional digits are truncated. The imaginary part of the complex value is ignored. If the round option is specified, and the complex value is the result of evaluating an expression other than a variable or function reference, the real part of the complex value is rounded to 27 bits of mantissa. Otherwise, the extra bits are truncated.

If the absolute value of the real part of the complex value is $\geq 2^{35}$, it cannot be converted to integer. The result of an attempt to perform such a conversion is undefined.

FORTRAN OPERATORS

There are four types of FORTRAN operators: arithmetic, relational, logical, and character. Operators can be unary (requiring a single operand) or binary (requiring two operands).

Precedence determines the order in which an operator is evaluated in an expression. Adjacent operators that have the same precedence, except for **, are evaluated from left to right; adjacent occurrences of ** are evaluated from right to left. In the expression a+b/c, for example, the division operation has greater precedence than the addition operation so that b/c is evaluated first and its result added to a. FORTRAN operators are listed below in order of precedence beginning at the highest level (those operators that are acted upon first) to the lowest.



Normal precedence can be overridden by the use of parentheses to separate a portion of an expression. For example, the expression a*b+c is evaluated by multiplying a times b and then adding c to the result. The expression a*(b+c) is evaluated by multiplying a times the total of b+c. Other examples of precedence are shown below.

Expression	Evaluation
a+b+c*d*e**f**g	(a+b)+((c*d)*((e**(f**g))))
a.eq.b.or.c.lt.d	(a.eq.b).or.(c.lt.d)
a.and.b.or.c	(a.and.b).or.c

ARITHMETIC OPERATORS

The arithmetic operators and their indicated meanings are:

+ -	plus minus (unary)
+ -	add subtract (binary)
* /	multiply divide
** or ^	exponentiation

The operands of an arithmetic operator must have values whose modes are arithmetic. Arithmetic modes have a rank that determines the mode of the result of arithmetic operations. If the modes of the two operands differ, the operand of the lower mode is converted to a temporary value whose mode is that of the higher operand, except in the case of raising an operand to an integer power, for which there is no conversion. The result of the operation has the higher of the two modes.

Evaluation of real, double precision, and complex expressions takes place in machine registers that may contain more bits of mantissa than is available in memory. Therefore, if intermediate results of computations involving such expressions must be stored in memory, the mantissas are rounded or truncated as specified by the round or truncate options.

All intermediate and final results of arithmetic operations must be within the range of the data type in order for the final result to the defined.

The rank of arithmetic modes (from highest to lowest) is:

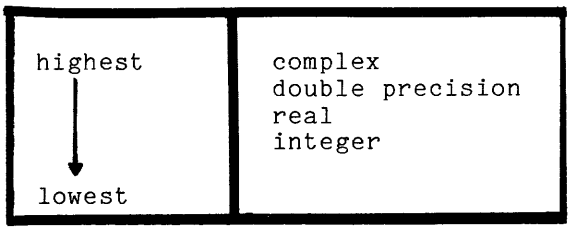


Table 3-1 defines the mode of the result of the operation for all arithmetic operators.

Table 3-1. Operands and Mode of Results

Left Operand	Right Operand			
	Integer	Real	Double Precision	Complex
Integer	Integer	Real	Double Precision	Complex
Real	Real	Real	Double Precision	Complex
Double Precision	Double Precision	Double Precision	Double Precision	Complex
Complex	Complex	Complex	Complex	Complex

CHARACTER OPERATOR

The only character operator is:

// concatenation

This operation can be done only when the program is compiled under the ansi77 option.

The concatenation operator is a binary operator. Both its operands must be character data; no data conversions are defined for it. The result is formed by concatenation of the two operands, and it also is produced as character data. In the case of multiple concatenation, the operands are evaluated from left to right, unless the order of evaluation has been altered by the use of parentheses.

It should be noted that the compiler is only required to evaluate as much of a character expression as is necessary to provide a result of the necessary length. To take a specific example, in a program fragment such as:

```
character*4 a,b
character*6 c, char_function
c = a // b // char_function()
```

the compiler need not call the function named char_function because the result of concatenating a and b provides an 8 character result, and only 6 characters are needed to completely set the character variable c.

RELATIONAL OPERATORS

Relational operators produce logical results--true if the relation is satisfied, .false. otherwise. The relational operators and their indicated meanings are:

.eq.	equal to
.ne.	not equal to
.gt.	greater than
.lt.	less than
.ge.	greater than or equal to
.le.	less than or equal to

The operands of relational operators must be character-string values or must be arithmetic values of integer, real, or double-precision modes. In addition, the .eq. and .ne. operators may have operands both of which are complex or both of which are logical.

If the mode of one operand is not character, and the other is a character-string constant, then the internal representations of the two are compared. If the mode of one operand is integer, real, or logical, then the character-string constant must be from one to four characters long. (If less than four characters long, it is padded on the right with blanks. For example, "i" is converted to "i ". If it is greater than four characters long, a warning is printed and only the first four characters of the character-string constant are used.) If the mode of one operand is double precision or complex, the character-string constant must be from one to eight characters long. (If less than eight characters long, it is padded on the right with blanks. If it is greater than eight characters long, a warning is printed and only the first eight characters of the character-string constant are used.)

LOGICAL OPERATORS

The operands of a logical operator must have logical values. The result of a logical operator is a logical value defined by the following:

a.and.b has the value true if both a and b are true. It has the value false if either a or b is false.

a.or.b has the value true if either a or b is true. It has the value false if both a and b are false.

.not.a has the value true if a is false. It has the value false if a is true.

a.eqv.b has the value true if a and b are either both true or both false. It has the value false if either a or b is true and the other false.

a.neqv.b has the value true if either a or b is true and the other false. It has the value false if a and b are either both true or both false.

EXECUTABLE STATEMENTS

The following statements are executable in Multics FORTRAN:

1. arithmetic, logical, and character assignment statements
2. statement label (assign) statements
3. arithmetic if and logical if statements
4. unconditional go to, computed go to, and assigned go to statements
5. do statement
6. continue statement
7. call and return statements
8. stop and pause statements
9. read, write, input, and print statements
10. open and close statements
11. inquire statement
12. rewind, backspace, and endfile statements
13. end statement

ASSIGNMENT STATEMENT

Syntax:

```
a=b
```

where a is a variable or an array element name, or the name of the major entry of the function subprogram in which this statement occurs, and b is an expression.

Semantics:

The expression b is evaluated and the resultant value is substituted for the value currently assigned to a.

If b is an arithmetic expression, its value is converted to the mode of a. The mode of a must be one of the arithmetic modes.

If b is a logical expression, the mode of a must be logical.

If the mode of a is character, b must be a character-string expression. If the length of b exceeds the length of a, only the leftmost characters of b are assigned. If b is shorter than a, it is padded on the right with blanks.

If b is a character-string constant and the mode of a is not character, the internal representation of b is assigned without conversion to a. If the mode of a is integer, real, or logical, then the character-string constant b must be from one to four characters long. (If less than four characters long, it is padded on the right with blanks. For example, "i" is converted to "i ". If b is greater than four characters long, a warning is printed and only the first four characters of b are assigned to a.) If the mode of a is double precision or complex, b must be from one to eight characters long. (If less than eight characters long, it is padded on the right with blanks. If b is greater than eight characters long, a warning is printed and only the first eight characters of b are assigned to a.)

Users who store character-string values in real, double-precision, or complex variables are warned that the value of b is rounded before assignment to a if the modes of a and b differ; even if the modes are the same, the value of b is rounded when b is not a variable or function reference and the round option has been specified. To round a character-string value is to destroy it.

The evaluation of real, double precision, and complex expressions is carried out in machine registers that may contain more bits of mantissa than are available in memory. As a result, the value of the expression must be truncated or rounded before assignment to the variable if the expression is not a constant, a variable reference, an array element reference, or a function subprogram reference. The rounding or truncation is controlled by the round and truncate options of the %global and %options statements.

Examples:

```
a(i)=b+c
k=5
i="tag"
```

If a is the name of the major entry of the function subprogram in which the assignment occurs, the value assigned to a is returned to the calling subprogram when the next return statement is executed. This mechanism is the way a FORTRAN function subprogram returns a value to its caller.

Example:

```
c  this function returns 2 * x
        function f(x)
        f=x+x
        return
        end
```

ASSIGN STATEMENT

Syntax:

assign n to i

where n is a statement label and i is a simple variable of integer mode.

Semantics:

A designator identifying the statement labeled n is assigned as the value of the integer variable i. Such a value can be used in an assigned go to statement or a format statement. This statement makes for efficient transfer of control because the statement label used in a particular reference can be varied.

Example:

```
assign 100 to k
```

ARITHMETIC IF STATEMENT

Syntax:

if(e) $[n_1],[n_2],[n_3]$

where e is an integer, real, or double-precision expression; and n_1, n_2, and n_3 are statement labels. Any of the labels (n_1, n_2, or n_3) can be omitted, but their relative positions must be retained by the use of commas.

Semantics:

The expression e is evaluated and control is transferred to the executable statement labeled: n_1 if e is negative, n_2 if e is 0, and n_3 if e is positive. If a label is omitted, the statement following the if is assumed. This statement allows control to be transferred on the basis of the sign of an arithmetic variable.

Example:

```
if(a-b)5,7,8
```

LOGICAL IF STATEMENT

Syntax:

if(e) s

where e is a logical expression and s is any executable FORTRAN statement except a do, logical if, block if, else if, else, end if, or end.

Semantics:

The expression e is evaluated. If it is true, statement s is executed; otherwise, s is not executed. This permits conditional execution of FORTRAN statements. Note that the expression e may or may not be fully evaluated. For example, in the context:

```
if (a .eq. 0 .or. func(x) .eq. y) go to 999
```

if a is in fact 0 the function func may or may not be called.

BLOCK IF STATEMENT

Syntax:

```
if (e) then
```

where e is a logical expression.

Semantics:

The block if statement is used with the end if statement to form a block if structure. A block if structure may contain one or more else if statements as well as a single else statement. It may also be empty. The block if structure allows the grouping of one or more blocks of code which are executed conditionally.

Block if structures can be nested, that is, a block if structure can contain other block if structures within it.

Each statement in a program unit has an if-level associated with it. The if-level of a given statement is the number of block if statements from the beginning of the program unit up to and including that statement minus the number of end if statements from the beginning of the program unit up to but not including that statement. The if-level of every statement must be zero or greater; the if-level of the end statement must be zero.

Execution of a block if statement causes the logical expression e to be evaluated. If the value of e is false, control is transferred to the next else if, else, or end if statement with the same if-level. If the value of e is true, the statements in the if-block are executed. Should control reach the end of the if-block, control is then transferred to the next end if statement with the same if-level as the block if statement.

Transfer of control into an if-block from outside the if-block is prohibited.

If an if-block contains a do statement, the entire range of the do-loop must be contained within the if-block.

A block if statement may have a statement label and can be referenced by another statement.

ELSE STATEMENT

Syntax:

```
else
```

Semantics:

The else statement provides an alternate path of execution for a block if statement or an else if statement. The else statement may appear only within a block if or else if structure, and its if-level must be greater than zero.

The statements following an else statement up to but not including the next end if statement with the same if-level comprise an else-block. An else-block may be empty. An else-block may not contain else if or else statements that have the same if-level as the else statement heading the else-block.

The execution of an else statement has no effect.

Transfer of control into an else-block from outside the else-block is prohibited.

If an else-block contains a do statement, the entire range of the do-loop must be contained within the else-block.

An else statement may have a statement label, but the label may not be referenced by any statement.

ELSE IF STATEMENT

Syntax:

```
else if (e) then
```

where e is a logical expression.

Semantics:

The else if statement combines the functions of the else and block if statements. This statement performs a conditional test and provides an alternate path of execution for a block if or another else if. The else if statement makes it possible to form block if structures with more than one alternative.

The else if statement may appear only within a block if or else if structure and must have an if-level greater than zero.

The statements following an else if statement up to but not including the next else if, else, or end if statement with the same if-level comprise and else-if-block. An else-if-block may be empty.

Execution of an else if statement causes the logical expression e to be evaluated. If the value of e is false, control is transferred to the next else if, else, or end if statement with the same if-level. If the value of e is true, the statements in the else-if-block are executed. Should control reach the end of the else-if-block, control is transferred to the next end if statement with the same if-level as the else if statement.

Transfer of control into an else-if-block from outside the else-if-block is prohibited.

If an else-if-block contains a do statement, the entire range of the do-loop must be contained within the else-if-block.

An else if statement may have a statement label, but the label may not be referenced by any statement.

END IF STATEMENT

Syntax:

end if

Semantics:

For each block if statement there must be a corresponding end if statement. That is, there must be an end if statement with the same if-level for every block if statement. This statement is used to indicate the end of a block-if structure.

Execution of an end if statement has no effect.

UNCONDITIONAL GO TO STATEMENT

Syntax:

go to n

where n is a statement label.

Semantics:

Control is transferred to the executable statement with the label n.

Example:

```
go to 5
```

COMPUTED GO TO STATEMENT

Syntax:

go to (n_1,n_2,...n_m) [,]i

where each n is a statement label and i is an arithmetic expression whose value is converted to an integer.

Semantics:

If i is within the range $1 \leq i \leq m$, control transfers to the statement labeled by the ith label in the list of statement labels. Otherwise, control passes to the statement following the computed go to. This statement permits transfer of control to an almost unlimited number of points, whereas the logical if and arithmetic if statements permit transfer to two and three points.

Example:

```
go to (5,10,17),k
```

ASSIGNED GO TO STATEMENT

Syntax:

go to i [[,] (n[,n]...)]

where n is a statement label and i is a scalar variable of integer mode.

Semantics:

Control is transferred to the statement identified by the statement label value of i. The variable i acquires a statement label value only by appearing in an assign statement in the same program unit (see "Assign Statement" above). If a list of statement labels appears in the statement, the value of i must be one of the labels in the list. Used in conjunction with the assign statement, the assigned go to statement allows transfer of control based on assigned statement labels, and the optional list permits checking of the assigned label.

Example:

```
go to k, (5,10,15)
```

DO STATEMENT

Syntax:

do n[,]i = m1,m2[,m3]

where n is the statement label of an executable statement that follows the do statement; i is an arithmetic scalar variable whose mode is not complex; and m_1, m_2, and m_3 are arithmetic expressions whose values are converted to the mode of i. If m_3 is omitted, it is assumed to be 1.

Semantics:

Statements following the do, up to and including statement n, are executed repeatedly for different values of i. The range of a do is the series of statements executed as a result of a do, up to and including the terminal statement of the do-loop. Processing of the do statement under the ansi66 option differs from that of the ansi77 option.

Under the ansi66 option, a do statement is processed by first computing the loop count. The loop count has the value $(m_2 - m_1)/m_3 + 1$, and is truncated to integer if necessary. If the loop count is positive, it specifies the number of times the range will be executed. If the loop count is zero or negative, the range will be executed once. Before the first execution of the range, i is assigned the value of m_1. After each execution of the range, i is incremented by the value of m_3.

Under the ansi77 option, the loop count is computed as for ansi66 do-loops. Next, the value of m_3 is saved, and i is assigned the value of m_1. If the loop count is zero or negative, the range is skipped entirely and execution of the do-loop is complete. If the loop count is positive, it specifies the number of times the range will be executed. After each execution of the range, i is incremented by the previously saved value of m_3.

A do range may include another do range as long as it includes it completely. Several do ranges may end with the same statement. If a do range contains a block if statement, the corresponding end if statement must also be contained in the do range.

A do range must not end with any of the following statements: unconditional go to, assigned go to, arithmetic if, return, stop, pause, do, block if, else if, else, end if, or end. When a logical if statement ends a do range, the conditional statement is considered to be a part of the do range.

Control must not be transferred into the range of a do from a statement outside the range of the do. Control may be transferred from within the range of the do to another statement either within or outside the range. If several do ranges end in a single statement, that statement is considered to be within the range of the innermost do.

Example:

```
          do 5 i=1,n
               .
               .
               .
          do 5 j=k,10,e
               .
               .
               .
     5 x(i,j)=y
```

CONTINUE STATEMENT

Syntax:

```
continue
```

Semantics:

A continue statement is a null statement normally used to mark the end of a do range. When control is to be directed to the last statement of a do range to continue the execution of the do, it may be convenient to use a continue statement as the last statement in the range.

Example:

```
          do 10 i=1,n
                .
                .
                .
          if(a.eq.b)go to 10
                .
                .
                .
     10 continue
```

CALL STATEMENT

Syntax:

```
call s[([a[,a]...])]
```

where s is a subroutine subprogram entry name. Each a is an argument that corresponds to a dummy argument of the subprogram entry s or is a dollar sign ($) followed by the statement label of an executable statement in the calling program. Each argument can be a constant, a variable, an expression, an array name, a subprogram entry name, a statement label, or one of the built-in function names listed in Table 7-1. In the case of subroutines and entry points that have no arguments, a left parentheses followed by a right parentheses may be used to indicate the absence of arguments. Parentheses used for this purpose may be omitted from a call statement. For example, the statement

```
call doit ()
```

is equivalent to

```
call doit
```

Semantics:

Each argument is evaluated and its associated storage address becomes the storage address of the corresponding dummy argument of the subprogram entry s. The call statement transfers control to the subroutine. Execution of the return statement in the subroutine transfers control back to the caller.

Each expression or constant argument, except a character-string constant argument, must correspond to a scalar dummy argument whose mode is identical to the mode of the constant or expression. A dummy argument associated with a constant or expression argument (other than a variable) cannot be the object of an assignment statement or input operation within the called subprogram.

Each character-string constant argument must correspond to a scalar dummy argument. If the mode of the dummy argument is integer, real, or logical, the character-string constant must be four characters long. If the mode of the dummy argument is complex or double precision, the character-string constant must be eight characters long. If the mode of the dummy argument is character, the length of the constant must equal the declared length of the dummy argument.

Each variable argument must correspond to a scalar dummy argument whose mode is identical to that of the argument. Any value assigned to the dummy argument during the execution of the subprogram immediately becomes the new value of the argument. Any subscripts of the argument are evaluated by the calling program unit before transfer of control to the called program.

Each array dummy argument must correspond to an array name argument or an array element argument of the same mode. The dimensionality of the array dummy argument is constrained only in that it cannot declare an array whose storage is greater than that of the actual array argument. The same subscripts access different elements when used to reference multidimensional arrays of differing dimensions. In the following example, the array elements a(1,2) and b(2,2) occupy the same storage, as do c(10) and d(10).

Example:

```
real a(5,5),c(25)
call builder(a,c)
.
.
.
subroutine builder(b,d)
real b(4,4),d(15)
```

Each subprogram entry name argument and each built-in function name argument must correspond to a dummy argument explicitly declared in the called subprogram as a subprogram entry name. If the actual argument was a built-in function name or function subprogram name, the corresponding dummy argument must be explicitly declared in the called subprogram as a function subprogram entry and must have a mode that correctly describes the mode of the value returned by an invocation of the actual argument.

If the argument list contains a statement label argument, the dummy argument of s must contain at least one asterisk (in any argument position). Statement label arguments and asterisk dummy arguments are not considered while associating arguments with corresponding dummy arguments. (See "Dummy Arguments of Subprograms" in Section 8 of this manual.)

Examples:

```
10 call x(a,b(i),5)
   call y("this is a constant",q)
   call z(b(i),$10)
```

RETURN STATEMENT

Syntax:

```
return [n]
```

where n is an arithmetic expression whose value is converted to an integer.

Semantics:

If n is omitted, or if n≤0, the return statement is said to be a normal return. This statement returns control to the calling program unit. If n is supplied, the return statement is said to be an alternate return, and control must have been transferred to the subprogram through an entry containing one or more asterisk dummy arguments. Control is returned to the statement label in the calling program corresponding to the nth label argument when the subprogram was invoked.

The three returns in the subprogram below illustrate the use of statement label arguments:

```
          .
          .
          .
          call foo (i, j, $8, $10)
          print, "i equals j"
          .
          .
          .
     8    print, "i is less than j"
          .
          .
          .
     10   print, "i is greater than j"
          .
          .
          .
          subroutine foo (ii, jj, *,*)
          if (ii .lt. jj) then
c
c         Return to statement labelled in first
c            position of statement label list
c
          return 1
          else if (ii .gt. jj) then
c
c         Return to statement labelled in second
c            position of statement label list
c
          return 2
          else
          return
          end if
          end
```

A return statement may not appear in a main program. An alternate return statement can appear only in subroutine subprograms. If fewer than n label arguments were passed in the call, the return acts as if a value of 0 were given for n; i.e., a normal return occurs.

If the return statement appears in a function subprogram, then the value most recently assigned to the major entry of the subprogram is returned as a result of the function.

PAUSE STATEMENT

Syntax:

```
pause [n]
```

where n is a string of not more than 5 digits, or a character-string constant.

Semantics:

The action resulting from the execution of a PAUSE statement differs for full Multics and the Multics FAST subsystem. In FAST, "PAUSE" or "Pause n" is printed on the terminal and execution resumes. In full Multics, the condition 'fortran_pause' is signalled and the string "n" is made available in the condition information. A handler for the 'fortran_pause' condition can then take appropriate action. In the absence of a handler for the condition, interactive processes will print out a standard error message and establish a new command level. Absentee and daemon processes will print out the standard error message and resume execution.

STOP STATEMENT

Syntax:

```
stop [n]
```

where n is a string of not more than 5 digits or is a character-string constant.

Semantics:

When a stop statement is executed, "STOP" or "STOP n" is printed on the user's terminal and program execution stops. At this time, all files opened by the FORTRAN program are closed. See the environment section of the FORTRAN Users' Guide for subsequent action.

INQUIRE STATEMENT

Syntax:

```
inquire (ilist)
```

where ilist is a list of specifiers separated by commas.

Semantics:

An inquire statement may be used to inquire about properties of a particular named file or of a particular unit. In an inquire by file statement, exactly one file= specifier must be present, and no unit= specifier may be present. In an inquire by unit statement, exactly one unit= specifier must be present, and no file= specifier may be present. The inquire statement may be executed before, while, or after a file is connected to a unit. All values assigned are those current at the time the statement is executed.

The form of a file= specifier is

file=fn

where fn is a character expression whose value specifies the name of the file being inquired about. The form of the unit= specifier is

unit=u

where u is an integer expression whose value specifies the number of the number of the unit being inquired about.

The remaining inquire statement specifiers are described below. In many of these specifiers a variable or array element must be provided, and execution of the inquire statement causes the variable or array element to become defined with a particular value or to become undefined. The inquire statement assigns values to variables and array elements using the rules for assignment statements.

err=s
is an error specifier, where s is the label of an executable statement. If an error is encountered during the execution of the inquire statement, control is transferred to the statement whose label is s.

iostat=ios
is an input/output status specifier, where ios is an integer variable or array element. If an error occurs during the execution of the inquire statement, ios becomes defined with a positive integer value (a Multics error table code). If no error occurs, ios becomes defined with the value zero.

exist=ex
ex is a logical variable or logical array element. Execution of an inquire by file statement causes ex to become defined with the value true if a file with the specified name exists, and with the value false otherwise. Execution of an inquire by unit statement causes ex to become defined with the value true if the unit exists, and with the value false otherwise.

opened=od
od is a logical variable or logical array element. Execution of an inquire by file statement causes od to become defined with the value true if the specified file is connected to a unit, and with the value false otherwise. Execution of an inquire by unit statement causes od to become defined with the value true if the specified unit is connected to a file, and with the value false otherwise.

number=num
num is an integer variable or integer array element which is assigned the number of the unit that is currently connected to the file. If there is no unit connected to the file, num becomes undefined.

named=nmd
nmd is a logical variable or logical array element that is assigned the value true if the file has a name and false if it does not.

name=pn
pn is a character variable or character array element. It is assigned the absolute pathname of the file is the file has a name, and becomes undefined otherwise.

access=acc
acc is a character variable or character array element that is assigned the value "sequential" if the file is connected for sequential access, and with the value "direct" if the file is connected for direct access. If there is no connection, acc becomes undefined.

sequential=seq
seq is a character variable or character array element that is assigned the value "yes" if sequential is included in the set of allowed access methods for the file, "no" if sequential is not an allowed access method for the file, and "unknown" if it cannot be determined whether or not sequential is an allowed access method for the file.

direct=dir
dir is a character variable or character array element. This specifier is analogous to sequential= above, except that the query is about direct access rather than sequential access.

form=fm
fm is a character variable or character array element that is assigned the value "formatted" if the file is connected for formatted I/O, or the value "unformatted" if the file is connected for unformatted I/O. If there is no connection, fm becomes undefined.

formatted=fmt

fmt is a character variable or character array element that is assigned the value "yes" is formatted is included in the set of allowed forms for the file, the value "no" is formatted is not an allowed form for the file, and "unknown" if it cannot be determined whether or not formatted is an allowed form for the file.

unformatted=unf

unf is a character variable or character array element. This specifier is analogous to formatted= above, except that the query is about unformatted form rather than formatted.

recl=rcl

rcl is an integer variable or integer array element that is assigned the value of the record length of the file. If the file is connected for formatted I/O, the length is the number of characters. If the file is connected for unformatted I/O, the length is the number of 9 bit bytes. If there is no connection, or if the connection is not for direct access, rcl becomes undefined.

nextrec=nr

nr is an integer variable or array element. nr is assigned the value n+1, where n is the number of record most recently read or written. If the file is connected but no records have been read or written since the connection, nr is assigned the value 1. If the file is not connected for direct access, or if the position of the file cannot be determined because of a previous error condition, nr becomes undefined.

blank=blnk

blnk is a character variable or character array element that is assigned the value "null" if null blank control is in effect for the file, or the value "zero" if zero blank control is in effect for the file. If the file is not connected for formatted I/O, blnk becomes undefined.

END STATEMENT

Syntax:

end

Semantics:

An end statement marks the end of a program unit. Every program unit must have an end statement as its last line. It cannot be continued, labeled, or followed by a semicolon. If control reaches the end of a subroutine or function subprogram, control is returned to the calling program unit exactly as when a normal return statement is executed. If control reaches the end of a main program, the effect is as if a stop statement were executed, except there is nothing printed on the user's terminal. See the environment section of the FORTRAN Users' Guide for subsequent action.

SECTION 5

INPUT/OUTPUT

INPUT/OUTPUT PROCESSING

There are two different types of FORTRAN I/O statements: data transfer statements and control statements. Data transfer statements provide means of reading and writing blocks of data called records. Control statements provide means of manipulating and operating on collections of records called files. Within a FORTRAN program, a file is identified by a unit number. A unit is a generalized name used by FORTRAN to identify a file known to Multics by another name. A file is a collection of records distinct from the storage medium on which it is stored, but via a unit number it may be associated with a particular location in the storage system (e.g., a segment) or a particular device (e.g., a terminal or a tape). A unit is either connected or disconnected. A unit is connected when its unit number is associated with a location in the storage system or with an I/O device by the FORTRAN runtime I/O routines. It remains connected until explicitly disconnected or until the end of the program run.

Records

There are two forms of record: formatted and unformatted. A formatted record consists of fields of characters. A formatted record is defined as the data from one newline up to and including the next newline. The length and form of such a record is determined by the format specification and the output data transfer list used to create the record. Within the file, a formatted record appears as a collection of characters. It does not contain any information concerning the fields used to create the record.

Unformatted records consist of fields of data each being the binary representation of integer, real, logical, double precision, complex, or character data. The length of each field depends on the data type of the value it represents: integer, real, and logical data require one machine word (36 bits) each; double precision and complex data require two words each; and character data requires as many (integral) words as is appropriate (there are four characters allocated per word). Within the file, an unformatted record appears as a collection of words. It does not contain any information concerning the fields used to create the record.

Record Length

The records read and written by FORTRAN data transfer statements are logical records and have no relation to the external storage medium on which they are stored. Formatted records are an integral number of characters long. Unformatted records are an integral number of words long.

On input, formatted records are padded on the right with sufficient blanks to satisfy the input format specified. Therefore, given a format specification and an input data transfer list it is possible to calculate the number of records required, regardless of the physical length of the records. On output, trailing blanks are not removed. If the length of an output record exceeds a limit imposed by the external storage medium, FORTRAN does not create several records each of which conforms to the limit. It is the responsibility of the programmer to observe these limits.

On input, only one unformatted record is read and it must be large enough to satisfy the entire data transfer list specified in the read statement. Data not used by the read statement is lost (i.e., the next read operation on that file reads the next record). On output, unformatted records are written exactly as indicated in the write statement, i.e., no truncation occurs.

Files

A file is a collection of records, all of the same form (i.e., all formatted records or all unformatted records). Usually the records in files are of different lengths. However, a maximum record length can be attributed to a file. In such a case all records are allocated the same amount of storage, although the entire storage area of a record need not be used. If a file has a maximum record length attributed to it, the programmer must be sure to keep the records in the file from exceeding that length. Once a file has a maximum record length attributed to it, moreover, the maximum record length cannot be changed. A write statement that creates a record longer than the maximum length is in error (see "Error Processing" below). The record is not automatically broken into smaller records that conform to the maximum record length.

IO Transfer Limits

Multics FORTRAN IO works through a Multics IO DIM (Device Interface Module) which actually performs the operation. To this date all DIMs are capable of transfering a segment or less in a single operation. This limits the size of a binary IO operation, which is record oriented, to a segment or less. Thus the programmer may not be able to do binary IO of arrays or combinations of arrays which exceed this single operation size limit. This will be particularly noticable to the programmer using Very Large Arrays (VLAs).

Access to files

There are two distinct means of accessing the records of a file-- sequential and direct access.

In a sequential file, records are accessed only in the order in which they were written. Although the rewind and backspace control statements can be used to position a sequential file to a particular record, a given record in a sequential file is normally read or written only when the record immediately preceding it was the last record read or written. Only records already written can be read. The backspace control statement makes it possible to back up a sequential file one record at a time, and the rewind statement positions to the beginning of a file. However, a write of a record after repositioning by a rewind or backspace will cause all records after it to be lost.

In general, sequential files do not allow the user to change the file position to any record other than the immediately contiguous record or the beginning of the file.

In a direct access file, records are accessed by explicit record number. A unique record number with a value greater than or equal to 0 is associated with each record in the file. Therefore, the user can access any record in isolation from any other records in the file, in any order whatever. Only records already written can be read. The rewind and backspace control statements cannot be used with a direct access file unless the file also has the maximum record length attribute.

Depending on the I/O module used, certain direct access files can also be accessed sequentially, that is, according to the numerical order of the record numbers.

External and Internal Files

External files are files that exist outside of the FORTRAN program. These are the storage files or devices that are connected by traditional input/output.

Internal files provide a means of converting and transferring data within internal storage. They provide much the same capability as the nonstandard encode and decode statements. An internal file is always positioned at the beginning of the first record prior to data transfer. Reading and writing records of internal files must be performed by sequential-access, formatted I/O statements, though list-directed formatting is not allowed. A record of an internal file is defined by writing the record, and a record can be used only if it is defined. Records can be defined, or undefined, by means other than I/O statements (e.g., assignment statements). If the number of characters written is less than the length of the record, the remaining portion of the record is filled with blanks.

An internal file can be a variable, array or array element, or substring, and its data type must be character. A record of an internal file can be a variable, an array element, or a substring. If an internal file is an array, each element of the array is a record of the internal file; otherwise the internal file contains one record.

Units

Each file referenced in a FORTRAN program has a unit number associated with it. A unit number, u, is an integer value in the range $0 \leq u \leq 99$. The association between unit numbers and external storage or devices (i.e., external files) is called a connection and is established when a unit that is not yet connected (i.e., disconnected) is referenced by a data transfer statement or by a FORTRAN open statement. Connection by a data transfer statement is implicit, and connection by a FORTRAN open statement is explicit. In either case, actions taken external to the FORTRAN program run (such as prior use of the io_call command, described in the MPM Commands) are taken into account. See the description of the open statement below for a full discussion of the actions taken. Fuller details of the input/output system appear in Section 10.

Unit numbers 1 to 99 can be connected to any file or device supported by Multics I/O and by the FORTRAN runtime I/O routines. Unit number 0 is connected only to a terminal. See "Default Input and Default Output" under "Unit Attributes" below for information about default connections.

The Terminal

Terminal input/output must be in the form of sequentially accessed formatted records. The rewind, backspace, and endfile control statement cannot be used on a unit connected to the terminal. On the other hand, the prompt attribute has meaning only for units connected to the terminal. The various attributes are discussed below.

For connections to unit number 0, the FORTRAN I/O runtime routines use the user_input and user_output I/O switches for input and output respectively.

Unit Attributes

The following attributes are associated with each unit. When the unit is connected, a value is assigned to each of these. These attributes are discussed in detail in Section 10 where the entire connection process is described. If the open statement is used to connect the unit, each of these attributes can be specified explicitly using the appropriate open statement specifier. For complete information on the use of the open statement, see Section 4 of the FORTRAN Users' Guide.

1. I/O switch
   the I/O switch associated with the unit. (The ioswitch specifier.)

2. Attachment
   the attach description associated with the unit. (The attach specifier.)

3. Filename
   name of the file associated with the unit. (The file specifier.)

4. Mode
   the types of data transfer allowed for the unit; possible values are in, out, and inout. (The mode specifier.)

5. Access
   the access method associated with the unit; possible values are sequential and direct. (The access specifier.)

6. Form
   the type of records associated with the unit; possible values are formatted and unformatted. (The form specifier.) Mixing formatted and unformatted records in a single file is not supported.

7. Maximum record length
   the maximum record length, in characters, associated with the unit, but only if a maximum record length was specified. The file must be a formatted sequential file. (The recl specifier.)

8. Binary stream
   a logical value indicating whether access to the unit follows binary stream conventions (see "Binary Stream Input/Output" below). (The binary stream specifier.)

9. Prompt
   a logical value indicating that a prompt character is to be printed when input from this unit is expected; this attribute is meaningful only with units connected to the terminal. (The prompt specifier.)

10. Carriage control
    a logical value indicating that carriage control characters are processed for each output record associated with the unit; this attribute is meaningful only for files for which the mode attribute is out or inout, form is formatted, and access is sequential. (The carriage specifier.)

11. Defer newline
    a logical value indicating that newline characters should precede each line of output rather than follow each line. The same number of newline characters is printed whether the newline character precedes or follows the line; however, by deferring the printing of the newline character it is possible to: a) allow the plus (+) carriage control character at the beginning of a format statement to cause overprinting; and b) allow a program to ask a question and have the user respond on the same line. (+ carriage control characters in the second and succeeding records produced by the use of the '/' in processing a single format always have the proper effect.) This attribute is only meaningful for files for which the mode attribute is out or inout, form is formatted, and access is sequential. (The defer specifier.)

CARRIAGE CONTROL

If the carriage-control option is in effect when a file is written, the first character of the line is not printed; instead, it is transformed into a carriage-control character as follows:

Character	Resulting Control Character
0	Newline 012 (double space)
1	Newpage 014 (page eject)
blank	None (single space)
+	The previous line and the current line are written as a single line split by a carriage-return character. This causes the second line to overprint the first. If the unit is connected to the terminal and the unit does not have the defer newline attribute, this carriage control character is treated as a blank for the first record produced by a given output data transfer. It always causes overprinting when preceded by a "/" format item.
all other	None (single space)

If carriage control is not in effect, all characters of the line are written.

DEFAULT CARRIAGE CONTROL

Certain units automatically have the carriage control attribute associated with them. This attribute is honored only if the mode of the file is out or inout, the form is formatted, and the access is sequential. Unit numbers 6 and 42 have the default carriage attribute.

DEFAULT INPUT AND DEFAULT OUTPUT

Certain units have the default input attribute or default output attribute. Under the right circumstances, a unit with one of these attributes will be connected to the terminal instead of to a file in the storage system. This provides a simple way to reference the terminal from a FORTRAN program without any special knowledge. Unit numbers 5 and 41 have the default input attribute. Unit numbers 6 and 42 have the default output attribute.

In order to connect a unit that has the default input attribute to the terminal, the I/O switch associated with the terminal must not be attached already at the time of connection. Connection is either implicit or explicit. A formatted sequential read statement causes implicit connection of the unit to the terminal. Explicit connection requires the use of the open statement, with the following specifiers: the mode attribute (as in), the form attribute (as formatted), and the access attribute (as sequential); otherwise, the unit is connected to a file in the storage system.

In order to connect a unit that has the default output attribute to the terminal, the I/O switch associated with the terminal must not be attached already at the time of connection. Connection is either implicit or explicit. A formatted sequential write statement causes implicit connection to the terminal. Explicit connection requires the use of the open statement, with the following specifiers: the mode attribute (as out), the form attribute (as formatted), and the access attribute (as sequential); otherwise the unit is connected to a file in the storage system.

BINARY STREAM INPUT/OUTPUT

A binary stream file is an unformatted file. The file is a collection of machine words. Each word is directly addressable by record number. The record number of the first word in the file is zero, and so on. The number of words written by an unformatted write depends on the number of words required to store the data transfer list. An unformatted read statement reads words from the file until the list is satisfied or until the end of file is reached.

ERROR PROCESSING

When an error occurs, several actions can be taken. If the iostat specifier is given, the status variable is assigned an error code value that describes the error. Then if the err specifier is not given, execution continues with the statement immediately following the statement containing the iostat specifier. If the err specifier is given, execution continues with the statement whose statement label is specified by the err specifier.

* If the iostat specifier is not given and the err specifier is, execution continues with the statement whose statement label is specified. No error information is available.

If neither specifier is given, the program run terminates with an error message.

Throughout the rest of this section, one of the error processing actions described above is implied by the phrase "an error occurs" or "an error is encountered."

DATA TRANSFER STATEMENTS

Data transfer statements perform the transfer of data between memory and an external file or device. The read statement specifies input. For terminal input, the synonymous input statement can be used. The decode statement, which accomplishes a memory-to-memory transfer of data using a format specification, is included among FORTRAN input statements.

Output is specified by a write statement. Terminal output can also be specified by the synonymous print statement. Included among output statements is the encode statement, which performs the reverse function of decode for memory-to-memory data transmission.

Read Statement

There are five variations of the read statement, each supporting a particular type of file access. Each of these is documented separately. The syntax model below is a generalized one, including all fields that can be supplied with a read. Each form of the read, however, has a different requirement for fields, as shown in the individual syntax models that follow.

```
read (u'k, n, end=a, err=b, unit=u, fmt=n, rec=k) list
```

where:

u — integer expression giving unit number (0-99), the name of an internal file, or an asterisk (*). An asterisk appearing as a unit specifier indicates a default, preconnected unit. In Multics FORTRAN this is unit 0, which is connected to the user's terminal.

k — integer expression giving record number; the record k must exist in the file at the time the read statement is executed. The value of k must be $\geq$ 0 and is specified by the user. The FORTRAN I/O runtime routines use the value of this expression as the record number. The value of this expression is not changed by these routines. If the record number field is present, the statement is a direct access read statement. If the field is not present, the separating apostrophe must also be omitted and the statement is a sequential read statement.

n — label of a format statement, a character-string constant, the name of a character variable, a character expression involving substrings and/or concatenation, a namelist name, the name of a character or integer array that contains the character-string representation of a format, or an asterisk (*) specifying list-directed formatting. It may also be a simple integer variable that is defined with a value representing a format string. Such a variable may be defined by executing an assign statement containing the label of a format statement. An n is used only with formatted operations and must be omitted (along with the separating comma) for unformatted ones; a list-directed operation can be specified by omitting n but retaining the comma. (See "Format Specifications" later in this section.)

a — a statement label indicating where control is to be transferred at the end of an input file. This field is optional for all sequential read statements. It is invalid for all direct access read statements.

b — a statement label indicating where control is to be transferred in case of error. This field is optional for all read statements.

list — a data transfer list (described later under "Data Transfer Lists"); the data transfer list can be omitted, in which case execution of the statement causes at least one record to be read and discarded.

END OF FILE RECORD

The "end=a" and "err=b" options can appear in either order and either or both can be omitted. If omitted, the separating comma or commas must be omitted. If end=a is not supplied and end of file is reached, the program run terminates with an error message. If err=b is not supplied, the program run terminates with an error message when any input/output error occurs.

If a record consisting of the backslash character "\" followed by the letter f is encountered during any formatted sequential read operation, it is considered an end-of-file record and the end-of-file condition is raised. If the read statement includes end=b, control is transferred to the statement whose label is b; otherwise, the program run terminates with an error message. Because this end-of-file record need not be the physical end of the file, data records can follow the end-of-file record and are read when subsequent read statements are directed at the file. This allows the end-of-file record to be used to separate sets of data read from a single file.

KEYWORDS

Three additional keywords can be used. The "unit=u" keyword can be used to specify the unit on which the data transfer will take place. The "fmt=n" keyword can be used in formatted read and write statements to specify the format which will control the data transfer. The "rec=k" keyword can be used in direct access read and write statements to specify the number of the record to be transmitted.

If the "unit=u" keyword is omitted, the unit specifier must appear first in the control list of the read or write statement. If the "fmt=n" keyword is omitted, the format specifier must appear second in the control list and the first item in the control list must be a unit specifier with the "unit=u" keyword omitted.

FORMATTED SEQUENTIAL READ

Syntax:

```
read (u,n,end=a,err=b,unit=u,fmt=n) list
```

Semantics:

Formatted records are read from the file or device associated with unit u under control of the format specified by n. The data contained in the records is transmitted to the variables specified by the list. The number of records that are read depends on the number of elements in the list and on the content of the format. Records are read beginning at the current file position. The file position is left following the last record read.

```
read(8,100) x,y,z
read(unit=8,fmt=100) x,y,z
read(i,65,err=101) (x(i),i=1,10)
read(5,,end=10) ab, c
read (75,)
```

FORMATTED DIRECT ACCESS READ

Syntax:

```
read (u'k,n,err=b,unit=u,fmt=n,rec=k) list
```

Semantics:

Formatted records are read beginning with record number k in the file or device associated with unit u under the control of the format specified by n. The data contained in the records is transmitted to the variables specified by the list. The number of records that are read depends on the number of elements in the list and on the content of the format. The file position is left following the last record read. If more than one record is requested, records with record numbers k, k+1, k+2,...etc. are read. The value of the last record number is not made known to the user.

Examples:

```
read(8'i,100)x,y
read(rec=i,unit=8,100)x,y
read(i'j,10)a,b,c,d(n)
read(8'n,)cur,next
read(10'i+m,55,err=37)
```

TERMINAL READ STATEMENT

Syntax:

```
read n,list
  or
read, list
  or
input n,list
  or
input, list
```

Semantics:

A terminal read statement is equivalent to a formatted sequential read statement as shown below:

```
read n,list              read(0,n)list
read,list                read(0,10)list
                      10 format(v)
input n,list             read(0,n)list
input,list               read(0,10)list
```

UNFORMATTED SEQUENTIAL READ STATEMENT

Syntax:

```
read (u,end=a,err=b,unit=u) list
```

Semantics:

An unformatted record is read from the file or device associated with unit u. The number of words specified by the items in the list must not exceed the number of words specified by the items in the list used to create the record. The file position is left following the record read.

The data contained in the record is transmitted to the variables in the list. The data contained in the record is assumed to be of the same mode as the list items. No mode conversion is performed between the data in the record and the list items, and no checking is performed.

Examples:

```
read(51)a,b,c
read(j,err=99)x,y,z
read(err=99,unit=j)x,y,z
read(1,err=64,err=19)p,q,r
```

UNFORMATTED DIRECT ACCESS READ STATEMENT

Syntax:

```
read(u'k,err=b,unit=u,rec=k) list
```

Semantics:

The unformatted record number k is read from the file or device associated with unit u. Such a record must exist; if not, an error occurs. The number of words specified by the items in the list must not exceed the number of words specified by the items in the list used to create the record. The file position is left following the record read.

The data contained in the record is transmitted to the variables in the list. The data contained in the record is assumed to be of the same mode as the list items. No mode conversion is performed between the data in the record and the list items, and no checking is performed.

Example:

```
read(10'm+n,err=25)x,y,z
```

DECODE STATEMENT

Syntax:

```
decode(c,n)list
```

where c is a variable, an array name, or an array element of any mode other than logical. For this statement, list is required.

Semantics:

The storage identified by c is considered to be one or more records. Values are read from those records and stored into the variables specified by the list under control of the format identified by n. New records are selected in the manner described in "Interaction Between Format and Input/Output List" below.

If c is a variable or array element, the format must request a single record containing no more than j characters where j is 4 if the variable or array element is of integer or real mode, 8 if complex or double-precision mode, and the declared length if the variable or array element is character mode. If it requests less than j characters from c, the leftmost characters are read and the remaining characters of c are ignored. If more than j characters are requested or if a second record is requested, an error occurs.

If c is an array of character mode consisting of k array elements, each j characters in length, the format must not request more than k records, each no more than j characters in length. Each record is read from one element of c beginning with the first element. If more than j characters are requested from a record or if more than k records are requested, an error occurs.

If c is an array of arithmetic mode, it must be an array of one dimension. The entire array is processed as if it were a character-string variable of length j where j is equal to the number of elements in the array, multiplied by 4 if the array is integer or real mode, or multiplied by 8 if the array is complex or double-precision mode.

Example:

```
      integer x(5)
      character*10 c(5)
      read 5,c
    5 format(5a10)
      decode (c,10) (x(i),i=1,5)
   10 format (5(3x,i7/))
```

All five elements of the array are "read" by this example.

NAMELIST READ STATEMENT

Syntax:

```
read(u,n,end=a,err=b)
```

Semantics:

Records of the form described under the "Namelist Statement" below are read from the file or device associated with unit u, under control of the namelist specified by n.

Examples:

```
namelist/input/a,b,c,d
read(5,input,end=10)
```

There are five variations of the write statement, each supporting a particular type of file access. Each of these is documented separately. The syntax model below is a generalized one, including all fields that can be supplied with a write. Each form of the write, however, has a different requirement for fields, as shown in the individual syntax models that follow.

```
write (u'k,n,err=b,unit=u,fmt=n,rec=k)list
```

where:

u — integer expression giving unit number (0-99), the name of an internal file, or an asterisk. An asterisk appearing as a unit specifier indicates a default, preconnected unit. In Multics FORTRAN this is unit 0, which is connected to the user's terminal. See "Keywords" above.

k — integer expression giving record number; k must be ≥ 0 and is specified by the user. The FORTRAN I/O runtime routines use the value of this expression as the record number. The value of this expression is not changed by these routines. If the record number field is present, the statement is a direct access write statement; if this field is not present, the separating apostrophe must also be omitted and the statement is a sequential write statement.

n — label of a format statement, a character-string constant, a character variable, a character expression involving substrings and/or concatenations, a namelist name, or the name of a character or integer array that contains the character-string representation of a format. It may also be a simple integer variable that is defined with a value representing a format string. Such a variable may be defined by executing an assign statement containing the label of a format statement. An n is used only with formatted operations and must be omitted (along with the separating comma) for unformatted ones; a list-directed operation can be specified by omitting n but retaining the comma. (See "Format Specifications" later in this section.)

b — a statement label indicating where control is to be transferred in case of error. The "err=b" option can be omitted; if omitted, the comma must be omitted. Without the "err=b" option, an error causes the program run to terminate with an error message.

list — a data transfer list (described later under "Data Transfer Lists"); the data transfer list can be omitted.

FORMATTED SEQUENTIAL WRITE STATEMENT

Syntax:

```
write (u,n,err=b,unit=u,fmt=n) list
```

Semantics:

Formatted records are written to the file or device associated with unit u under the control of the format identified by n. The data contained in the variables specified in the list is written to the records. The number of records written depends on the number of elements in the list and on the content of the format. The file position is left following the last record written.

Examples:

```
write(10,format) a,b,c
write(8,105) (a(i),i=1,in)
write (fmt=105,unit=8)a,b,c
write(37,err=909)x,y,z
```

FORMATTED DIRECT ACCESS WRITE STATEMENT

Syntax:

```
write (u'k,n,err=b,unit=u,fmt=n,rec=k)list
```

Semantics:

The current values of the list elements are written as formatted records beginning with record k to the file or device associated with unit u, under the control of the format identified by n. The data contained in the variables specified by the list is written to the records. Existing records are overwritten. The number of records written depends on the number of elements in the list and on the content of the format. The file position is left following the last record written.

If more than one record is produced, records with record numbers k, k+1,k+2,...k+n are written. The value of the last record number is not made known to the user.

Examples:

```
write(8'j,800)a,b,c
write(rec=j, unit=8,800)a,b,c
write(30'n+m,format)xyz
```

PRINT STATEMENT

Syntax:

```
print n,list
  or
print,list
```

Semantics:

A print statement is equivalent to a formatted sequential write statement as shown below:

```
print n,list              write (0,n)list
print,list                write (0,10)list
                       10 format (v)
```

UNFORMATTED SEQUENTIAL WRITE STATEMENT

Syntax:

```
write (u,err=b,unit=u) list
```

Semantics:

A single unformatted record is written to the file or device associated with unit u. The record contains a copy of the current value of every element in the list, written in the order in which it appears in the list.

Examples:

```
write (51) a
write (k) x,y,z(i)
```

UNFORMATTED DIRECT ACCESS WRITE STATEMENT

Syntax:

```
write(u'k,err=b, unit=u, rec=k) list
```

Semantics:

A single unformatted record is written to the file or device associated with unit u. The record contains a copy of the current value of every element in the list, written in the order in which it appears in the list.

Examples:

```
write(10'250,err=6)a,b,c
write(15'n+m)x,y
write(rec=n+m,unit=15)x,y
```

ENCODE STATEMENT

Syntax:

```
encode(c,n)list
```

where c is a variable, an array element, or an array name of any mode other than logical. For this statement, the list is required.

Semantics:

The current values of the elements specified by the list are transmitted to the storage identified by c under control of the format identified by n. The storage identified by c is considered to be one or more records. New records are selected in the manner described in "Interaction Between Format and Input/Output List" under "Format Specification."

If c is a variable or an array element, the format must produce a single record containing no more than j characters, where j is equal to 4 if the variable or array element is of integer or real mode, 8 if complex or double-precision mode, and the declared length if character mode. The record is contained (left justified) within c; any excess characters in c are unmodified. If more than j characters are produced or if a second record is produced, an error occurs.

If c is an array of character mode consisting of k array elements, each j characters in length, the format must not produce more than k records, each no more than j characters in length. Each record is stored into one element of c beginning with the first element. If more than j characters are produced from a record or if more than k records are produced, an error occurs.

If c is an array of an arithmetic mode, it must be an array of one dimension. The entire array is processed as if it were a character-string variable of length j, where j is equal to the number of elements in the array, multiplied by 4 if the array is integer or real mode, or multiplied by 8 if the array is complex or double-precision mode.

Example:

```
      integer x,z
      character*40 c(5)
      encode(c,10)x,y,z
   10 format (i10,f10.3,1,10x,i10)
```

The first two elements of the array c are modified by this example.

NAMELIST WRITE STATEMENT

Syntax:

```
write(u,n,err=b)
```

Semantics:

The current values of the variables identified by namelist n are written as a single record of the form described under the "Namelist Statement" below to the file or device associated with unit u.

Examples:

```
namelist/output/x,y,z
write(6,output)
```

I/O CONTROL STATEMENTS

The I/O control statements are used to change attributes associated with units. The open and close statements connect or disconnect the unit respectively. In addition, the open statement, used primarily to associate a unit with an external file or device, can also be used to specify and change attributes of units that are already connected.

The rewind, backspace, and endfile statements affect the "current position" associated with the file or device.

The following descriptions assume the use of standard I/O modules (see Section 10 for information about them). The use of nonstandard I/O modules may impose additional restrictions on the features and capabilities described. Some of the more commonly used nonstandard I/O modules are discussed in Section 4 of the FORTRAN Users' Guide.

Open Statement

The open statement is used to:

- connect a file or device to a unit,
- create a file and then connect it to a unit, or
- change the value of the access, mode, prompt, defer newline, or carriage control attributes for a connected unit.

The syntax of the open statement is:

```
open(u,olist)
```

where u is an integer expression giving the unit number and olist is a list of zero or more specifiers in any order. (A specifier is a keyword followed by an equal sign followed by an expression.) In most cases, the keyword used to specify an attribute is the same as the name of the attribute. There is a difference between the keyword (specifier) and the attribute. Not all keywords allowed in the open statement specify file attributes, e.g., err=, iostat=, and status=. The unit= keyword may be used to specify the unit number instead of placing the unit number in the first position as indicated above. If this keyword is used, it can appear anywhere in the list of specifiers. The recognized specifiers are:

```
access=acc
attach=atd
binary stream=bs
blank=blnk
carriage=car
defer=d
err=s
file=fname
form=f
iostat=ios
ioswitch=sw
mode=io
prompt=p
recl=len
status=x
unit=u
```

The specifiers are described in more detail below.

access=acc
: specifies the access method desired, where acc is a character expression whose value when any trailing blanks are removed is either:

```
sequential
direct
```

This specifier defines the access method for the unit. If not specified when the unit is connected, the default is:

```
sequential
```

attach=atd
specifies an attach description to be used to attach the I/O switch, where atd is a character expression that is passed directly to the Multics I/O system as an attach description. (For a description of Multics I/O, see Section 10 below and MPM Reference Guide, Section 5). See the FORTRAN Users' Guide for a detailed explanation of how this specifier interacts with other specifiers in an open statement. This specifier cannot be given if the file specifier is present, if the I/O switch is already attached, or if the unit is already connected.

blank=blnk
specifies the manner in which trailing and embedded blanks are to be treated during number conversions, where blnk is a character expression whose value when any trailing blanks are removed is either:

null
zero

This specifier indicates whether trailing and embedded blanks are to be ignored or treated as zeros. If the specifier is "null," then such blanks are ignored and given no numerical significance. If the specifier is "zero," then these blanks are treated as zeros. The default for handling trailing and embedded blanks depends on whether the program is compiled under the ansi77 option or the ansi66 option. Under the ansi77 option, the default is to ignore those blanks, and under the ansi66, they are treated as zeros.

binary stream=bs
specifies the value of the binary stream attribute, where bs is a logical expression. If bs evaluates to .true., the binary stream attribute is associated with the specified unit. See Section 10 below for a detailed description of the binary stream attribute. If the binary stream attribute evaluates to .true., the recl specifier must not be given and the file must be an unformatted file. This specifier cannot be given if the unit is already connected.

carriage=car
specifies the value of the carriage control attribute, where car is a logical expression. This attribute is meaningful only if the mode is:

out
inout

the form is:

formatted

and the access is:

sequential

For a description of carriage control processing, see the section "Carriage Control" above. If not specified, the default value is copied from the previous use of this unit in the program run. At the beginning of a program run its value is .false., except for unit numbers 6 and 42.

WARNING: Once this attribute is associated with a unit, it remains with that unit for the life of the program run, or until it is explicitly changed. If a unit is given this attribute and then disconnected, all subsequent connections acquire this attribute also.

defer=d

specifies a value for the defer newline attribute, where d is a logical expression. If d evaluates to .true., the specified unit is given the defer newline attribute. If a unit has the defer newline attribute, then the last newline character of a write statement is not printed immediately. Instead, it is printed before the next line of output is printed or when the program run terminates. This mode of operation allows the "+" carriage control to work in the first record produced by a given write statement and also allows the program to ask questions that are answered on the same line. This attribute is meaningful only if the mode is:

out
inout

the form is:

formatted

and the access is:

sequential

If not specified, the default value is copied from the previous use of this unit in the program run. At the beginning of a program run its value is .false.

WARNING: Once this attribute is associated with a unit, it remains with that unit for the life of the program run, or until it is explicitly changed. If a unit is given this attribute and then disconnected, all subsequent connections acquire this attribute also.

err=s

specifies an error return statement label, where s is a statement label indicating where control is transferred in case of an error during the execution of an open statement. If an error occurs and this specifier and the iostat specifier are not given, the program run terminates with an error message. See "Error Processing" above for a complete description of error processing.

file=fname

specifies a file name, where fname is a character expression whose value when any trailing blanks are removed is:

the pathname of the file to
be connected to the unit

If this specifier is not given when the unit is connected and a file name is required for the attachment, the value filenn is used, where nn is a two-digit representation of the unit number. This specifier cannot be given if the attach specifier is present, if the I/O switch is already attached, or if the unit is already connected.

form=f

specifies the form of the records in the file or device, where f is a character expression whose value when any trailing blanks are removed is either:

formatted
unformatted

If not specified when connecting a unit, the default is:

unformatted

This specifier cannot be given if the unit is already connected.

form=f
specifies the form of the records in the file or device, where f is a character expression whose value when any trailing blanks are removed is either:

formatted
unformatted

If not specified when connecting a unit, the default is:

unformatted if ansi66 or access = "direct", else formatted

This specifier cannot be given if the unit is already connected.

iostat=ios
specifies a status variable, where ios is an integer variable or an integer array element. After execution of the open statement, the integer variable or integer array element contains zero if no error occurred during the execution of an open statement or a nonzero value if an error occurred. Whether or not there is an error, no message is printed and execution continues with the statement immediately following the open statement (unless the err specifier is given). If an error occurs and this specifier and the err specifier are not given, the program run terminates with an error message. See "Error Processing" above for a complete description of error processing.

A character string describing the meaning of an I/O status code can be obtained by calling the convert_status_code_, com_err_, or sub_err_ subroutines. Note that the absolute value of the status code must be passed, since the negative of a standard system status code is returned where the FORTRAN Standard requires a negative value. See "List of System Status Codes and Meanings" in the Multics Programmer's Reference Manual for a list of error codes.

ioswitch=sw
specifies the name of the I/O switch, where sw is a character expression whose value when any trailing blanks are removed is:

the name of the I/O switch to be
associated with the specified unit

See Section 12 for a detailed explanation of how this specifier interacts with others in an open statement. This specifier cannot be given if the unit is already connected.

mode=io
specifies the types of data transfer that are desired for the unit, where io is a character expression whose value when any trailing blanks are removed is:

in
out
inout

If out or inout is specified, the user must have write access to the file or device associated with the unit. If not specified when the unit is connected, the default is:

inout

prompt=p
specifies the value of the prompt attribute, where p is a logical expression. If p evaluates to .true., the unit is given the prompt attribute. This causes the two-character sequence "? " to be output to the terminal immediately prior to each input request for that unit. If the unit is not the terminal, this attribute is ignored. If not specified, the default value is copied from the previous use of this unit in the program run. At the beginning of a program run its value is .false.

WARNING: Once this attribute is associated with a unit, it remains with that unit for the life of the process or run unit, or until it is explicitly changed. If a unit is given this attribute and then disconnected, all subsequent connections acquire this attribute also.

recl=len
specifies a maximum record length, where len is an integer expression whose value must be greater than zero. This specifier can be used to specify that a file or device has a maximum record length attribute or it can be used to change the value of that maximum record length attribute. If the file does not exist or if the file is empty, any value can be specified. If the file already exists and is not empty, the file must already have the maximum record length attribute, and the maximum record length attribute associated with the file must equal the value specified. This specifier cannot be given if the unit has the binary stream attribute or if the unit is already connected and it does not already have the maximum record length attribute.

status=x
where x is a character expression (i.e., quoted) whose value when any trailing blanks are removed is:

new
old
scratch
unknown

If the status value is "old," a file= keyword must be present in the open statement, and the named file must exist. If the status value is "new," a file= keyword must be present in the open statement, but the named file cannot exist already. If the status value is "scratch," no file= keyword can be used. A temporary file is created for the unit in the process directory, and it is deleted when the unit is closed. The corresponding close statement cannot have the status value "keep" if the unit is opened with status value "scratch." If the status value is "unknown," a file will be created if necessary. The file= keyword may be used; if it is not, the file name will be derived from the unit number. When no status specifier is present in an open statement, the status value is assumed to be "unknown."

WARNING: Prior to implementation of ansi77, specifying this field had no effect on execution of a program, but that is no longer true. The status= keyword specifications in previously written programs may now affect the execution of those programs.

unit=u
specifies the unit to be used, where u is the unit's number. Exactly one unit specifier must be present in the open statement. If this keyword is not used, the unit number must be placed in the first position of the open statement.

Opening a Connected Unit

If an open statement specifies a unit that is already connected, only certain specifiers are permitted. If an invalid specifier is given, an error occurs. The permitted specifiers for an open statement referencing an already connected unit are:

```
err=s
iostat=ios
recl=len
mode=io
access=acc
prompt=p
defer=d
carriage=car
```

The effect of giving any of these specifiers in an open statement for a connected unit is to change the corresponding attribute to have the value specified. Changing the maximum record length is permitted only if there are no records in the associated file. Specifying a maximum record length for an already existing file that does not have this property is not permitted.

Close Statement

The close statement is used to disconnect a unit, i.e., remove the association between a unit number and the file it currently represents. The close statement places the file in a consistent state by:

- flushing any internal buffers managed by the FORTRAN runtime I/O routines, and
- restoring the Multics internal I/O data bases to their state prior to the corresponding open.

Certain changes in the use of a file require that the unit be closed and reopened. For a discussion of such constraints see Section 10.

The syntax of the close statement is:

```
close([unit=]u,clist)
```

where u is an integer expression giving the unit number and clist is a list of specifiers. The unit= keyword may be used to specify the unit number instead of placing the unit number in the first position as indicated above. If the optional keyword is used, it can appear anywhere in the list of specifiers. The recognized specifiers are:

```
err=s
iostat=ios
status=st
unit=u
```

The err, iostat, and unit specifiers have the same meaning as for the open statement; the status specifier is described below.

status=st
specifies the file disposition after it is disconnected, where st is a character expression whose value when any trailing blanks are removed is either:

keep
delete

If delete is specified, the file associated with the specified unit is deleted after disconnecting if and only if the file was attached and opened by the FORTRAN runtime I/O routines. If keep is specified, no further action is taken after closing. If this specifier is not given, the default value is keep. See above for the status of scratch files.

Rewind Statement

The rewind statement is used to set the current position of a file or device to the beginning of the file or device. It is permitted only for units that have the sequential attribute.

The syntax of the rewind statement is:

```
rewind ([unit=]u[,err=s,iostat=ios])
```

where u is an integer expression representing the unit number of the file to be repositioned, s specifies the error label, and ios is the input/output status specifier. If the position is already at the beginning, there is no change. The unit specifier must appear exactly once, and if the unit= keyword is omitted, the specifier must be in the first position. The err= and iostat= keywords can appear no more than once.

Backspace Statement

The backspace statement is used to set the current position of a file to the record immediately preceding the current record. It is permitted only for units that have the sequential attribute.

The syntax of the backspace statement is:

```
backspace ([unit=]u[,err=s,iostat=ios])
```

where u is an integer expression representing the unit number of the file to be positioned, s specifies the error label, and ios is the input/output status specifier. If the position is at the beginning of the file or device, there is no change. The unit specifier must appear exactly once, and if the unit= keyword is omitted, the specifier must be in the first position. The err= and iostat= keywords can appear no more than once.

NOTES ON SEVERAL ENDFILE VERSIONS

There are currently three separate actions that can be taken as a result of executing an endfile statement, depending on the version of the compiler used to generate the object code actually executing. The old_fortran compiler generates an endfile statement that behaves like a close statement instead of performing the actions mentioned above. The new_fortran compiler generates an endfile statement whose execution depends on the version of the runtime FORTRAN I/O routines used (see MPM Commands) The MR5.0 and MR6.0 versions of the runtime FORTRAN I/O routines execute it as a statement that does not affect the file or device but prints a warning message. With releases MR9.0 and beyond, the endfile statement as executed by the runtime FORTRAN I/O routines behaves as indicated in the previous paragraph.

DATA TRANSFER LISTS

A data transfer list specifies values to be transmitted during the execution of the data transfer statement in which the list appears.

Syntax:

e[,e]...

If the list is used in a read or decode statement, each e is the name of a variable, an array element, an array name, or an implied do-loop. If the list is used in a write or encode statement, each e is an expression, an array name, or an implied do-loop.

Semantics:

Beginning with the leftmost item, each item is evaluated and its value transmitted to or from the record(s) referenced by this data transfer statement. The evaluation of an item consists of the computation of its subscripts, evaluation of the expression, or the execution of all implied do-loops contained in the item.

Examples:

```
a,x(i,j),(y(k),k=1,n)
a,b,c
```

IMPLIED DO-LOOPS

An implied do-loop is similar in function to a do statement. Its execution causes one or more list elements to be transmitted to or from a record.

Syntax:

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

where list is a data transfer list; i is an arithmetic variable whose mode is not complex; and m_1, m_2, and m_3 are arithmetic expressions whose modes are not complex and whose values are converted to the mode of i. If m_3 is omitted, it is assumed to be 1.

Semantics:

The semantics of an implied do-loop are analogous to those of a do-loop. Processing of an implied do-loop may cause no elements to be transmitted if the ansi77 option is in effect (see DO STATEMENT in Section 4). The range of an implied do-loop is specified by the enclosing parentheses.

Examples:

Implied Do List	Analogous Do Statements	Equivalent List Naming Each Array Element Separately
(x(i),i=1,n)	do 99 i=1,n 99 x(i)	x(1),x(2),...,x(n)
((a(i,j),j=1,n),i=1,m)	do 80 i=1,m do 80 j=1,n 80 a(i,j)	a(1,1),a(1,2),...,a(1,n) a(2,1),a(2,2),...,a(2,n) . . . a(m,1),a(m,2),...,a(m,n)
(i,j,a(k,i,j),k=1,15)	do 60 k=1,15 60 i,j,a(k,i,j)	i,j,a(1,i,j) i,j,a(2,i,j) . . . i,j,a(15,i,j)

The order in which list elements are transmitted makes it possible for the evaluation of subscripts or an implied do-loop variable to be affected by the transmission of previous list elements in an input statement.

Example:

```
read(5,100)i,k,a(i,k)
```

The values of i and k used in the evaluation of a(i,k) are the values just transmitted by this read operation. If a nonsubscripted array name is used as a list element, it is equivalent to the use of each of the array elements listed in the order in which the elements occur in storage, except that the resultant code may be more efficient.

Example:

```
      dimension a(2,3)
           .
           .
           .
      read (2) a
           .
           .
           .
```

is equivalent to:

```
      dimension a(2,3)
           .
           .
      read(2) a(1,1),a(2,1),a(1,2),a(2,2),a(1,3),a(2,3)
           .
           .
           .
           .
```

FORMAT SPECIFICATIONS

Data transfer operations performed by formatted read, formatted write, input, print, encode, or decode statements are controlled by format specifications. A format specification may be contained in a format statement, or expressed as a character-string constant, a character-string variable, or an integer or character-string array. It consists of a set of zero or more field descriptors separated by commas, colons, or record delimiters. If the format specification is contained in a variable, array, or constant, the first nonblank character must be a left parenthesis. The matching right parenthesis terminates the specification. Characters following the terminating right parenthesis in a character-string constant or a format statement are invalid and cause an error during compilation. Characters following the terminating right parenthesis in a variable or an array are ignored.

Syntax:

```
([[^l,][/]...f[cf]...[,$]])
              or
([[^l,]v[,$]])
              or
([[^l,]*[,$]])
```

where ^l are the characters caret and l; / is zero or more slash characters; each f is either a control item, a field descriptor, or a repeat group; each c is a comma, a colon, or a set of one or more slash characters; $ is the dollar sign character; v is the lowercase letter v; and * is an asterisk. The ^l is optional and if omitted, the comma must also be omitted. Under the ansi66 option, s may be used to serve the same function as ^l. The $ is optional and if omitted the comma must also be omitted. Both s and $ cannot appear in the same format specification. A format specification may be empty (i.e., it may be a left parenthesis followed immediately by a right parenthesis).

Semantics

If the format specification is empty, the data transfer list must be omitted. For input operations, one record is read and discarded; for output operations, a blank record is written. The s indicates that the file contains line numbers that are to be skipped on input. The s is invalid for output. A slash delimits records. For input operations, the next record is read. For output operations, the current record is written and output continues into a new record. See the following pages for control items, field descriptors, and repeat groups. Commas or colons are used to delimit the separate specifications. When a colon is used to delimit specifications, it implies a comma and terminates processing of the format specification if the data transfer list has been exhausted. When a set of one or more slashes is used to delimit specifications, it implies a comma and delimits records. For a complete description of format processing refer to "Interaction Between Format and Input/Output List" later in this section. The $ indicates that the newline character that would normally follow the final output record is suppressed. On input, the $ is ignored. The v and the asterisk indicate list-directed input/output, which is performed as described under "List-Directed Input/Output" later in this section.

A control item can have one of the following forms:

```
s
sp
ss
^1
nxs
tn
tln
trn
nha1a2...an
"a1a2...an"          character-string constants
'a1a2...an'
$
bn
bz
```

where n is an unsigned positive integer constant, except in the case of bn; a_1, a_2, through a_n are ASCII characters; and the letters s, x, t, h, bn, and by represent themselves and are control codes.

The s control item's meaning depends on whether the program is compiled using ansi66 or ansi77. In program units compiled under the ansi66 option, the s control item indicates that the file contains line numbers that are to be ignored. For this purpose, a line number is a string of one or more digits followed by one nondigit and occurring at the beginning of a record. When used in this way, the s control item must appear at the beginning of a format and can be used only for input.

In programs compiled under the ansi77 option, the s control item pertains to the production of plus (+) signs in numeric output fields and is directly related to the sp and ss control items. At the beginning of a formatted output statement's execution, the system has the option of producing plus signs in numeric output fields. If an sp control item is encountered in the format specification, the system must produce plus signs that would ordinarily be optional. If the ss control item is encountered, the system must not produce plus signs. If an s control item is encountered in a format statement compiled under the ansi77 option, the option of producing the plus signs is returned to the system. (The sp and ss control items operate under both the ansi77 and ansi66 options.) On Multics, optional plus signs are not produced unless an sp control item is used. These three controls pertaining to plus signs have no effect on the execution of an input statement.

The ^1 (caret 1) control item can be used only at the beginning of a format and only for input. As with the s control item in the ansi66 option, the ^1 control item indicates that the file contains line numbers that are to be ignored. Under the ansi66 option, either the s or the ^1 control item can be used to indicate that line numbers in the input records are to be skipped, while under the ansi77 option, only the ^1 control item can indicate this.

The nx form causes the next n characters of the current input record to be skipped. If the ansi66 option is in effect, skipped characters in an output record are cleared to blanks.

The tn form causes the processing of the current record to continue with the nth character position of the record. This control item is used to permit the order of the elements in the data transfer list to differ from the order of the fields of a record. If the ansi66 option is in effect and the new position is to the right of the current end of the record in the buffer, the record will be extended to the new position with blanks.

The tln and trn forms indicate that transmission of the next character to or from the record is to occur at the character position n characters from the current position. The tln form causes processing of the current record to continue n characters to the left of the current position. If n is greater than the current position, the next transmission takes place at position 1 of the record. The trn form causes processing to continue n characters to the right or forward of the current position. If the ansi66 option is in effect and the new position is to the right of the current end of the record in the buffer, the record will be extended to the new position with blanks.

A character-string constant has the same form as described for character-string constants in Section 2. However, in this case, when quotation marks are used to delimit a character string, the string itself cannot include quotation marks; likewise apostrophes cannot appear within character-string constants delimited by apostrophes. Used as a control item, a character-string constant causes the value of the constant to be placed in the current output record. If a character-string constant is used to describe an input record, the constant is replaced by the corresponding field of the record.

Examples:

5x	causes five characters of the current record to be skipped. If this is an output record and the ansi66 option is in effect, those five characters will be cleared to blanks.
t25	causes the processing of the current record to continue with the 25th character position of the record.
"x equals"	causes the letters "x equals" (without the quotation marks) to appear in the current output record or causes the constant "x equals" (without the quotation marks) to be replaced by the next eight characters of the current input record.
8hx equals	this item is equivalent to the previous example.

The $ control item can be used only at the end of a format, and is meaningful only on output. It indicates that the newline character that would normally follow the final output record is suppressed.

The bn and bz control items specify the interpretation of trailing and embedded blanks in numeric input fields. They are not meaningful for output. At the beginning of execution of a formatted input statement, the interpretation of all blanks other than leading blanks is dictated by the blank= keyword in the open statement for the unit or by the appropriate default if no blank= keyword is given. If a bn control item is encountered in the format specification, trailing and embedded blanks are ignored (treated as null) in succeeding numeric input fields. If a bz control item is encountered, trailing and embedded blanks are treated as zeros in succeeding numeric input fields.

A field descriptor can have one of the following forms:

```
Sr_w.d
Srew.d
Srdw.d
Srgw.d
Srew.dee
Srew.dee
riw
riw.m
raw
ra
rrw
row
rlw
```

where the underlined letters (a, d, e, f, g, i, l, o, r) are conversion codes that control the conversion of input fields or output variables.

The w is an unsigned, nonzero, integer constant whose value describes the width of a field in the record. The d is an unsigned integer constant whose value describes the number of decimal fractional digits expected in an input field or required in an output field. Its meaning varies slightly when used with a g conversion code. The e is a signed, nonzero, integer constant whose value indicates the exponent in an output field. The m is an unsigned, integer constant whose value indicates the minimum number of digits in an output field. The r is an optional, unsigned, nonzero, integer constant whose value indicates the number of times this field descriptor is to be repeatedly used. The S is a scale-factor designator as defined below. The scale factor designator is in no way related to the s control mentioned earlier.

NUMERIC CONVERSION

The numeric field descriptor i is used to specify input/output of integer data. The numeric field descriptors f, e, g, and d are used to specify input/output of real, double-precision, and complex data. The following rules apply:

1. With all numeric input conversions, leading blanks are not significant. Other blanks are treated in one of two ways. For programs compiled under the ansi77 option, they are ignored by default and thus have no numerical significance. For programs compiled under the ansi66 option, they are treated as zeros by default. Plus signs may be omitted. A field of all blanks is considered to be zero.

2. With the f, e, g, and d input conversions, a decimal point appearing in the input field overrides the decimal point specification supplied by the field descriptor. If the input value does not contain a decimal point, a decimal point is assumed to immediately precede the rightmost d digits.

3. With all output conversions, the output field is right justified. If the number of characters produced by the conversion is smaller than the field width, leading blanks are inserted in the output field.

4. With all output conversions, the character-string representation of a negative value is signed; a positive value is unsigned. This is the default. Under ansi77, this signing convention can be changed.

5. If the number of characters produced by an output conversion exceeds the field width, the entire output field is filled with asterisks.

INTEGER CONVERSION

The numeric field descriptors iw and iw.m describe a field w characters wide that contains the character-string representation of an integer. The list element associated with this field descriptor must be an integer value.

For input conversion, the field consists of an optional sign followed by a string of digits. Leading blanks are not significant. Other blanks are treated in one of two ways. For programs compiled under the ansi77 option, they are ignored by default and thus have no numerical significance. For programs compiled under the ansi66 option, they are treated as zeros by default. Plus signs may be omitted. A field of all blanks is considered to be zero.

For output conversion, the field contains a right-justified, possibly signed, integer constant whose value is that of the associated list element. All leading zeros are suppressed. The value 0 is represented as a single digit, right justified within the field. In the case of the iw.m descriptor, the integer consists of at least m digits and contains leading zeros if necessary. The value of m must not exceed the value of w. If m is zero and the value of the associated list element is zero, the output field will consist entirely of blanks, regardless of the sign control in effect.

Examples:

Format	Input	Output Value
i5	+Ø1Ø0	ØØ100
i7	ØØØØ-10	ØØØØ-10
2i4	ØØØØØØ-1	ØØØ0ØØ-1

FLOATING-POINT CONVERSION VIA fw.d

The numeric field descriptor fw.d describes a field w characters wide that contains the character-string representation of a floating-point value. The list element associated with this field descriptor must be a real value, a double-precision value, or the real or imaginary part of a complex value.

For input conversion, the field consists of an optional sign followed by a string of digits optionally containing a decimal point, and optionally followed by an exponent. An exponent may be a signed integer constant, or an e, E, D, or d followed by an optionally signed integer constant. The value in the field is converted to the mode of the list element.

For output conversion, the field contains a floating-point constant, possibly signed, without an exponent, whose value is that of the associated list element rounded to d fractional digits. All leading zeros to the left of the units position of the output are suppressed.

See "Scale Factor Effects" below, for the effect of scale factors on _w.d input/output conversions.

Examples:

Format	Input	Output Value
f7.3	0123456	123.456
	123.456	123.456
f10.3	ØØØØ123456	ØØØ123.456
f12.6	Ø5000000e+02	ØØ500.000000

FLOATING-POINT CONVERSION VIA ew.d, dw.d

The numeric field descriptor ew.d or dw.d describes a field w characters wide that contains the character-string representation of a floating-point value. The list element associated with this field descriptor must be a real value, a double-precision value, or the real or imaginary part of a complex value.

For input conversion, this field descriptor is equivalent to an fw.d descriptor.

For output conversion, the field contains:

$-0.x_1...x_dE$ or $-.x_1...x_dE$ for negative values
$0.x_1...x_dE$ or $.x_1...x_dE$ for positive or 0 values

where x_1 through x_d are the most significant d digits of the value rounded and E is an exponent of the form E±yy or D±yy where yy is a 2-digit decimal exponent. The digit represented by x_1 will be nonzero unless the value is zero, the exponent is -38, or scaling (see below) is used.

The leading 0 is produced only if there is sufficient room within the field specified by w.

See "Scale Factor Effects" below for the effect of scale factors on ew.d and dw.d input/output conversions.

Examples:

Format	Input	Output Value
e15.8	ßßßßßßßßßß10000	ß0.10000000E-03
	ßßßßß14398624.ß	ß0.14398624E+08
	ßßßßß1439.86241	ß0.14398624E+04

FLOATING-POINT CONVERSION VIA gw.d

The numeric field descriptor gw.d describes a field w characters wide that contains the character-string representation of a floating-point value. The list element associated with this field descriptor must be a real value, a double-precision value, or the real or imaginary part of a complex value.

For input conversion, this field descriptor is equivalent to an fw.d descriptor.

For output conversion, the character-string representation of the value depends on the magnitude of the value. Let n be the magnitude of the value. The following describes the output conversion performed in terms of an fw.d or ew.d field descriptor:

Magnitude Of Data	Equivalent Conversion Effected
n<0.1	ew.d
0.1≤n<1	f(w-4).d,4x
1≤n<10	f(w-4).(d-1),4x
.	.
.	.
.	.
10**(d-2)≤n<10**(d-1)	f(w-4).1,4x
10**(d-1)≤n<10**d	f(w-4).0,4x
10**d≤n	ew.d

Examples:

Format	Input	Output
g10.3	123456.789	123456.8
g14.2	0.0123456789	0.1234568E-01
g14.7	12345678.9	0.1234568E+08

FLOATING-POINT CONVERSION VIA ew.dee

The numeric field description ew.dee describes a field w characters wide that contains the character-string representation of a floating point value. The list element associated with this field descriptor must be real, double precision, or the real or imaginary part of a complex value.

For input conversion, this field descriptor is equivalent to fw.d.

For output conversion, this field descriptor is the same as ew.d, except that the exponent field contains e digits, with leading zeros if necessary.

FLOATING-POINT CONVERSION VIA gw.dee

The numeric field descriptor gw.dee describes a field w characters wide that contains the character-string representation of a floating point value. The list element associated with this field descriptor must be real, double precision, or the real or imaginary part of a complex value.

For input conversion, this field descriptor is equivalent to fw.d.

For output conversion, this field descriptor is the same as gw.d, except that when the magnitude of the value to be converted is less than 0.1 or greater than 10**d, a conversion equivalent to ew.dee is performed.

A complex list element requires a pair of floating-point (f,e,d,g) field descriptors. The first of these refers to the real part of the complex value and the second refers to the imaginary part.

Example:

```
2f10.5
```

SCALE FACTOR EFFECTS

The character p indicates a scale factor (n or -n) to be applied to f, e, g, and d conversions. The form used is:

np

where n, the scale factor, is an optionally signed integer constant.

When the format control is initiated at the beginning of each input or output statement, a scale factor of 0 is established. Once a scale factor has been established, it applies to all subsequently interpreted f, e, g, and d field descriptors until another scale factor is encountered. The scale factor n affects the appropriate conversions in the following manner:

1. For f, e, g, and d input conversions (provided no exponent exists in the external field) and f output conversions, the scale factor n is used as follows:

   externally represented number equals internally represented number times the quantity 10**n

   This means that the input value is divided by 10**n to obtain the internal value, but that the internal value is multiplied by 10**n to obtain the output value. Note that in all the above input conversions and for f output conversion the value of the number is changed.

2. For f, e, g, and d input, the scale factor has no effect if there is an exponent in the external field.

3. For e and d output, the real constant part of the output quantity is multiplied by 10**n and the exponent is reduced by n. Note that for e and d conversions the appearance of the output is changed but not the value.

   The output field will be filled with stars if the scale factor is less than or equal to -d (i.e., there are no significant digits) or greater than d+1 (i.e., there are too many significant digits for the field).

The scale factor n modifies e and d output as follows: If n≤0, there are exactly -n leading 0's and d+n significant digits after the decimal point. (Therefore, a negative scale factor reduces the number of significant digits printed.) If n>0, there are exactly n significant digits to the left of the decimal point and d-n+1 significant digits to the right of the decimal point. (Therefore, a positive scale factor prints d+1 significant digits.) The scale factor must be greater than -d (otherwise there would be no significant digits) and less than d+2 (otherwise there would appear to be more than d+1 significant digits).

4. For g output, the effect of the scale factor is suspended unless the magnitude of the data to be converted is outside the range that permits the effective use of f conversion. If the effective use of a conversion is required, the scale factor has the same effect as with e output.

Examples:

Format	Input	Actual Internal	Output Without Scaling	Scaled Output
-3p f7.3	ß123456	123456.	123456.	123.456
-3p e12.4	ßßßßßß123456	12345.6	ßß0.1234E+05	ßß0.0001E+08
1p d10.4	ßßß 12.3456	1.23456	ß.1235D+01	1.2346D+00

NOTE: An f7.3 field descriptor cannot be used to output the value 123456 without scaling. It is included in this example only to show the steps in scaling the number.

CHARACTER-STRING FIELD DESCRIPTOR

The field descriptor aw describes a field w characters wide. The field descriptor a may be used without any width specifier. This is equivalent to the aw descriptor, with w being determined by the list element associated with the field descriptor. The list element associated with these two field descriptors may be of any mode.

On input, the bit-string representation of the rightmost n characters of the field are stored in the list element. (If the list element is a character-string variable, n is the length of the variable. If the list element is a double-precision or complex variable, n is 8. If the list element is not character-string, double-precision, or complex mode, n is 4.) In the case where w is specified, if the field is less than n characters wide, the w characters are stored left justified with n-w trailing blank characters in the list element.

On output, when w is specified, the bit-string content of the list element is right justified in the output field and the remainder of the field is filled with blank characters. If the field is less than n characters wide, the output field consists of the leftmost w characters from the list element if w has been specified. The bit-string content of a variable output via an aw or a field descriptor must represent valid ASCII characters.

Examples:

```
a4
10a8
```

The field descriptor rw functions exactly like an aw field descriptor, except that on input it stores its data right justified within the list element and fills the remainder of the element with 0 bits; on output, it takes the rightmost w or n characters from the list element.

Examples:

```
r4
2r10
```

OCTAL-STRING FIELD DESCRIPTOR

The octal-string field descriptor ow describes a field w characters wide that contains a string of octal digits. The list element associated with this field descriptor may be of any mode.

On input, the field must contain a right-justified string of octal digits. The last m digits, padded with zeroes if necessary, are stored right justified in the variable (m is 24 for double-precision list elements; otherwise, it is 12).

On output, the bit-string representation of the variable is converted to an octal string of m digits and the rightmost w (if $w \leq m$) digits are stored in the output field. If w>m, m digits are stored right justified in the output field and the remainder of the field is blank.

Examples:

```
o5
3o12
```

LOGICAL FIELD DESCRIPTOR

A logical field descriptor lw describes a field w characters wide that contains the character-string representation of a logical value. The character-string representation of a logical value is either T, F, t, or f. The list element associated with the field descriptor must be a logical variable.

On input, the field is examined from left to right (one character at a time) until the first nonblank character is encountered. If that character is a t or T, the value .true. is transmitted to the list element. If the first character is an f or F, the value .false. is transmitted to the list element.

Output consists of the letter T or F, right justified with preceding blanks in a field whose width is w.

Examples:

```
l8
215
```

Repeat Groups

A repeat group consists of a parenthesized format specification optionally preceded by an unsigned, nonzero, integer constant known as a repeat factor. During the interpretation of a format specification, the contents of the repeat group are reused as many times as are indicated by the repeat factor. If the repeat factor is omitted, the contents of the group are used once. Repeat groups may be nested to a maximum depth of three.

Example:

```
5x,i10,2(3x,"a(i)=",f10.3),"sum=",e12.4
```

is equivalent to:

```
5x,i10,3x,"a(i)=",f10.3,3x,"a(i)=",f10.3,"sum=",e12.4
```

Interaction Between Format and Input/Output List

To process an element of a data transfer list, the format is scanned in the steps given below. If an input statement is being executed, these steps apply to the record read; if an output statement is being processed, they apply to the record being created.

1. If a control item is encountered, the control requests are performed and scanning of the format continues.

2. If a record delimiter is encountered and a read is being processed, the next record is read. If a write is being processed, the current record is written and a new one is begun. Scanning then continues.

3. If a colon is encountered and there is another item in the list, the colon is ignored and scanning is continued; if the list has been exhausted, processing of the format specification terminates.

4. If a field descriptor is encountered and there is another item in the list, the descriptor and item are processed and scanning continues; if the list is exhausted, processing of the format specification terminates.

5. If the end of the format is reached and there are no more items, processing is completed; if there is another item and input processing is being performed, the next record is read and the format is scanned again. If output processing is being performed, the current record is written and a new one begun; the format is scanned again.

6. When the format is being rescanned, control reverts to that repeat group specification terminated by the last preceding right parenthesis or, if none exists, then to the first left parenthesis of the format specification. (This action has no effect on the scale factor.)

In the three examples below, control reverts to the parenthesis immediately preceding 5e15.8:

```
(5e15.8,1x)
(1x,4h,(5e15.8,1x))
(1x/10f12.7,5(4x,a4),6(5e15.8,1x))
```

In the following example, the format labeled 101 produces the output "1,2,3,4,5"; the format labeled 102 produces "1,2,3,4,5,":

```
    integer x(5)/1,2,3,4,5/
    print 101,x
    print 102,x
101 Format (5(i1:1h,))
102 Format (5(i1,1h,))
```

An empty data transfer list causes the format to be interpreted until a colon or field descriptor is encountered or until the end of the format is reached. If the data transfer list is not empty, the format specification must contain at least one field descriptor.

FORMAT STATEMENT

A format statement is a nonexecutable statement containing a format specification. A format statement must be labeled and can appear anywhere in the text of a subprogram (after any implicit, subroutine, function, or block data statements).

Syntax:

label format s

where label is a statement label and s is a format specification as described above under "Format Specifications."

FORMAT SPECIFICATIONS CONTAINED IN ARRAYS

The character-string representation of a format can be stored in an array by the use of a data statement, a formatted read or encode statement that uses a format containing an aw field descriptor, or through the execution of assignment statements that store characters into the elements of the array.

Example:

```
      dimension k(20)
      read 100, k
  100 format (20a4)
        .
        .
        .
      read k,a,b,c
```

The first read statement reads 80 characters from the user's terminal and stores them into the integer array k. The second read statement reads a, b, and c according to the format contained in k.

LIST-DIRECTED INPUT/OUTPUT

If a format specification consists entirely of the single letter v or an asterisk (*), the format specifies list-directed input/output. If a format specification consists of the characters "s,v" or "s,*" under the ansi66 option or "^1,v" or "^1,*" under either the ansi66 or ansi77 option, the format specifies list-directed input from a file containing line numbers.

List-Directed Input

Execution of a list-directed input operation begins when a record is read from the formatted input file designated by the read statement. (Record marks, which are newline characters in a binary stream file, are treated like blank characters.) The elements of the read statement's data transfer list cause fields to be extracted from the input record. Additional records are read until no more list items remain to be satisfied or a list termination character (either "/" or ";") is encountered in the input stream. If all the list elements have been assigned values or a list termination character is encountered, the remaining characters of the last record read are ignored. A complex list element requires a complex constant of the form (n,n). Character constants may span multiple records (in which case the last character of one record is assumed to immediately precede the first character of the next record).

Each record has the following form:

$f_d[f_d]...$

where each f is a field that is either all blank, contains a single constant, or contains a single constant with a repetition factor of the form r*c, where r is the number of times the constant c is to be read. Each is optionally surrounded by blanks and each d is a comma or one or more blanks. A null field can be specified by the occurrence of consecutive commas. A null field can also contain repetition factor of the form r*, where r is the number of null values to be read.

In programs compiled under the ansi66 option, a null field assigns a 0 to the list element when the list element is an arithmetic variable. If the list element is a character-string variable, a null field assigns a string of blanks to the list element. If the list element is a logical variable, a blank or null field assigns .false. to the list element.

In programs compiled under the ansi77 option, a null field has no effect on the corresponding list element. If the list element is defined already, it remains defined with the same value; if the element is undefined, it remains so.

When the record field actually contains a constant, that constant is assigned to the list element as if the constant were the right side of an assignment statement whose left side was the list element. Unlike the assignment statement, list-directed input does not permit a character-string constant to be assigned to an arithmetic or a logical list element.

Each constant may be any constant that could be written in the text of a subprogram, except:

1. Character-string constants must be enclosed in quotation marks or apostrophes if they span multiple input records or if they contain embedded blanks, the character ":" or the character "/"; the nh form is not allowed.

2. The constant must not contain blanks, unless they are within a quoted character-string constant or unless they immediately follow the e or d in an exponent.

3. Acceptable input for a logical variable is not restricted to just .true. or .false.: any value beginning with an optional period followed by the letter (t) or T is considered to be true, and any value beginning with an optional period followed by the letter f (or F) is considered to be false.

Execution of a list-directed output operation creates a single record that is the catenation of the values to be output, in the order that they are specified in the output statement. The format of such a record depends on whether the ansi66 or the ansi77 option is in effect.

If the ansi66 option is selected, values are output in a fixed format determined by the type of the value: 'i16' format for integer values, 'e16.8' for real values, 'd26.18' for double precision values, '"(",e16.8", "e16.8")"' for complex values, 'l2' for logical values, and 'a' for character values.

If the ansi77 option is in effect, nonnumeric values are output in fixed format ('l1' for logical values and 'a' for character values), while numeric values are output in a variable format (appropriate to the type of the value) that suppresses spaces and insignificant zeroes. A space is inserted in the output record before the first value (to act as carriage control) unless the carriage attribute is in effect on the output unit. A space is also inserted between each pair of consecutive noncharacter values to serve as a separator.

NAMELIST STATEMENT

Syntax:

```
namelist/n/list[/n/list]...
```

where each n is the name of a namelist consisting of list. Each list is a list of array or variable names. For any list that is too long for a single source statement, multiple declarations of the same namelist may be used; such declarations must be in consecutive statements. The lists are merged.

Semantics:

Input

The execution of a namelist formatted read statement that refers to a namelist causes records from the designated input file to be read until the following has been completely processed:

```
$x [f,]...$
```

where x is the name of the namelist. Blanks may be used as separators in addition to or instead of any comma. Each f is a field consisting of:

```
all blanks
  or
name=c
  or
name=clist
```

where name may be subscripted with one or more integer constants.

Each c is:

```
const
  or
k*const
  or
k*
```

or
all blanks

where k is an integer constant; const is any constant that could be written in the text of a program unit; k*c is equivalent to c[,c]...; and k* is equivalent to k occurrences of a comma.

A clist is:

c[,c]...

Each name must appear in the namelist.

If one or more subscripts are given with a name, the name must be an array name and the subscripts must satisfy the constraints of an array reference.

After expansion of all replicated constants (those of the form k*c or k*), the number of constants or blank fields in the constant list must not exceed the number of elements in the named variable.

The assignments indicated by the equal sign (=) are performed as assignment statements, except that a constant list is assigned to an array, by assigning each constant to one element of the array in column-major order; i.e., varying the leftmost subscript most rapidly.

A blank field is considered to be a 0 if the required mode is arithmetic, and to be .false. if the required mode is logical; otherwise, it is a blank character.

Output

The execution of a namelist formatted write statement that refers to a namelist causes the current value of each variable in the namelist to be written into a single record of the form:

$x [f[,f]...]$

where x is the name of the namelist and each f is:

name=constant[,constant]...

The form of f that uses a list of constants is output for each array, while the form using a single constant is output for each variable. The constants are edited in the same format as list-directed output.

If the carriage control attribute is in effect for the unit to which the record is to be written, all commas are replaced by blanks and the $x and trailing $ are removed.

The formatted record is written to the file or device designated by the write statement.

SECTION 6

DECLARATIVE STATEMENTS

A user-constructed name within a FORTRAN subprogram is either explicitly declared to have a particular meaning by its use in some definitive context or assumed, by default, to be a variable name. In general, the same name cannot be used for more than one purpose within the same subprogram.

EXPLICIT DECLARATIONS

A name whose meaning is established by its use in a definitive context is said to be explicitly declared. A name can be explicitly declared in any of the following ways:

1. As an array, if it is used as an array name in an array declaration.

2. As a common block name, if it is used as a common block name in a common statement.

3. As a statement function name, if it is not explicitly declared as an array and it appears as a statement function name in a statement function definition.

4. As a subprogram entry name, if it:

   a. follows the word "subroutine" in a subroutine statement, or

   b. follows the word "function" in a function statement, or

   c. follows the word "entry" in an entry statement, or

   d. follows the word "call" in a call statement, or

   e. appears in the list of an external statement and is not one of the built-in function names, or

   f. is not explicitly declared as an array, statement function or built-in function and is used in an expression immediately followed by a parenthesized list.

5. As a generic or built-in function, if it:

   a. has not been explicitly declared as an array, statement function, or subprogram entry, and

   b. is one of the names listed in Section 7, and

   c. is used as a function name in a function reference or in the list of an external statement.

VARIABLE ATTRIBUTES

A variable has, in addition to its name, characteristics that determine the way in which it can be used. These characteristics are called attributes and are as follows:

mode

storage class

initial value

The mode attribute can be: integer, real, double precision, complex, logical, or character. The mode of a variable is specified in a mode statement or by a FORTRAN convention called implicit typing, described later in this section. Functions and arrays also have associated modes.

The storage class attribute determines the type of storage the variable will occupy. For details on storage classes in Multics FORTRAN see the FORTRAN Users' Guide.

INITIALIZATION

By default, all FORTRAN variables are assigned the initial value of binary zero. (If the no_auto_zero option is specified in a %global or %options statement, automatic values get no default initial value. Such variables contain whatever values happen to be in the storage addresses allocated for them.) This default initial value of zero represents an extension of standard FORTRAN, which defines no initial values for variables. The FORTRAN standard specifies that variables that are not explicitly initialized have undefined values. Programs that depend on default initialization to zero on the Multics system are in error. Initial values can be explicitly assigned in an assignment statement, a data statement, a mode statement, or a %global statement. See Section 2 of the FORTRAN Users' Guide for discussion of related issues.

Variables can be explicitly initialized by a data statement, a mode statement, or an assignment statement. Variables not explicitly initialized are automatically initialized to binary zero except when the no_auto_zero option is in effect. The meaning of this for the different modes is as follows:

data type	default initial value
integer	zero
real	unnormalized representation of zero
double precision	unnormalized representation of zero
complex	unnormalized representation of zero
logical	.false.
character	not a valid character

Common variables are initialized at the beginning of a program run. For local variables, see "Local Storage" above.

IMPLICIT TYPING

If a mode is not explicitly declared for a variable, implicit typing is performed. When implicit typing is used, the mode of the name is determined by its initial letter. Names that begin with any of the letters i through n are given integer mode and all others are given real mode. Capital letters are different letters from lowercase letters. Thus I through N have integer value, but are not the same letters as i through n. These default conventions may be changed by the programmer through the use of the implicit statement. Letters not explicitly changed by the programmer retain their normal signification.

IMPLICIT STATEMENT

Syntax:

```
implicit mode[*len](l[,l]...)[,mode[*k](l[,l]...)]...
```

where each mode is one of the following keywords: integer, real, double precision, complex, logical, or character and each k is an optional unsigned nonzero integer constant. If len is omitted, the asterisk character must also be omitted. If mode is real and the value of len is greater than 7, then mode is translated to double precision. If mode is character, the value of len must be from 1 to 256 and give the number of characters in the string values. If mode is character and len is omitted, the default value is used. For ansi66 it is 8, and for ansi77 it is 1. The value len is ignored for all other cases.

Each l is a single letter or a pair of letters separated by a hyphen (if a pair of letters is used, they must have the same case). The letter on the left must alphabetically precede or be the same as the letter on the right.

More than one implicit statement may appear in a program unit, but a letter can be assigned only one mode and cannot be redefined. Implicit statements must precede all other statements except the block data, function, program, and subroutine statements.

Semantics:

The implicit statement associates a mode with variables according to the first letter of their names. This is useful when variables require a mode other than the one assigned to them by implicit typing and when they do not have the desired mode explicitly assigned to them by a mode statement.

Example:

```
implicit integer(i-n),complex(c),double precision(d,p)
```

MODE STATEMENT

Syntax:

```
mode[*len]e[/i/][,e[/i/]]...
```

where mode is specified by one of the following keywords: integer, real, double precision, complex, logical, or character and len is an optional unsigned nonzero integer constant. If len is omitted, the * character must also be omitted. If mode is specified as real and the value of len is greater than 7, then mode is double precision.

Each e is a list of one or more of the following items: an array declarator, an array name, a variable name, a function subprogram entry name, or a statement function name.

The *len may follow any of the names in the list e, if it is not placed immediately after the mode keyword.

Each i is an optional list of initial values that is assigned to the preceding list, e, according to corresponding list position. When a constant list is supplied, it must have the same number of elements as the preceding name list. If i is supplied, e can only contain array names, array declarators, and variable names. If i is omitted, the slashes must also be omitted. The list must follow the rules given below for the data statement.

If mode is character, it is optionally followed by a * and the length specifier, in the form:

```
character[*len] e[/i/][,e[/i/]]
```

If len is omitted, the * character must also be omitted, and the default length is used. When the ansi66 option is in effect, the default length is 8; when ansi77 is in effect, the default length is 1. If len is specified, it may be an unsigned integer constant whose value is between 1 and 256 inclusive, or an integer constant expression enclosed in parentheses. Additionally, len may be an asterisk enclosed in parentheses if the ansi77 option is in effect.

Semantics:

The indicated mode is associated with the names specified by the list e.

When mode is character, and len is (*), the character length is determined either by the length of the corresponding actual argument for dummy argument names, or by the length of the associated constant value for named constants.

Examples:

```
integer i,j,k(10)
character*10 a,b(7)
character x*5,y*7(10,10)
character *11 a,b*5,c(10,12)*3
```

DIMENSION STATEMENT

Syntax:

```
dimension a[,a]...
```

where each a is an array declaration as described under "Arrays" in Section 3.

Semantics:

Each a is declared as an array with the dimensions indicated.

Example:

```
dimension a(10,15),b(n)
```

The maximum number of dimensions allowed is seven. If, as in the example b(n), a variable name specifies a dimension, the dimension statement must appear in a function or subroutine subprogram and the array name, b, must be a dummy argument of the function or subroutine. The dimension indicator, n, may be a dummy argument of the function or subroutine, or it may appear in a common block.

SAVE STATEMENT

Syntax:

```
save [a[,a]...]
```

where each a is the name of a variable or an array in local storage, or a common block. When a is a common block name, it must be preceded and followed by a slash.

Semantics:

Each a retains its value after the execution of a return statement and is not initialized each time the program unit is entered (only the first time the program unit is entered in a process, or after the program has been terminated or recompiled, or, when using the run command, at the beginning of a run unit). This makes it possible to save the value of a variable that is assigned by a subprogram. In the case of a common block, the entire block retains its value after execution of the return statement. (On Multics, this is always true of common blocks, whether or not they are named in a save statement.) If no a's are given, all allowable items in that program unit are retained and not initialized each time the segment is entered. The save statement changes the storage class from automatic to static for the specified variables. (See the FORTRAN User's Guide for information about storage classes.) The save statement, when it appears, must precede any equivalence statements in a program. This permits a case by case override for variables and arrays designated by the automatic option in a %options or %global statement. A save statement and an automatic statement cannot be used in the same program.

Example:

```
save x,y,z
```

AUTOMATIC STATEMENT

Syntax:

```
automatic e[,e]...
```

where each e_i is an array declarator or an array or variable name.

Semantics:

The variables and arrays specified are assigned locations within automatic storage. That is, storage for the variables is allocated and initialized at each entry into the segment. Values are discarded on execution of a return or end statement. The total size of automatic storage is limited by the size of the stack segment in the user's process. This statement changes the default storage class to static. This permits a case by case override for variables and arrays designated by the save option in a %options or %global statement.

Example:

```
automatic x,y,z(10)
```

COMMON STATEMENT

Syntax:

```
common [/[b]/]a[[,]/[b]/a]
```

where each b is a common block name or is empty. If b is empty, the first two slashes are optional. Each a is a list of array declarators or array and variable names.

Semantics:

The variables in each list a are assigned locations in the common block b. The variables are assigned locations in the order in which they appear in the common statements of the subprogram. The common statement coordinates the naming of variables between the main program and one or more subprograms. It can be used to conserve memory and to allow the sharing of data by means other than an argument list.

If b is empty, it is understood to be the name of a block of common storage known as unlabeled common. Unlabeled common storage differs from labeled common in that its length is not checked and no initial values can be assigned. All subprograms containing declarations of unlabeled common in fact refer to the same block of storage. If b contains a dollar sign ($), the common block is assumed to reside in the named segment at the named offset (or zero if no offset is specified). Each labeled common block must be declared the same length in all program units that declare it. To ensure this, common blocks can be declared in include files (see "The %include Statement" in Section 1). See "Dynamic Linking" in Section 4 of the Multics Programmer's Reference Manual and in Section 1 of the FORTRAN Users' Guide for more information. The length of each common block is normally restricted to that of a segment (261120 words). However, if the vla option (Very Large Array) is specified, each nonpermanent, noncharacter block may be up to 2**24 words long.

If the ansi77 option is in effect, and any of the variables in list a are character mode, then all variables in list a must be character mode. That is, a given common block may not contain both character and noncharacter data.

Examples:

```
common /block1/a,b,c/block2/x,y(10)

common/shared_perm_data$/count,sum,data(100,100)
```

DATA STATEMENT

Syntax:

```
data v/k/[,v/k/]...
```

where each v is a list that can contain variable names, array element names, array names, and implied do-loops. Each k is a list of optionally signed constants, any of which may be preceded by a replication factor consisting of a positive unsigned integer constant followed by an asterisk.

Semantics:

At the time storage is allocated for a variable (i.e., on entry to the corresponding segment), the corresponding constant is assigned to it. This provides a means of initializing variables, especially a group of variables, with one statement. With the exception of octal constants, which are not permitted in assignment statements, the rules for mode conversion of the constants in data statements are exactly the same as those for mode conversion in assignment statements. (See "Assignment Statements" in Section 4.)

Subscripts to arrays can be integer constants or constant expressions. When an array element name appears within the list of an implied do-loop, it can be subscripted by integer constants, constant expressions, or by the index variable of a containing do-loop. The subscripts can take on negative values. The number of subscripts must be either one or equal to the number of dimensions.

The number of variables and array elements in each v must equal the number of constants in k.

A constant written with a replication factor n is equivalent to n occurrences of that constant. An array name corresponds to n constants where n is the total number of elements in the array. An array element corresponds to one constant.

A character-string constant given as the initial value of an arithmetic or logical variable is stored left justified in the storage of the variable. If the corresponding member of v is the name of the first element of an array, then the first element and subsequent elements are set until the constant is expended.

The use of character variables and character substring expressions are allowed in data statements. The substring ranges are integers whose value is determined by constants, constant expressions, or simple expressions that refer to variables that are indexes of implied do-loops.

An octal constant given as the initial value of a double precision or complex, integer or real variable initializes all the storage of such a variable. Octal constants cannot be used to initialize character or logical variables.

Examples:

```
data a,i,c/5.2,10,(-5.0,+7.2)/
data b(1),b(2),b(3)/3*1.45/x,y/3.14,2.17/
data array /1.0,2.0,3.0,4.0,5.0/
integer WW (10)
data WW (1)/"a long string for WW"/
data ((a(j,i), j=1,5) i=1,7) /35*0/
```

The first five elements of the array WW are initialized by the second-to-last data statement.

An implied do-loop corresponds to n*m constants where m is the number of elements specified by the list of the implied do-loop and n is the number of times the implied do-loop is evaluated.

The control values m_1, m_2, and m_3 of implied do-loops used in data statements can be integer constants, constant expressions, or the index variable of a containing do-loop, and these values can be negative. The loop count of an implied do-loop appearing in a data statement must be greater than zero.

Automatic variables initialized with a data statement are initialized when the program run is initiated or whenever a new generation of automatic storage is allocated (when a separately compiled subroutine or function is invoked). Automatic variables of a subroutine or function are not reinitialized when the subprogram is called from another subprogram or a main program that was compiled from the same source segment--i.e., subprograms compiled in one object segment have the same storage.

EQUIVALENCE STATEMENT

Syntax:

equivalence (e,e[,e]...) [,(e,e[,e]...)]...

where each e is a variable or array name. Subscripts are all integer constants or constant expressions, and they may take on negative values. The number of subscripts must be either one or equal to the number of dimensions. Each parenthesized group is known as an equivalence group.

Semantics:

The elements listed in an equivalence group are assigned storage so that they all occupy the same storage location. An array name is considered to be a reference to the first element of the array. This permits the use of more than one name for a single variable.

An equivalence statement cannot alter the relative order of the elements of an array, and cannot cause consecutive array elements to occupy noncontiguous storage.

An equivalence statement cannot alter the relative order of variables within a common block.

Any statements in a subprogram that affect the storage classes of items in an equivalence statement must appear in the subprogram before the equivalence statements (e.g., the save and automatic statements).

An equivalence statement that equivalences an array to a common variable may extend the upper end of the common block, but it cannot extend the lower end. The length of the common block includes the length of the variables equivalenced to it.

If the ansi77 option is in effect, and any of the items in an equivalence group are character mode, then all items in the equivalence group must be character mode. That is, a given equivalence group may not specify both character and noncharacter data.

Example:

```
common /b/x,y,z
dimension r(5)
equivalence (r(1),z)
```

causes common block to contain:

x	y	z				
		r(1)	r(2)	r(3)	r(4)	r(5)
1	2	3	4	5	6	7

but:

```
equivalence (z,r(4))
```

is invalid because r(1) would occupy a location below the lower end of the common block.

The modes of the variables equivalenced together may differ, but the programmer must ensure that the value obtained by a reference to an equivalenced variable is consistent with the declared mode of the variable. When one equivalence group contains variables of mixed modes, the values can be defined only for one mode. Values of variables of other modes are truly undefined. The casual FORTRAN programmer should avoid the use of mixed mode equivalence groups.

Examples of valid equivalence groups:

```
dimension x(10),y(20,20),a(10)
equivalence (i,j,k),(a(5),x)
equivalence (x,y),(x(10),q)
```

Double-precision and complex variables are always allocated on double word boundaries. An equivalence statement that contradicts this requirement is in error.

EXTERNAL STATEMENT

Syntax:

```
external f[,f]...
```

where each f is a subprogram name, a subprogram name followed by "(descriptors)," or is one of the names of a built-in function (listed in Table 7-1).

Semantics:

Each name is declared as a subprogram name. An external statement is necessary when a subprogram name or built-in function name is to be passed as an argument but is not used as a function or subroutine within the calling subprogram.

If a subprogram name is followed by "(descriptors)" and the subprogram name is not the name of any entry point declared in this compilation, then standard Multics descriptors are generated for all arguments passed in a call. Furthermore, argument consistency is not checked during calls to the subprogram.

In general, descriptors are required for external names that are either:

- Multics commands
- PL/I procedures that are declared with options (variable), e.g., ioa_
- PL/I procedures whose declarations contain the character "*"
- external entry points

If the ansi77 option is in effect, standard Multics descriptors are generated automatically if any of the arguments are character mode.

Example:

```
external fun, sqrt, ioa_(descriptors)
```

INTRINSIC STATEMENT

Syntax:

```
intrinsic fun [,fun...]
```

where fun is the name of an intrinsic (built-in) function.

Semantics:

The purpose of this statement is to identify fun as an intrinsic function. This is useful when an intrinsic function is passed as an argument to another subprogram. It distinguishes fun from user-defined functions with the same name, and indicates that that intrinsic function is to be called rather than the user-defined function.

PARAMETER STATEMENT

Syntax:

```
parameter (a=c[,a=c]...)
```

where each a is a symbolic name and each c is a constant expression.

Semantics:

The name a assumes the value of c. This statement provides a means of assigning names to constants. Its mode is determined by the FORTRAN rules for the typing of symbolic names or by the use of an implicit statement or an explicit declaration prior to its first use in a parameter statement. The name a may appear anywhere the constant c could appear, except a may not specify a statement label, or appear within a format specification or where the syntax would be ambiguous. For example, a can be used as an array declarator, as a constant or replication factor in a data statement, as the real or the imaginary part of a complex constant, and so on. The declaration of each a must precede its use in the source text.

If the mode of the name a is arithmetic, the corresponding constant c must be an arithmetic expression consisting of the operators +, -, *, /, **, and operands that are either constants or named constants defined in an earlier parameter statement or earlier in the same parameter statement.

If the mode of the name a is logical, the corresponding c must be a logical constant expression consisting of one of the logical constants .true. or .false., or one including the logical operators .and., .or., or .not., previously defined logical parameters, or constant relational expressions. A constant relational expression, in turn, is an expression consisting of the relational operators .lt., .le., .eq., .ne., .ge., and .gt., and their operands. The operands of constant relational expressions are either constants, previously defined parameters, or constant arithmetic or logical expressions.

If the mode of the name a is character, the corresponding c must be a character expression consisting of the character operator // and operands that are either constants or named constants defined in an earlier parameter statement or earlier in the same parameter statement. Array references, function references, and substring references are not allowed. The // operator is not allowed unless the ansi77 option is in effect. If a has a declared length, the expression will be truncated or padded with blanks on the right, whichever is necessary to get the required length. If a is declared with length (*), its length will be set to the length of the result of the evaluation of the expression c.

A now obsolete form of the parameter statement in Multics FORTRAN is as follows:

```
parameter a=c[,a=c]...
```

where each a is a name and each c is a constant (but not another named constant). The name a assumes the mode and value of c. The name a must not have been explicitly declared. This version of the statement will not be supported after some future release.

SECTION 7

FUNCTIONS

A FORTRAN function reference produces a value that is obtained by performing a series of computations that have been established as the function's definition. In addition to the built-in functions of FORTRAN, a program can contain function references to functions defined by the programmer. Two types of function definitions can be written in FORTRAN--statement functions, described below, and function subprograms, described in Section 8, "Subprograms."

STATEMENT FUNCTIONS

A statement function is a programmer-defined function whose entire definition is expressed by a single statement. The name of a statement function is known only to the subprogram in which it is defined. A statement function is defined by a statement of the form:

```
f(p[,p]...)=e
```

where f is the statement function name and e is an expression. Each p is a name known as a dummy argument. Each dummy argument behaves as a temporary variable. When the statement function is invoked by a reference of the form f(a[,a]...), each expression a is evaluated and assigned to the dummy argument p. The relationship between the modes of p and a must be valid for an assignment statement, except that a must not be a character-string constant if p is arithmetic. After the actual arguments have been assigned to the dummy arguments, the expression e is evaluated and converted to conform to the mode of f. The converted value is the value of the reference f(a[,a]...).

The names used to identify the dummy arguments of a statement function may be used elsewhere within the subprogram as the names of variables.

All statement functions must be defined before the first executable statement of the subprogram.

Example:

```
f(x)=x-3.14
```

BUILT-IN FUNCTIONS

Built-in functions are predefined functions available for use in a FORTRAN program. The arguments of each built-in function must be of a specific mode. No conversions are performed to force an argument to conform to the required mode.

The FORTRAN language defines two classes of built-in functions: external built-in functions and internal built-in functions. The names of external built-in functions can appear in the list of an external statement and can be passed as arguments to a subprogram. The names of internal built-in functions cannot be passed as arguments nor can they appear in the list of an external statement.

GENERIC FUNCTIONS

A generic function name identifies a set of built-in functions. Each reference to a generic function is transformed by the compiler into a reference to one of the built-in functions in the set identified by the generic function name. The compiler makes this transformation by first converting all arguments to the highest mode of any of the arguments. That mode is then used to select a built-in function from the set of built-in functions identified by the generic function name. If no member of the set can accept arguments of the mode of the converted arguments, the program is in error.

For example, the generic function reference min(a,b,c) is transformed by converting a, b, and c to the highest mode of a, b, or c. The generic function reference min(a,b,c) is then replaced by a reference to min0(a,b,c), amin1(a,b,c), or dmin1(a,b,c) depending on the mode of the converted values a, b, and c. If the mode is character, complex, or logical, the program is in error because no member of the set identified by min accepts arguments of these modes.

Table 7-1 lists each generic function name and defines the set of built-in functions represented by that name. Those functions with an asterisk (*) in column E are external and can be passed as arguments and used in external statements. Column A contains the number of arguments that the built-in function accepts. If there is no name in the "Generic Name" column, that built-in function has no generic name.

Function	Definition	A	E	Generic Name	Specific Name	Type of Argument	Type of Function
Type Conversion	Conversion to Integer See Note 1	1		int	int ifix idint	integer real double complex	integer integer integer integer
	Conversion To Real See Note 10	1		real	real float sngl	integer integer real double complex	real real real real real
	Conversion to Double Precision See Note 10	1		dble		integer real double complex	double double double double
	Conversion to Complex See Note 9	1 or 2		cmplx		integer real double complex	complex complex complex complex
	Conversion to Integer See Note 5	1			ichar	character	integer
	Conversion to Character See Note 6	1			char	integer	character
Truncation	See Note 1	1		aint	aint dint	integer real double	real real double
Absolute Value	|a| See Notes 2 and 3 $(ar^2+ai^2)^{1/2}$	1	*	abs	iabs abs dabs cabs	integer real double complex	integer real double real
Nearest Whole Number	(int(a+.5) if a≥0 (int(a-.5) if a<0	1		anint	anint dnint	integer real double	real real double

Table 7-1
Built-In Functions (continued)

Function	Definition	A	E	Generic Name	Specific Name	Type of Argument	Type of Function
Nearest Integer	(int(a+.5) if a≥0 (int(a-.5) if a<0	1		anint	nint idnint	real double	real double
Remaindering	a_1-int(a_1/a_2)*a_2 See Notes 1 and 3	2	*	mod	mod amod dmod	integer real double	integer real double
Transfer of Sign	$\|a_1\|$ if $a_2 > 0$ $-\|a_1\|$ if $a_2 < 0$	2		sign	isign sign dsign	integer real double	integer real double
Positive Difference	a_1-a_2 if $a_1>a_2$ 0 if $a_1\leq a_2$	2		dim	idim dim ddim	integer real double	integer real double
Choosing Largest Value	max(a_1,a_2,...)	≥2		max	max0 amax1 dmax1	integer real double	integer real double
					amax0 max1	integer real	real integer
Choosing Smallest Value	min(a_1,a_2,...)	≥2		min	min0 amin1 dmin1	integer real double	integer real double
					amin0 min1	integer real	real integer
Length of String or Substring		1			len	character	integer
Index of Substring	See Note 7	2	*		index	character	integer
Lexical Comparison	Compares Characters See Note 8	2			lge lgt lle llt	character character character character	logical logical logical logical

Table 7-1
Built-In Functions (continued)

Function	Definition	A	E	Generic Name	Specific Name	Type of Argument	Type of Function
Imaginary Part of Complex Argument	ai See Note 2	1			aimag	complex	real
Conjugate of a Complex Argument	(ar,-ai) See Note 2	1			conjg	complex	complex
Double Precision Product	a1*a2	2			dprod	real	double
Square Root	(a)**(1/2)	1	*	sqrt	sqrt sqrt dsqrt csqrt	integer real double complex	real real double complex
Exponential	e^a	1	*	exp	exp exp dexp cexp	integer real double complex	real real double complex
Natural Logarithm	log(a)	1	*	alog log See Note 4	alog alog dlog clog	integer real double complex	real real double complex
Common Logarithm	log10(a)	1	*	alog10 log10 See Note 4	alog10 alog10 dlog10	integer real double	real real double

Table 7-1
Built-In Functions (continued)

Function	Definition	A	E	Generic Name	Specific Name	Type of Argument	Type of Function
Sine	sin(a)	1	*	sin	sin sin dsin csin	integer real double complex	real real double complex
Cosine	cos(a)	1	*	cos	cos cos dcos ccos	integer real double complex	real real double complex
Tangent	tan(a)	1	*	tan	tan tan dtan	integer real double	real real double
Arcsine	arcsin(a)	1		asin	asin asin dasin	integer real double	real real double
Arccosine	arccos(a)	1		acos	acos acos dacos	integer real double	real real double
Arctangent	arctan(a)	1	*	atan	atan datan	real double	real double
	arctan(a_1/a_2)	2	*	atan2	atan2 datan2	real double	real double
Hyperbolic Sine	sinh(a)	1	*	sinh	sinh sinh dsinh	integer real double	real real double
Hyperbolic Cosine	cosh(a)	1	*	cosh	cosh cosh dcosh	integer real double	real real double
Hyperbolic Tangent	tanh(a)	1	*	tanh	tanh tanh dtanh	integer real double	real real double

Notes

1. For a of type integer, int(a) = a. For a of type real or double precision, int (a) is the value obtained by truncating the fractional part of a. For example,
int(3.7) = 3
int (-3.7) = -3
In other words, the truncation is always toward zero. For a of type complex, int(a) is the value obtained by truncating the real part of a toward zero. The functions aint and dint perform the same truncation, but do not convert the truncated value to integer.

2. A complex value is expressed as an ordered pair of real values (ar, ai) where ar is the real part and ai the imaginary part.

3. Only dmod and cabs of their respective groups are external.

4. The generic functions log, log10, alog, and alog10 are not external built-in functions. All the specific names, including alog and alog10, are external built-in functions.

5. Returns the ASCII code of the character argument.

6. Returns the character whose ASCII code is passed as an argument.

7. Returns an integer that gives the starting position of the substring represented by the second argument. The first argument represents the string of which the substring is a part. If the substring does not occur, or if the length of the second argument is greater than the length of the first, then zero is returned.

8. Returns a logical value indicating, respectively, whether the first character string is >=, >, <=, or < the second, when they are compared lexically using the ASCII collating sequence.

9. The cmplx function may be used with either one or two arguments. When used with two arguments, both arguments must be integer, real, or double precision. The value of the first argument becomes the real part, and the value of the second argument becomes the imaginary part. When used with a single integer, real, or double precision argument, the value of the argument becomes the real part and the imaginary part is zero. When used with a single complex argument, the value of the argument is the entire complex result.

10. Complex values are converted to real or double precision by converting the real part of the complex number. The imaginary part is ignored.

SECTION 8

SUBPROGRAMS

There are three types of subprograms in FORTRAN: subroutines, functions, and block data subprograms. The type of a subprogram is indicated by its beginning statement. Subroutine subprograms begin with a subroutine statement; function subprograms begin with a function statement; and block data subprograms begin with a block data statement.

Subroutine subprograms are invoked by a call statement in another program unit (the Multics implementation also allows subroutines that take no arguments to be called from command level). Functions are invoked by function references within an expression in another program unit. Control is returned to the caller from a subprogram by the execution of a return statement. A subprogram may not be invoked twice without the execution of a return statement between the invocations. A block data subprogram is not executed and cannot be referenced. It must be compiled with the first program unit that references any variables in its common blocks.

BLOCK DATA SUBPROGRAMS

Block data subprograms initialize labeled common blocks. They may contain only declarative statements. Naming a block data subprogram is optional. The name has no significance for the compiler or dynamic linker, but it may be useful for documentation. If a name is given, it is supplied with the block data statement.

Syntax:

```
block data
```

or

```
block data name
```

Semantics:

A program may contain several block data subprograms. Each block data subprogram may initialize any number of common blocks. A given common block can be initialized within only one block data subprogram. A block data subprogram must be compiled in the same source segment as the first subprogram (in order of execution) that references variables in its common blocks, or it must be bound with the referencing subprogram. The FORTRAN Users' Guide describes the use of the set_fortran_common command when this constraint cannot be satisfied.

DUMMY ARGUMENTS OF SUBPROGRAMS

A dummy argument is a name whose attributes describe the attributes of the actual argument to be associated with it by the execution of a function reference or call statement. The storage address of a dummy argument is the storage address of the actual argument with which it is associated and may vary from one invocation of a subprogram to the next. No statement executed in a subprogram can reference a dummy argument that is not in the argument list of the entry point specified by the subprogram reference.

There are four kinds of dummy arguments; entry dummy arguments, dummy label arguments, scalar dummy arguments, and array dummy arguments.

An entry dummy argument is identified by a name that has been explicitly declared as an entryname by its use in one of the contexts described in Section 6. An entry dummy argument is equivalent to an entryname and may be used as such anywhere in the subprogram.

If an entry dummy argument is used as a function name in a function reference, the actual argument associated with the dummy argument must be a function whose returned value has a mode identical to the mode of the dummy argument.

If the actual argument associated with an entry dummy argument is a function entry or external built-in function, the entry dummy argument must not be used as a subroutine entryname in a call statement. Conversely, if the actual argument associated with an entry dummy argument is a subroutine entry, the entry dummy argument must not be used as a function reference.

A dummy label argument is identified by an asterisk. When label arguments appear in a call statement, they are removed from the argument list and replaced by a single integer argument following all the user-supplied arguments. This argument is initialized to zero before the subroutine is called. The entry point called must have at least one label argument. Since label arguments are removed from the argument list, they need not be in the same argument position as the dummy label arguments and, in fact, only one asterisk is needed regardless of the number of label arguments that appear in the call statement.

If an alternate return statement is executed, the expression is assigned to the dummy argument associated with the special argument and control is returned to the calling program. Following the call statement is a computed go to statement built using the label arguments in the call statement and the value returned. See Section 4 for a description of the computed go to statement. If a normal return statement is executed, the statement following the call statement is executed.

A dummy argument is identified by a name declared as a variable. A dummy argument is completely equivalent to a variable and may be used as such anywhere in the subprogram, except in those declarative statements that establish the storage class of a variable.

The actual argument associated with a dummy argument can be a variable, an array element name, a character-string constant, or an expression whose mode is identical to the mode of the dummy argument. Refer to Section 4 for a description of the use of character-string constants as arguments.

An array dummy argument is identified by a name declared as an array. An array parameter is completely equivalent to an array and may be used as such anywhere in the subprogram, except in those declarative statements that establish the storage class of an array.

Each array dummy argument must correspond to an array name argument or an array element argument of the same mode. The dimensionality of the array dummy argument is constrained only in that it cannot declare an array whose storage is greater than that of the actual array argument. The same subscripts access different elements when used to reference multidimensional arrays of differing dimensions.

Example:

```
dimension x(10,10)              subroutine s(a,m)
      .                         dimension a(m,m)
      .                               .
      .                               .
n=10                                  .
call s(x,n)                           .
      .
      .
      .
      .
```

The values of dummy arguments used as array bounds cannot be altered during the execution of the subprogram. The values of array bounds of dummy arguments are determined by expressions using integer constants and integer variables that are parameters.

SUBROUTINE SUBPROGRAMS

A subroutine subprogram begins with a subroutine statement that explicitly names the subroutine's major entry and defines any dummy arguments required by that subroutine.

Syntax:

```
subroutine e[([d[,d]...])]
```

where e is the name of the subprogram and each d is the name of a dummy argument. The name e must not be used as the name of any other entry point in the compilation. The maximum number of dummy arguments is 63. Any label parameters, used for counting argument list elements, together count as one dummy argument. The absence of arguments may be indicated by a left parenthesis followed by a right parenthesis. Parentheses used for this purpose may be omitted from a subroutine statement.

Semantics:

When the subprogram is invoked, the arguments (if any) specified by the call are associated with the dummy arguments and control is transferred to the first executable statement in the subroutine. Refer to "Dummy Arguments of Subprograms" above for a discussion of dummy arguments.

FUNCTION SUBPROGRAMS

A function subprogram begins with a function statement that explicitly names the function's major entry and defines the dummy arguments required by that entry.

Syntax:

```
[mode [*k]] function f[*k] (d[,d]...)
```

where mode is not specified or is specified by one of the following keywords: integer, real, double precision, complex, logical, or character; and k is an unsigned, nonzero, integer constant. The *k is optional when mode is specified and can appear, if used, in only one of the two positions shown. If mode is specified as real and the value of k is greater than 7, the mode is double precision. For all other cases, k is ignored.

If the mode is specified character, in the form

```
[character[*len]] function f[*len] (d[,d])
```

*len may appear in at most one of the positions shown. The character length may be an unsigned integer constant between 1 and 256 inclusive, or "(*)" if the ansi77 option is in effect. If len is omitted, the default length is used (8 if the ansi66 option is in effect, 1 if the ansi77 option is in effect.) The name of the function f must not be the name of any other entry point in the same compilation. Each d is the name of a dummy argument. The maximum number of dummy arguments is 62.

Semantics:

When the function is invoked by the execution of a function reference, the arguments specified by that function reference are associated with the dummy arguments and control is transferred to the first executable statement of the function. Refer to "Dummy Arguments of Subprograms" above for a discussion of dummy arguments.

If the mode associated with the function name is not specified in the function statement, it may be explicitly specified by the use of the function name in a mode statement or it may be determined by the implicit typing convention of the FORTRAN language. (Refer to "Implicit Typing" in Section 6.)

The value returned by the invocation of a function is determined by an assignment statement whose left side consists of the entryname of one of the entries to the function.

The mode associated with a function entryname describes the mode of the value returned by the function. All entries to a function subprogram must be associated with the same mode. In other words, the mode of the value returned by the invocation of a function is invariant; it does not depend on the entry used to invoke the function.

Example:

```
function f(x)
     .
     .
     .
f=7.0
     .
     .
     .
return
     .
     .
     .
entry e(x)
     .
     .
     .
f=5.0
     .
     .
     .
return
```

The function returns either 7.0 when invoked as f(a) or 5.0 when invoked as e(a).

ENTRY POINTS

The first executable statement in an executable subprogram is called the major entry point of the subprogram. For subroutines and functions, the name of this entry point is supplied with the subroutine or function statement. For a main program, the name of this entry point is main_ or the name supplied with the program statement. (The identifier main_ is a reserved keyword in Multics FORTRAN.)

Secondary entry points can also be defined in subroutine and function subprograms. These are additional points in the subprogram where control can be transferred from other program units. Secondary entry points are defined using the entry statement.

Syntax:

```
entry e [([d[,d]...])]
```

where e is the secondary entryname and each d is the name of a dummy argument. The name e must not be used as the name of any other entry point in the compilation. The maximum number of dummy arguments is 63. Any label parameters, used for counting argument list elements, together count as one dummy argument. The absence of arguments may be indicated by a left parenthesis followed by a right parenthesis. Parentheses used for this purpose may be omitted from an entry statement.

Semantics:

When the subprogram is invoked by the use of the entryname e, the actual arguments (if any) specified by the call or function reference are associated with the dummy arguments specified by this entry and control is transferred to the first executable statement following this entry statement. Refer to "Dummy Arguments of Subprograms" above for a discussion of dummy arguments.

In a multiple entry subprogram, the number of parameters in each entry may differ and the same parameter may appear in different positions within the parameter lists of several entries. It should be noted, however, that if one parameter appears in the parameter lists of several entries, more efficient code is generated if the parameter always appears in the same position.

Example:

```
subroutine x (a,b,c)
     .
     .
     .
entry y (a,d)
     .
     .
     .
entry z (d,c,x,y,z)
     .
     .
     .
     .
```

A secondary entry to a function subprogram must specify at least one dummy argument, and the mode associated with the entryname must be identical to the mode associated with the major entryname.

The association of an actual argument with a dummy argument occurs at the time of the execution of the call statement or function reference. This association provides a storage address for the dummy argument. Any attempt to reference a dummy argument that does not appear in the entry through which control enters the subprogram yields unpredictable results.

SECTION 9

MULTICS FAST SUBSYSTEM ENVIRONMENT

The Multics FAST subsystem is a subset of the full Multics system that supports most of the same FORTRAN command repertoire as Multics. The storage hierarchy and dynamic linking features of the Multics system are available to the Multics FAST user. Under Multics FAST, the user can invoke the compiler explicitly using the fortran command or implicitly using the run command. For details on Multics FAST, see the Multics FAST Subsystem Reference Manual (Order No. AU25).

RUNNING A PROGRAM

If the Multics FAST temporary text contains a FORTRAN source program, issuing the run command causes the program to be compiled and then executed if the compilation succeeds (i.e., if no serious errors are detected during compilation). The name of the temporary text must have ".fortran" as a suffix.

Example:

```
new test.fortran
10 x = 1
    .
    .
    .
100 end
run
```

To compile and execute a FORTRAN source program that has been saved, use the "run prog_name" command, where prog_name is the name of the source program (which must end in ".fortran").

Example:

```
new
10 x = 1
    .
    .
    .
100 end
save test.fortran
run test.fortran
```

To execute a FORTRAN object program that was created by explicit compilation, use the "run prog_name" command, where prog_name is the name of the object program and must not have the ".fortran" suffix.

Example:

```
run test
```

TERMINATION OF A RUN

A program run terminates when control reaches the end of the main program, when a stop statement is executed, when an error is detected, or when a quit signal occurs. All files are closed, and all storage used by the run is released. This includes the storage for the temporary object program produced when a source program is run.

COMPILING A PROGRAM

If a FORTRAN source program has been saved with a name containing the ".fortran" suffix (e.g., xx.fortran), it may be compiled by issuing the fortran command in either of the following ways:

```
fortran xx
  or
fortran xx.fortran
```

If the compilation succeeds, an object program with the name "xx" is saved in the user's working directory.

Example:

```
new
10 x = 1
    .
    .
    .
100 end
save test.fortran
fortran test
run test
```

SEPARATE SUBPROGRAMS

Subprograms may be compiled separately from the main program; that is, they may be in a different source segment. This is advantageous when the same subprogram is to be used with more than one main program, or when some subprograms are changed frequently while others remain the same.

Generally, the name of the source segment for a set of separate subprograms should end with the ".fortran" suffix (e.g., xx.fortran, where xx is the name of one of the subprograms). After compilation, the object segment has the name xx; this enables the system to find the name xx when it is referred to by another object segment. If other subprograms in the object segment are to be called from external segments, their names must be added to the object segment. Similarly, if a subprogram in a compiled main program object segment is to be called from external segments, its name must be added to the main object segment.

To add names to an object segment, use the Multics FAST add_name command:

```
add_name object_segment_name additional_name
```

Example:

```
new
10 x = 1
     .
     .
     .
50 call b
     .
     .
     .
100 end
110 subroutine a
     .
     .
     .
200 end
save test.fortran
fortran test
add_name test a
new
10 subroutine b
     .
     .
     .
50 call a
     .
     .
     .
100 end
save b.fortran
fortran b
run test
```

LINKING

The system uses the following three-step rule to find a named subprogram xx:

1. If the object segment containing the reference to xx also contains a subprogram named xx, use that subprogram.

2. Otherwise, if the main program was run as a source program, and if its segment contains a subprogram named xx, use that subprogram.

3. Otherwise, search the user's working directory for a segment named xx, and search that segment (if found) for a subprogram named xx.

A block data subprogram cannot be linked to by name; thus, it must be compiled with the first program unit to reference the block in the program run.

All names needed to resolve references between separately compiled segments must be found in the user's working directory. However, by means of links, the names in the user's working directory may be connected to segments in other directories. The link command has the form:

```
link complete_pathname entry_name
```

where entry_name is the name of a subprogram and, complete_pathname is the name of an object segment containing that subprogram.

Example:

```
link >udd>Demo>Smith>plot plot
run plot
```

If the object segment contains additional separately callable subprograms, their names should be added to the link.

Example:

```
new
10 x = 1
    .
    .
    .
60 call plot
80 call plotx
100 call ploty
150 end
save test.fortran
link >udd>Demo>Smith>plot plot
add_name plot plotx ploty
run test.fortran
```

RUNNING A SUBPROGRAM

It is possible to run subprograms in Multics FAST. If the command "run f" is executed and f is the object segment containing a subprogram named f, that subprogram is executed as the main program. This is not normal FORTRAN practice, but it succeeds provided the subprogram has no parameters.

WARNING: This feature applies even to object segments containing main programs. Thus, a subprogram name should not be used as the main name of a main program object segment.

Example:

```
new
10 x = 1
    .
    .
    .
100 end
110 subroutine test2
    .
    .
    .
200 end
save test.fortran
fortran test
add_name test test2
run test2
```

When a program run is initiated by running a subprogram, the execution of a return statement in the subprogram terminates the whole run.

PAUSE STATEMENT

Execution of a pause statement in Multics FAST causes the string "PAUSE" or "PAUSE string" to be printed and is otherwise ignored.

RESERVED ENTRY NAMES

The names "main_" and "symbol_table" are used internally by the Multics system and cannot be used as entry point names.

SECTION 10

FORTRAN AND THE MULTICS INPUT/OUTPUT SYSTEM

This section describes the relationship between the input/output features of FORTRAN (described in Section 5) and the Multics I/O system. It provides sufficient information to allow a FORTRAN programmer to understand the Multics I/O system. (The FORTRAN User's Guide provides more information on how to use FORTRAN to do I/O in the Multics system.) A complete description of the Multics I/O system, as well as definitions for some of the terms used in this section, is given in Section 5 of the Multics Programmers' Manual, Reference Guide (Order No. AG91).

Throughout this section, references are made to many of the specifiers and attributes of Multics FORTRAN input/output. For information on these specifiers and how they relate to the attributes, refer to Section 5 of this manual.

FILES AND I/O SWITCHES

A FORTRAN program refers to a file or device by an integer valued unit number, u, where $0 \leq u \leq 99$. There is a file table maintained by the FORTRAN runtime I/O routines, containing 100 entries (0 through 99). The connection of a unit identified by a unit number in a FORTRAN program is recorded in the corresponding file table entry.

Each unit number used in a FORTRAN program run is associated with the following data structure:

```
file table (unit number 0 - 99)
     flags
       connected/disconnected
       fortran_attached/not
       fortran_opened/not
       formatted/unformatted
       prompt/no
       defer/no
       allow_input/no
       allow_output/no
       allow_reopen/no
       allow_direct/no
       allow_sequential/no
       allow_positioning/no
       carriage_control/no
     switch ptr---------->I/O switch (switch name)
     I/O type              file designator-------->Multics file
       stream              attach description      or device
       record              attached/detached
       blocked             opened/closed
       binary stream       opening mode
```

ERRORS AND ERROR MESSAGES

When an error occurs, one of several actions is possible. If the iostat attribute is specified, a standard error code is stored in the designated variable or array element and execution continues with the statement following the statement in which the error occurred unless the err specifier is present. If the err specifier is present, control is passed to the designated statement label. If both are present, the designated variable is always set, although the transfer to a designated statement label occurs only if there is an error. If neither is present, an error message is printed and the program run terminates.

The err and iostat specifiers can always be given in an open statement and are therefore not mentioned explicitly.

Throughout the rest of this section, one of the error processing actions described above is implied by the phrases "an error occurs" or "an error condition exists."

CONNECTING UNITS TO FILES AND DEVICES

The file table maintained by the FORTRAN runtime I/O routines contains information specific to FORTRAN. A FORTRAN file is associated with the file table entry via the unit number in the program. A file table entry--like a unit--is either connected or disconnected. A disconnected file table entry becomes connected when its corresponding unit number is referenced by an open statement or a data transfer statement. When it is connected, it is associated with a Multics file or device through the use of a named I/O switch, and the I/O switch is attached and opened. The user can attach or open the I/O switch explicitly before the program is run or let the FORTRAN runtime I/O routines perform these functions when they connect the unit. (See the FORTRAN Users' Guide for a fuller explanation of connection from the standpoint of the program.) If the user attaches the I/O switch, the FORTRAN runtime I/O routines honor that attachment and choose a compatible opening mode. If the user attaches and opens the I/O switch, both are accepted by the FORTRAN runtime I/O routines. When the file table entry is disconnected, only the functions actually performed by the FORTRAN runtime I/O routines are undone.

The Multics file being referenced can be specified by using the file specifier in an open statement, but only in the case where the FORTRAN runtime I/O routines generate the attach description. If no name is specified, filenn is used.

The user can specify the I/O switch name in an open statement. If it is not specified, the name is filenn, where nn is a two-digit representation of the unit number in the source program.

The attach description used to attach the I/O switch can be specified in an open statement, can be specified externally to the FORTRAN system (e.g., with the io_call command), or is generated by the FORTRAN runtime I/O routines depending on the properties of the Multics file or device being referenced and by the mode, access, and form attributes specified in the connection request.

Unit number 0 is handled separately and the user cannot specify an I/O switch name, an attach description, or a file pathname.

When a file table entry is being connected because of an open statement any of the open statement specifiers can be given, except that a specifier cannot be used if it implies an attribute that conflicts with the actual nature of the connection; conflicting specifiers must not be given. For example, the file and attach specifiers are mutually exclusive, as are binary stream and recl. Also, the use of file or attach is invalid if the I/O switch is already connected.

Details of Connection

Three major steps are required to connect a unit. The first is to attach the I/O switch; the second is to open the I/O switch; and the third is to assign the appropriate attributes. The FORTRAN runtime I/O routines are prepared to do as much of this as is necessary. In particular, if the I/O switch is not attached, the FORTRAN runtime I/O routines attach the I/O switch. If the I/O switch is not open, the FORTRAN runtime I/O routines open the I/O switch with an opening mode derived from the access, form, and mode attributes specified by the open statement as well as by the properties of the associated Multics file. If the program contains no open statement, the necessary attributes are established by default (see "Opening" below). Attaching and opening are caused either by an open statement or by a data transfer statement when the unit is not open.

ATTACHING

The FORTRAN runtime I/O routines use the following algorithm:

1. If the unit number is 0, the file table entry is always connected and only the prompt=, carriage=, and defer= specifiers are allowed. The appropriate attributes are changed and no further action is taken. The connection is complete.

2. If the file table entry is already connected, only the recl=, mode=, access=, prompt=, defer=, and carriage= specifiers are allowed.

3. If the user supplies the name of an I/O switch, use it. Otherwise the FORTRAN runtime I/O routines use file*nn* (e.g., file03).

4. If the I/O switch is already attached, the FORTRAN runtime routines honor that attachment. If the user specifies an attach or file specifier, an error occurs. Continue with "Opening" below.

5. If the user supplies an attach description, the FORTRAN runtime I/O routines attach the I/O switch using it. If the user supplies the file specifier, an error occurs. Continue with "Opening" below.

6. If the user supplies the name of a file, the FORTRAN runtime I/O routines use it. Otherwise, they use file*nn*.

7. The FORTRAN runtime I/O routines generate an attach description and then attach the I/O switch, choosing the first case description that exactly matches the connection request, and attach as specified. (If a filename is needed, FORTRAN I/O uses the one derived from step 6.)

   Case 1: If only form, access, and mode are specified (as formatted sequential input), and the unit has the default input attribute, then the FORTRAN runtime I/O routines attach with:

   ```
   syn_ user_input -inhibit put_chars
   ```

Case 2: If only form, access, and mode are specified (as formatted sequential output), and the unit has the default output attribute, then the FORTRAN runtime I/O routines attach with:

```
syn_ user_output -inhibit get_line get_chars
```

Case 3: If the binary stream attribute is specified, then the FORTRAN runtime I/O routines attach with:

```
vfile_ filename -no_trunc
```

Case 4: If a maximum record length, len, is specified, then the FORTRAN runtime I/O routines attach with:

```
vfile_ filename -blocked len -no_end
```

Case 5: If the external file is a vfile_ blocked file, then the FORTRAN I/O runtime routines attach with:

```
vfile_ filename -no_end
```

Case 6: Otherwise the FORTRAN runtime I/O routines, attach with:

```
vfile_ filename
```

OPENING

If the I/O switch is not open, the next step is to open the I/O switch with the appropriate iox_ opening mode. Table 12-1 gives the modes of opening for iox_ attempted for each combination of file type (including nonexistent and unknown), and the form, access, and mode attributes. The mode inout in an open statement is treated as the mode in if the file exists and is not zero length. It is treated as out if the file does not exist or is zero length.

If any of the access, form, or mode attributes is not specified when a file table entry is connected, its default value is used to connect the file table entry. The defaults are:

```
access="sequential"
form="unformatted"
mode="inout"
```

(The default value of form= is, however, formatted if the ansi77 option is in effect and unformatted anytime that access="direct".)

If the I/O switch is already opened, the iox_ opening mode must be one of those supported by the Multics I/O system, and this mode is used to determine the type of I/O supported by the I/O switch.

If the mode is out or inout, the I/O module is vfile_, and the file does not exist or is a zero length segment, a file is created depending on the access and form supplied in the open statement. A vfile_ unstructured file is created for formatted sequential, a vfile_ sequential file is created for unformatted sequential, and a vfile_ indexed file is created for both formatted and unformatted direct access files.

Table 10-1. Opening Modes Used by FORTRAN

	INPUT				OUTPUT			
	SEQUENTIAL		DIRECT		SEQUENTIAL		DIRECT	
File Type	FMT	UNF	FMT	UNF	FMT	UNF	FMT	UNF
Nonexistent or Zero Length	TIN	QIN	KIN	KIN	TIO	QIO	KUP	KUP
vfile_ Unstructured	TIN	X	X	X	TIO	X	X	X
vfile_ Sequential	QIN	QIN	X	X	QIO	QIO	X	X
vfile_ Indexed	QIN	QIN	KIN	KIN	X	X	KUP	KUP
vfile_ Blocked	QIN	QIN	QIN	QIN	QUP	QUP	QUP	QUP
vfile_ Binary Stream	X	TIN	X	TIN	X	TIO	X	TIO
tape_mult_ Binary Stream	X	TIN	X	TIN	X	TOT	X	TOT
Unknown	TIN QIN	QIN	KIN DIN	KIN DIN	QIO QUP QOT TIO TOT	QIO QUP QOT	KUP KOT DUP DOT	KUP KOT DUP DOT

Opening mode abbreviations:

TIN - stream_input
TOT - stream_output
TIO - stream_input_output

QIN - sequential_input
QOT - sequential_output
QIO - sequential_input_output
QUP - sequential_update

KIN - keyed_sequential_input
KOT - keyed_sequential_output
KUP - keyed_sequential_update

DIN - direct_input
DOT - direct_output
DUP - direct_update

X - Error. Cannot be opened.

If the mode is in and the I/O module is vfile_, a file must exist in the storage system and be of a type compatible for the access and form or an error occurs when the I/O switch is opened. The vfile_ I/O module considers a zero length file to be an unstructured file when opened for input.

If the I/O module is vfile_ and the iox_ opening mode is stream_input_output or sequential_input_output, the file is truncated (i.e., its contents are lost) when it is opened unless one of the control arguments -extend, -append, or -no_trunc is supplied in the attach description. This is a feature of the vfile_ I/O module. Each of these control arguments prevents the truncation of the current contents of the file while providing its own set of side effects. While all three are described in detail in the description of the vfile_ I/O module in the MPM Subroutines, a brief description is in order here. The -extend control argument causes the file to be positioned to the end of the file when opened for output or input/output. It is ignored if the file is opened for input. The -append control argument causes all output data transfers to be appended to the current contents of the file. When opened, the file is positioned to the beginning of the file. The -no_trunc control argument prevents vfile_ from truncating the contents of the file that follow the output data transfer. When the file is opened, the file is positioned to the beginning of the file.

If the binary stream attribute is specified, the associated storage system file must be a vfile_ unstructured file or it must be attached with an I/O module other than vfile_, such as tape_mult_.

If the maximum record length attribute is specified, the associated storage system file must be a vfile_ blocked file or it must be attached with an I/O module other than vfile_.

ASSIGN UNIT ATTRIBUTES

Unit attributes, as supplied by the open statement or implied by data transfer statements, are compared to the actual attributes of the file or device and must be consistent. The user can supply the following attributes:

- I/O Switch Attribute

  See Section 5 of the MPM Reference Guide

- Attachment Attribute

  See Section 5 of the MPM Reference Guide.

- Filename Attribute

  If none is specified in attach description or open statement, filenn.

- Mode Attribute

  The mode attribute is processed only if it is specified or if the file table entry is being connected. If it is not specified when the file table entry is being connected, i.e. in an open statement, its default value is used (implicit connection via data transfer). This attribute can be specified with any opening, except those referencing unit number 0.

  If the FORTRAN runtime I/O routines do not open the I/O switch, the opening mode of the I/O switch must at least support those modes the user has requested. Only the mode(s) specified in the open statement are allowed in subsequent data transfers, even if the actual I/O switch opening mode is less restrictive.

  If the file table entry is being connected and the FORTRAN runtime I/O routines open the I/O switch, Table 10-1 indicates the opening mode used. If the mode is in, the I/O switch is opened for input only. If the mode is out, the I/O switch is opened for input/output, update or output only, and the FORTRAN runtime I/O routines only allow output data transfers. If the mode is inout or the connection is implied by a data transfer statement, the file table entry acquires the allow_reopen attribute. This attribute allows the I/O switch to be reopened, as necessary, to support the inout mode. This attribute allows the FORTRAN runtime I/O routines to:

  1. support inout on files and devices that are unidirectional (many tape I/O modules are limited to input only and output only openings), and

  2. prevent the vfile_ I/O module from truncating a file opened for stream_input_output or sequential_input_output. To prevent premature destruction of the file, it is opened for stream_input or sequential_input. The first write request to the unit reopens the I/O switch.

  Finally, if the file table entry is already connected, the FORTRAN runtime I/O routines have opened the I/O switch, and the user specifies a new (and different) mode, the mode is processed as follows. If the new mode is inout, the file table entry acquires the allow_reopen attribute. (Refer to the previous paragraphs.) No other processing is required at this time. If the new mode is in or out and the current I/O switch opening does not support the new mode, the I/O switch is reopened. If the I/O switch cannot be reopened, the original opening is restored and an error occurs. The file is restricted to data transfers of the mode specified.

- Access Attribute

  The access attribute is processed only if it is specified or if the file table entry is being connected. If it is not specified when the file table entry is being connected, its default value is used. This attribute can be specified with any opening, except those referencing unit number 0.

The external file must support the access requested by the user. The FORTRAN runtime I/O routines do not check that the file does in fact support the access requested.

If the access is sequential, direct access read and write statements are not permitted to the file.

If the access is direct, then the backspace, rewind, sequential read, and sequential write statements are not permitted to the file.

If the access is direct, the external file cannot be: the terminal, a vfile_ unstructured file, a vfile_ sequential file, a tape attachment, or a discard_ attachment.

For vfile_ blocked files and binary stream files, if the access is sequential, the normal restrictions apply. If the access is direct, only the backspace and rewind statements are not permitted to the file.

- Form Attribute

  The form attribute can be specified only when a file table entry is being connected. If it is not specified, its default value is used. This attribute cannot be specified with any opening referencing unit number 0.

  If the form is formatted, the external file cannot be a binary stream file.

  If the form is unformatted, the external file cannot be a vfile_ unstructured file or the terminal.

- Maximum Record Length

  The maximum record length attribute is only processed if it is specified. This attribute can be specified with any opening, except those referencing unit number 0.

  If the external file exists prior to connection, it must be a vfile_ blocked file. If the external file is not a vfile_ blocked file, an error occurs.

  If the file contains records, this value must equal the maximum record length value used to create those records.

- Binary Stream Attribute

  If the external file exists prior to connection, it must be a vfile_ unstructured file or the I/O module used to attach it must not be vfile_. The form of the file must be unformatted.

  This attribute can be supplied only when connecting a unit. It cannot be supplied if the unit is already connected or the unit number is 0.

- Prompt Attribute

  This attribute can be supplied with any opening. If it is not supplied, it retains the value it had prior to the execution of the open statement. Once this attribute is associated with a unit it remains associated with all subsequent openings during the life of the process, until explicitly changed.

- Carriage Control Attribute

  This attribute can be supplied with any opening. If it is not supplied, it retains the value it had prior to the execution of the open statement. Once this attribute is associated with a unit it remains associated with all subsequent openings for the life of the process, until explicitly changed.

- Defer Newline Attribute

  This attribute can be supplied with any opening. If it is not supplied, it retains the value it had prior to the execution of the open statement. Once this attribute is associated with a unit it remains associated with all subsequent openings during the life of the process, until explicitly changed.

SECTION 11

EXAMPLES

This section gives an example of a FORTRAN program. The example consists of a main program that is a random number generator (it produces a bill and determines payment). Then it prints out the change due.

CHANGE MAKER

This program is called change. The line numbers in this example are not part of the source text and have been included to make referencing easier. This example illustrates the use of formatted output and a FORTRAN subprogram referencing a PL/I procedure (random_ in lines 24 and 26).

The subprogram could be made interactive by replacing lines 24 and 25 with a read statement for namt and replacing lines 26 through 28 with a read statement for npay.

The namelist "bugs" was included in this subprogram to aid in debugging. The payment can only be less than the amount if there is a logic error. In the interactive version, the namelist group would probably not be needed.

```
 1  c     Change maker test program
 2
 3  c     Declarations
 4
 5        dimension val(10), names(20)
 6        integer val
 7        character*12 names /"Fifties", "Fifty", "Twenties", "Twenty",
 8       &    "Tens", "Ten", "Fives", "Five", "Dollars", "Dollar",
 9       &    "Half Dollars", "Half Dollar", "Quarters", "Quarter",
10       &    "Dimes", "Dime", "Nickels", "Nickel", "Pennies",
11       &    "Penny"/
12
13        namelist /bugs/ a_new_number, what_change, namt, npay, mc
14        data (val(i), i=1, 10)/5000, 2000, 1000, 500, 100, 50, 25,
15       &    10, 5, 1/
16
17  c     This program calls a random number generator to get an amount
18  c     of purchase on the interval $10.00 to $0.50.
19  c     The same number generator is called to determine how payment
20  c     was made--exact payment or change due.
21  c     Make change ten times.
22
```

```
23          do 100 item = 1, 10
24              call random_$uniform(a_new_number)
25              namt = 1000 * a_new_number + 50
26              call random_$uniform(what_change)
27              i = what_change * 9
28              npay = (namt+val(i+1)-1) / val(i+1) *val(i+1)
29              pay = float(npay) / 100.
30              amt = float(namt) / 100.
31              mc = npay - namt
32              c = mc
33              c = c/100.
34              print 51, amt, pay
35              if (c .ne. 0) print 52, c
36  51          format("0Amount of bill is $", f5.2/t9,  "Payment is $", f5.2)
37  52          format("   Total change is $", f5.2)
38              if (c) 60,70,50
39  70          print, "No change."
40              go to 100
41  60          print, " Program error.  Payment is less than amount."
42              print bugs
43              go to 100
44
45  c       Now figure out the actual
46  c       pieces of change--what denominations and
47  c       how many of each.
48
49  50          j = 1
50  15          do 27 i = j,10
51                  if (val(i) .le. mc) go to 5
52  27              continue
53              go to 100
54  5           n = mc/val(i)
55              ipos = i * 2 - 1
56
57  c       Use correct English (singular or plural).
58  c       The next statement selects the right word.
59
60              if (n .eq. 1) ipos = ipos + 1
61              print 53, n, names(ipos)
62  53          format(i6, 1x, a12)
63
64  c       Now compute the change on the amount
65  c       remaining.  If none, we are done.
66
67              mc = mc - n*val(i)
68              if (mc .le. 0) go to 100
69              j = i + 1
70              go to 15
71  100         continue
72          end
```

A portion of one run of the change maker program is shown below. The exclamation mark denotes a line typed by the user.

```
!  change

   Amount of bill is $ 2.73
          Payment is $10.00
     Total change is $ 7.27
       1 Five
       2 Dollars
       1 Quarter
       2 Pennies

   Amount of bill is $ 9.14
          Payment is $ 9.25
     Total change is $ 0.11
       1 Dime
       1 Penny

   Amount of bill is $ 1.01
          Payment is $ 1.10
     Total change is $ 0.09
       1 Nickel
       4 Pennies

   Amount of bill is $ 3.70
          Payment is $ 3.70
   No change.

   Amount of bill is $ 2.40
          Payment is $50.00
     Total change is $47.60
       2 Twenties
       1 Five
       2 Dollars
       1 Half Dollar
       1 Dime
```

APPENDIX A

FORTRAN COMPARISON

Table A-1 compares various features of two versions of FORTRAN: Multics and the American National Standard (1977).

Table A-1 does not attempt to list and describe every FORTRAN feature but concentrates on those features that are different among the two FORTRANs and also those features that extend the standard FORTRAN.

Table A-1. Comparison of FORTRAN Features

FORTRAN Features	Multics	Standard FORTRAN
GENERAL INFORMATION		
character set	ASCII	ASCII
distinction between uppercase/lowercase	yes[a]	no
character variables and functions	yes[b]	yes
line length	no limit	72
maximum statement length	1000 tokens or 1320 chars[c]	19 card images
continuation convention	free form or standard	nonzero, non-blank in col 6
statements per line	multiple	one
comment lines	c, C, or * in col 1	c/C in col 1[d]
end statement	end/END	end/END
variable name length	1 to 256	1 to 6
underscore character in variable name	yes	no
character *(*) function	yes	yes
variable-expression array bounds	yes	yes
STATEMENT ORGANIZATION		
declarative statements can be mixed with executable statements	no	no
implicit statement can appear anywhere	no	no
FORMAT SPECIFICATIONS		
for n slashes at end of specification:		
total records read	n+1	n+1
total records written	n+1	n+1
carriage-control characters implemented for terminal files	blank,1,0,+	not defined
format field descriptors:		
o	yes	no
r	yes	no
v	yes	no
s[e]	yes	no
:	yes	yes
STATEMENT FUNCTIONS		
formal parameters specify mode, order, and number of arguments	yes	yes
mode of function and its defining expression are the same	yes	yes

Table A-1 (cont). Comparison of FORTRAN Features

FORTRAN Features	Multics	Standard FORTRAN
DYNAMIC LINKING		
link overlay	no(f)	not defined
common blocks initialized by first program to reference them	yes	not defined
common block storage can be a permanent segment	yes(g)	no
Array Storage Limits	2**24	not defined
EXTENSIONS		
maximum number of array dimensions	7	3
abnormal statement	no	no
automatic statement	yes	no
decode statement	yes	no
encode statement	yes	no
entry statement	yes	yes
implicit statement	yes	yes
mode statement allows mode *n	yes	yes(h)
mode statement allows data initialization	yes	no
namelist statement	yes	no
namelist I/O statement	yes	yes
parameter statement	yes	yes
punch statement	yes	no
terminal read statement	yes	no
direct access I/O statements	yes	yes
alternate return statement	yes	yes
typeless expressions	no	no
in arithmetic if statement	no	no
character-string constants	yes	no
interpratation of x**y**z as expression	x**(y**z)	x**(y**z)
runtime symbol table can be generated	yes	no
dynamic debugger	yes	no

NOTES: a. The case of a letter depends on the control arguments used by the compiler; -fold and -card both map all uppercase letters into lowercase letters unless they appear within a character-string constant.

b. The FORTRAN 77 Standard allows this mode statement for characters but not for other data types. Multics FORTRAN allows it for all data types.

c. A token is a constant, name, operator, delimiter, or label argument. For example, a statement consisting of 1000 1-character tokens requires at least 15 card images (66 columns to a card image) and a statement consisting of 1000 2-character tokens requires at least 30 card images.

d. The FORTRAN standard does not specify the case (i.e., uppercase or lowercase) of the alphabetic characters.

e. The s control item has different meanings under the ansi66 and ansi77 options. The ^1 control item is used in ansi77 to perform the function s performs in ansi66, and s performs another function in ansi77. See "Format Specifications" in Section 5.

f. There is no need for a LINK overlay because Multics dynamically links all external references.

g. Permanent common storage is not allowed when using Very Large Arrays.

h. The maximum length of a character-string value is 256 characters. The maximum length of a character-string constant is 256 characters.

APPENDIX B

DIFFERENCES BETWEEN ANSI66 AND ANSI77

The differences between the options of ansi66 and ansi77 are listed below. These options may be specified as control arguments (-ansi66 and -ansi77) or as keywords in %options and %global statements. This list contains only those items that are incompatible; all other aspects of the language are identical under these two options. This list is not yet complete; additional incompatibilities will appear as Multics FORTRAN approaches FORTRAN 77. Presently, ansi66 is the default.

	ansi66	ansi77
Default Character Length.....	8	1
Can mix character and non-character in common block.....	yes	no
Can mix character and non-character in equivalence group.....	yes	no
Character and array elements word aligned.....	yes	no
Substring references.....	no	yes
Concatenation operator.....	no	yes
(*) length character strings.....	no	yes
Descriptors automatically generated for character arguments.....	no	yes
Embedded blanks in numeric input treated as zeros.....	yes	no
Format control item used for shipping line numbers on input records.....	s	^1
Null values in list-directed input effect corresponding item in input statement.....	yes	no
Blank lines treated as initial lines..	yes	no
Zero trip do loop allowed.....	no	yes

MULTICS FORTRAN MANUAL
ADDENDUM B

SUBJECT

Changes to the Manual

SPECIAL INSTRUCTIONS

This is the second addendum to AT58, Revision 3, dated December 1981. Refer to the Preface for "Significant Changes."

Insert the attached pages into the manual according to the collating instructions on the back of this cover. Throughout the manual, change bars in the margins indicate technical additions; asterisks denote deletions.

Note:

Insert this cover after the manual cover to indicate the updating of the document with Addendum B.

SOFTWARE SUPPORTED

Multics Software Release 10.2

ORDER NUMBER

AT58-03B

December 1983

39097
11183
Printed in U.S.A.

Honeywell

COLLATING INSTRUCTIONS

To update the manual, remove old pages and insert new pages as follows:

Remove	Insert
iii through viii, ix, blank	iii through viii
1-1, 1-2	1-1, 1-2
1-5 through 1-8, 1-9, blank	1-5 through 1-10
3-3, 3-4	3-3, 3-4, 3-4.1, blank
4-11, 4-12	4-11, 4-12 4-12.1, blank
5-1, 5-2	5-1, 5-2 5-2.1, blank
5-19 through 5-22	5-19 through 5-22
5-25 through 5-28	5-25 through 5-28
5-31, 5-32	5-31, 5-32 5-32.1, blank
5-35 through 5-38, 5-39, blank	5-35 through 5-38, 5-39, blank
6-5 through 6-8	6-5, 6-6, 6-7, blank, 6-7.1, 6-8
8-3, 8-4	8-3, 8-4
10-3, 10-4	10-3, 10-4
A-1 through A-4	A-1 through A-4
i-1 through i-8	i-1 through i-6

File No.: 1L23, 1U13

12/83

AT58-03B

C

D

E

F

G

CUT ALONG LINE

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

TITLE: MULTICS FORTRAN MANUAL
ADDENDUM B

ORDER NO. AT58-03B

DATED DECEMBER 1983

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be investigated by appropriate technical personnel and action will be taken as required. Receipt of all forms will be acknowledged; however, if you require a detailed reply, check here. ☐

FROM: NAME _____ **DATE** _____

TITLE _____

COMPANY _____

ADDRESS _____

_____

PLEASE FOLD AND TAPE—
NOTE: U. S. Postal Service will not deliver stapled forms

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 39531 WALTHAM, MA02154

POSTAGE WILL BE PAID BY ADDRESSEE

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTN: PUBLICATIONS, MS486

Honeywell

CUT ALONG

FOLD ALONG LINE

FOLD ALONG LINE

Honeywell

Honeywell Information Systems
In the U.S.A.: 200 Smith Street, MS 486, Waltham, Massachusetts 02154
In Canada: 155 Gordon Baker Road, Willowdale, Ontario M2H 3N7
In the U.K.: Great West Road, Brentford, Middlesex TW8 9DH
In Australia: 124 Walker Street, North Sydney, N.S.W. 2060
In Mexico: Avenida Nuevo Leon 250, Mexico 11, D.F.

33632, 7.5C182, Printed in U.S.A.

AT58-03