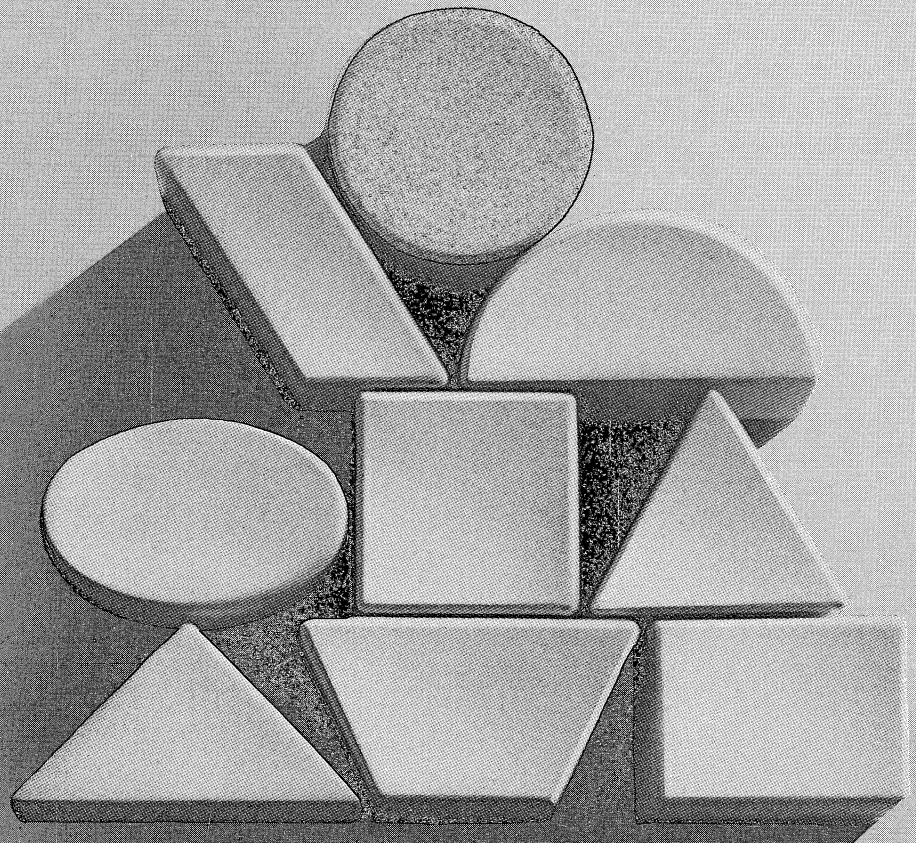
MP/Fortran IV Programmer's Reference



Data General

MP/Fortran IV Programmer's Reference

DataGeneral
Data General Corporation, Westboro, Massachusetts 01581

Notice

Ordering No. 093-400004

Rev. 01, April 1981

Contents

Preface

In writing this manual, we have asssumed that you are a knowledgeable high-level language programmer who has some prior experience with Fortran. This manual was created as a reference that would enable you to start almost immediately to write programs in MP/Fortran IV.

If you are a beginning MP/Fortran IV programmer, we recommend that you read one of the standard Fortran tutorial texts before you start to use this manual.

Before you begin programming in MP/Fortran IV, you should become familiar with the contents of the following MP/OS manuals: *MP/OS Assembly Language Programmer's Reference* (DGC No. 093-40001) and *Learning to use the MP/OS Operating System.* (DGC 093-40000)

Chapter 1
Introduction

This manual is a quick reference source for MP/Fortran IV. It is arranged as follows:

Chapter 2 - Character Set and Program Conventions
Describes the character set of MP/Fortran IV, the rules for naming a variable or a program unit, and the conventions for writing lines of program text.

Chapter 3 - Data
Describes the various different types of data accepted by MP/Fortran IV.

Chapter 4 - Expressions
Describes the MP/Fortran IV arithmetic and logical operators, and describes the rules applied by the compiler when it evaluates expressions containing variables of different data types.

Chapter 5 - Format Specifications
Covers the numerous ways that you may specify the representation of your data for input and output.

Chapter 6 - Classification of MP/Fortran IV Instructions
Classifies all of the MP/Fortran IV language elements by the kinds of operations they perform. Each statement, function, or routine appears in a table with others of its kind, and a syntax description and a brief description of its effect are included for each one.

Chapter 7 - Dictionary
Covers every command, built-in function, and routine in MP/Fortran IV. Entries are arranged in strict alphabetical order. Each entry includes the name of the language element, a brief definition, a syntax diagram, and an example of how the instruction might appear in a program. There are also notes on how the element works, possible errors, and programming hints in some cases.

Chapters 6 and 7 are designed to complement each other. If you need the name of a statement that performs a certain type of operation, it will be easiest to check the tables in Chapter 6. If you want more detailed technical information on a certain language element, you should look in Chapter 7.

Chapter 8 - Compiling, Binding, and Executing MP/Fortran IV Programs
Describes how you produce an executable MP/Fortran IV program under the MP/OS system. This chapter also covers how to run MP/Fortran IV programs under the AOS system (Data General Corporation's Advanced Operating System for the Eclipse computer line) so that your programs can be made fully portable from the start.

Appendices
There are five appendices. Appendix A lists compiler error messages; and Appendix B covers the system error messages that may be returned by subroutines that have an *error parameter*. Appendix C covers the variations among different versions of Fortran IV, and Appendix D describes some aspects of MP/Fortran IV memory allocation of variables and arrays. Appendix E covers information you need in order to write assembly language subroutines for your MP/Fortran IV programs, and develop stand alone programs.

Typesetting Conventions

We have employed the following typesetting conventions throughout this manual:

WHERE	MEANS
COMMAND	You must code the statement or routine call as shown.
[optional(1)]	You have the option of coding this argument. Do not type in the brackets; they just set off the choice.
[...optional(n)]	You can repeat the argument zero or more times.
< argument | argument >	Indicates that you must choose one of the arguments and substitute it in the expression. Do not use the angle brackets or the bar: they just set off the choice.
□	Denotes exactly one character space.

Table 1.1 TYPESETTING CONVENTIONS

Chapter 2
MP/Fortran IV Character Set and Program Conventions

MP/Fortran IV Character Set

Table 2.1 lists the ASCII characters that make up the MP/Fortran IV character set.

Symbol	Name of Symbol
Letters:	A through Z
Digits:	0 through 9
□	Blank
=	Equal Sign
+	Plus
−	Minus
*	Asterisk
/	Slash
(	Left parenthesis
)	Right parenthesis
,	Comma
.	Decimal point
$	Currency symbol
:	Colon
'	Apostrophe
"	Quotation mark
!	Exclamation point

Table 2.1 MP/Fortran IV Character Set

Blanks are significant delimiters in MP/Fortran IV, except as noted in regard to names and to the GO TO statement. (See the Dictionary.)

All ASCII characters are allowed in Hollerith constants with the exception of those that have special meaning for the operating system or for the Macroassembler. Table 2.2 lists the characters that are not permitted.

ASCII Code	Character
012	New Line
014	Form Feed
015	Carriage Return
034	CTRL-\
074	<

Table 2.2 Characters not permitted in Hollerith constants

Names

The symbolic name of a variable must consist of from 1 to 31 alphanumeric characters, beginning with a letter. Subprogram names must be unique within the first five characters. Names of variables, functions, and subprograms must also be distinguishable from statement names, library function names, and operator names (.AND., .LE., etc.).

Embedded blanks in names are not significant, unless they cause a portion of the name to be recognized as a reserved name. For example:

DOZ EN legal name

DO ZEN illegal name

Program Units

An MP/Fortran IV program is made up of one or more program units. A program unit consists of a sequence of statements, with optional comment lines. You compile and assemble each program unit separately, then bind them together at a later time.

MP/Fortran IV program units are implemented as *re-entrant procedures*. All variables and arrays not declared as being in **COMMON** storage are placed on a run-time stack. Repeated entry prior to taking a **RETURN** can be made to any procedure. This, however, is not true if the **COMPILER NOSTACK** option is used.

A program unit may be any of the following:

- Main program
- Subroutine subprogram
- Function subprogram
- Block data subprogram
- Task subprogram

Main Program

A MP/Fortran IV program can have only one main program as a program unit. A main program unit does not contain a **FUNCTION, TASK, SUBROUTINE,** or **BLOCK DATA** statement.

Subroutine, Function and Block Data Subprograms

These subprograms contain either a **FUNCTION, SUBROUTINE** or **BLOCK DATA** statement. **FUNCTION** and **SUBROUTINE** subprograms define executable procedures which you may then reference throughout your program, as needed. A **BLOCK DATA** subprogram allows you to initialize data in labelled **COMMON** storage. Chapter 7 describes all of these subprograms in more detail.

Task Subprogram

A *task* subprogram is a logically complete unit of program execution that requires the use of system resources, such as memory and CPU control. A task subprogram is defined by a **TASK** statement and is referenced in an **ITASK** or **FTASK** subroutine call. For more information on tasking under MP/OS, see Part 3 of the *MP/OS System Assembly Language Programmer's Reference* (DGC No.093-40001).Table 6.10 in Chapter 6 lists the task statements and subroutine calls that are discussed in Chapter 7.

Lines of Program Text

The source text of a program unit consists of those ASCII characters that make up the MP/Fortran IV character set. The text is divided into lines. The compiler flags blank lines as errors.

Comment Line (C)

A line of text that has a **C** in character position 1 is a comment line. The comment may be written anywhere in the line following the **C.**

Optionally Compiled Line (X)

A line of text that has an **X** in character position 1 will be compiled only if you specify the **/X** switch when you compile your program. Otherwise these lines will be ignored by the compiler. See Chapter 8 for more information on the **/X** switch.

Assembly Source Code Line (A)

Lines of assembly source code may be included in an MP/Fortran IV source program. Each line must have an **A** in character position 1. The compiler will delete the **A** when the line is encountered and pass the line intact to the assembler.

Label

If a line does not have a **C, X,** or **A** in character position 1, character positions 1 through 5 are reserved for a label. If the line contains an **X** in column 1, character positions 2 through 5 are reserved for a label.

A label can be any unsigned integer of 1 to 5 digits and can be placed anywhere in character positions 1 through 5.

Leading zeroes are significant in labels; 12 and 0012 will be treated as two different labels.

Comment Following Semicolon

The syntactical scan of a line may be terminated by a semicolon. A semicolon in column 7 or any character position thereafter reserves the remainder of the line for an optional comment. (A semicolon appearing within a Hollerith constant, or a text string for a STOP or PAUSE statement are not recognized for this purpose.)

MP/Fortran IV Statements

The basic semantic unit of an MP/Fortran IV program unit is called a statement. A line of text may contain an MP/Fortran IV statement or part of one.

A statement must start at character position 7 or beyond.

Continuation Lines

When an MP/Fortran IV statement requires more than one line of text, continuation lines must be indicated by putting a character other than a 0 or a blank in character position 6 of the second and subsequent lines. All other MP/Fortran IV statement lines must have a 0 or a blank in character position 6. You must never label a continuation line.

Partial Ordering of Statements

For compiler efficiency, Data General's MP/Fortran IV requires a partial ordering of statements. The order of statements is:

1. **COMPILER DOUBLE PRECISION** or **COMPILER NOSTACK** statement.
2. **OVERLAY** or **CHANTASK** statement.
3. **PARAMETER** statements.
4. **FUNCTION, SUBROUTINE,** and **TASK** statements.
5. Declaration statements. These begin with the keywords: **COMMON, COMPLEX, DIMENSION, DOUBLE PRECISION, EQUIVALENCE, EXTERNAL, INTEGER, LOGICAL,** or **REAL.**

*6. Statement functions.

*7. Executable statements.

*__FORMAT__ *statements and* __DATA__ *initialization statements are permissible.*

Chapter 3
Data

Constants, Variables and Parameters

A *constant* is a known value that does not alter during program execution.

A *variable* is represented by a symbolic name and is a quantity that may be altered during execution.

A *parameter* is represented by a symbolic name and can be used anywhere a constant of the same type can be used. (Statement numbers, octal strings in **PAUSE** and **STOP** statements, and numbers in **FORMAT** statements are *not* constants). See the description of **PARAMETER** statements in Chapter 7 for details.

Constants and variables have data types associated with them. Mathematical data may be of types **INTEGER, REAL, DOUBLE PRECISION, COMPLEX,** or **DOUBLE PRECISION COMPLEX.**

INTEGER type data is represented internally in fixed-point notation. All other mathematical data types are represented internally in floating-point notation.

Constants and variables may be associated with a **LOGICAL** data type. In addition, *string constants* are permitted in the source code. String constants cannot be associated with parameters.

Integer Data

An integer constant is a signed or unsigned whole number written without a decimal point.

An integer variable is usually implicitly typed; i.e., if the first character of the symbolic name is I, J, K, L, M, or N, the symbolic name represents an integer variable unless otherwise specified. Examples of integer constants and variables are:

Constants	Variables
-125	ITEM
0	JOBNO
+4525	LUCKY
377K	MASKBYTE

As shown, integer constants can be specified in octal format by writing the number followed by the letter K. Some additional examples are:

Octal Constant	Decimal Value
10K	8
777K	511
-1K	-1

An integer datum is stored in one word (16 bits). The range of integer values is -32,767 to 32,767 inclusive.

Real Data

A real constant is signed or unsigned and consists of one of the following:

1. One or more decimal digits written with a decimal point.
2. One or more decimal digits written with or without a decimal point, followed by a decimal

exponent written as the letter **E** followed by a signed or unsigned integer constant. When the decimal point is omitted, it is always assumed to be immediately to the right of the rightmost digit. The exponent value may be explicitly 0; the exponent field may not be blank.

A real variable is usually implicitly typed. If the first character of the symbolic name is not I, J, K, L, M, or N the symbolic name represents a real variable unless otherwise specified. Some examples of real constants and variables follow.

Constants	Constant Value	Variables
0.0	0.0	ALPHA
.000056789	.000056789	B25
+15.E-04	+.0015	EXIT
-0005E2	-500	C

A real datum is stored in two 16-bit words. The range of real values is $2.4 * 10^{-78}$ to $7.2 * 10^{75}$.

Double Precision Data

A double precision constant is signed or unsigned and consists of a sequence of decimal digits written with or without a decimal point, followed by a decimal exponent written as the letter **D** followed by a signed or unsigned integer constant.

When the decimal point of a double precision constant is omitted, it is always assumed to be immediately to the right of the rightmost digit. The exponent value may be explicitly 0; the exponent field may not be blank.

A double precision variable must be explicitly specified as such in a **DOUBLE PRECISION** type statement.*

Constants	Constant Value
-21987654321D0	-21987654321
5.0D-3	.005
.203D0	.203

If the first statement of the MP/Fortran IV program is* COMPILER DOUBLE PRECISION, *each real variable or constant will be forced to type* **DOUBLE PRECISION.

Variable Type Statement
DOUBLE PRECISION D, E, F2

A double precision datum is stored in four words.

Complex Data

A complex constant is an ordered pair of signed or unsigned real constants, separated by a comma and enclosed in parenthesis.

A complex single-precision variable must be explicitly specified as such in a **COMPLEX** type statement.

Constants	Constant Value
(3.2, 1.86)	3.2+1.86i
(2.1, 0.0)	2.1+0.0i
(5.0E3, -2.12)	5000.-2.12i

Variable Type Statement
COMPLEX C1, C2

A complex single-precision datum is stored in four words.

Double Precision Complex Data

A double precision complex constant is an ordered pair of signed or unsigned double precision constants separated by a comma and enclosed in parenthesis.

A double precision complex variable must be explicitly specified as such in a **DOUBLE PRECISION COMPLEX** type statement.*

Constant
(-3456.0012D-5, .0034567D+3)

Variable Type Statement
DOUBLE PRECISION COMPLEX DC1, DC2

If the first statement of the MP/Fortran IV program is* COMPILER DOUBLE PRECISION, *each complex variable or constant will be forced to type* **DOUBLE PRECISION COMPLEX.

A double precision complex datum is stored in eight words.

Logical Data

A logical constant is a truth value written as **.TRUE** or **.FALSE.**

A logical variable must be explicitly specified as such in a **LOGICAL** type statement. For example:

```
LOGICAL BOOL1, BOOL2
```

A logical datum is stored in one word. The value **.TRUE.** is stored as 177777_8, and **.FALSE.** as 000000. When testing for logical, any non-zero word will be treated as the value **.TRUE.**. (177777_8 is also the integer value -1.)

String (Hollerith) Constants

String constants are strings of characters of the MP/Fortran IV character set, including blanks. A string may be enclosed in quotation marks, it may be enclosed in apostrophes, or it may be represented by a Hollerith constant. A Hollerith constant is an integer constant, followed by the letter **H** followed by the string. The integer constant indicates the number of characters in the string. For example,

```
"END"
'END'
3HEND
```

are equivalent strings.

Quotation marks may appear in strings that are surrounded by apostrophes; apostrophes may appear in strings that are surrounded by quotation marks. Both apostophes and quotation marks may be used as characters within Hollerith constants.

String constants may appear in:

- The I/O list of a **TYPE** or **ACCEPT** statement. The constant is written out precisely as it appears in the statement.
- A **FORMAT** statement. On output, the string is written to the output device. On input, the string is overwritten by an equal number of characters from the input record.
- A **DATA** initialization statement. The data type of the corresponding variable (or variables) in the variable list is ignored.
- The argument list of a **CALL** statement or the argument list of a function reference.

String constants may also appear in expressions. In these cases, however, only the first two characters of the string constant are significant, and they are treated as a one-word integer operand rather than a string.

Within string constants, octal codes for ASCII control characters delimited by angle brackets may appear. The codes will be passed to the assembler for interpretation. For example, a carriage return can be passed in a string as follows:

```
"DATA FOLLOWS: <15> "
```

Note that when using formatted I/O (**WRITE/FORMAT** statements), carriage control information should *not* be passed in string constants but should follow ANSI Fortran standard conventions.

String constants are stored one character per byte (two characters per word). Normally, if the character count of a string is even, a word of all zeroes is generated to indicate the end of a string. This does not occur, however, in the case of **DATA** initialization using a string constant.

Arrays and Subscripts

An array is an ordered set of data of one or more dimensions. Up to 128 dimensions are permitted. A single symbolic name identifies the array. Each element of the array is identified by a qualifier of the array name, called a subscript.

An array is specified by the appearance of its symbolic name in a **DIMENSION, COMMON** or data type statement with parenthesized dimensioning information. Some examples are:

DIMENSION A(10,10)	A is a two-dimensional real array of 100 elements
COMMON B(2,5,5)	B is a 3-dimensional real array of 50 elements stored in common.
INTEGER C(5,2,2,2)	C is a 4-dimensional integer array of 40 elements.

Variables may be used in the specification of array subscripts in **DIMENSION** and data type statements. This is called adjustable dimensioning.

Variables can only be specified if the array name and the variable subscript bounds are dummy arguments of a subprogram in which they specify the array. Then, actual arguments giving values to variable subscript bounds can be passed when the subprogram is called. For example, in

```
SUBROUTINE SUB25 (MAT,I,J)
INTEGER I,J
REAL MAT (I, 0:J)
```

values for *I* and *J* would be passed in a call to SUB25.

Note that array MAT is subscripted as

```
MAT(I,0:J)
```

This specifies that the lower bound of the second subscript is 0 and the upper bound is *J*. If only the upper bound of an array dimension is specified, the lower bound is assumed to be 1. For example,

```
DIMENSION B(5)
```

assumes that the first array element is B(1) and the last is B(5).

Both upper and lower bounds are specified in the following example:

```
DIMENSION F(0:11), G(-2:4, -3:0)
```

The array *F* is a one-dimensional array of 12 elements, the first of which is F(0). The second array is a two-dimensional array of 7x4 or 28 elements; the first element is G(-2, -3).

The subscript of an array element is written as a parenthesized list of subscript expressions. Each subscript expression can have one of the following forms:

- In the non-executable statements **EQUIVALENCE** and **DATA**, each element of the subscript list must be an integer constant (or a parameter representing an integer constant.)
- In expressions within executable statements, each element of the subscript list must be an expression whose value is of type integer.

Some examples are:

```
A(I, J)
```
could be an element of array A(10, 10).

```
B(1,1,1)
```
is the first element of array B(2,5,5,).

```
C(ITEM-1)
```
could be an element of array C(0:6).

```
D(I+IFIX(SQRT(R -B/3.0)), J/K)
```
could be an element of D(3,8).

The subscript of an array element cannot assume a value during execution that is less than the lower bound for that dimension of the array or larger than the upper bound for that dimension of the array, except if a single subscript is given for a multidimensional array. (A single subscript can be used to index a multidimensional array. If *A* is dimensioned (0:4, 0:4), the 25 elements of *A* can be referenced as A(1) through A(25). This is the same as the single subscript reference allowed in **DATA** and **EQUIVALENCE** statements.

Values are assigned to array elements so that the first subscript expression varies most rapidly, then the second subscript expression, etc.

For example, elements of array C(15) are stored:
C(1), C(2),..., C(15)

Elements of array A(10,10) are stored:
A(1,1), A(2,1),..., A(9,1),
A(10,1), A(1,2),...,A(9,10), A(10,10)

Elements of array B(2,3,4) are stored:
B(1,1,1), B(2,1,1), B(1,2,1), B(2,2,1),
B(1,3,1), B(2,3,1), B(1,1,2),...,
B(1,3,4), B(2,3,4)

Chapter 4
Expressions

Definition of an Expression

An expression is a combination of data elements (variables, array elements, functions, and constants) with operators. The MP/Fortran IV operators are arithmetic, relational, and logical.

Arithmetic Expressions

An arithmetic expression is formed with arithmetic operators and arithmetic data elements. Table 4.1 lists the operators.

Operator	Operation
+	Addition (or unary plus)
−	Subtraction (or unary minus)
*	Multiplication
/	Division
**	Exponentation

Table 4.1 Arithmetic Operators

An arithmetic datum has one of five possible data types:

Type	Rank of Data
Double precision complex	1 (Highest)
Complex	2
Double precision	3
Real	4
Integer	5 (Lowest)

Table 4.2 Arithmetic Data Types

Evaluating Arithmetic Expressions

The following rules apply to evaluating arithmetic expressions:

- When either plus (+) or minus (−) is used as a unary operator, the data type of the result is the same as that of the operand.
- When two operands of the same data type are used in an expression containing one of the operators + − * / the data type of the result is the same as that of the operand.
- Mixed data type operands are permitted. In expressions formed with the operators + − * / the lower-ranking operand is converted temporarily to the higher-ranking type, and the result of evaluation will have the higher-ranking data type. When a **COMPLEX** or **DOUBLE PRECISION COMPLEX** operand is combined with an operand that is not complex, the temporary has an imaginary part equal to zero.
- When an expression consists of a **DOUBLE PRECISION** operand and a **COMPLEX** operand, the **DOUBLE PRECISION** operand is converted to a single precision **COMPLEX** temporary, and the result of evaluation is **COMPLEX.**
- Arguments of library functions are not converted to temporaries of the appropriate

type.

- Mixed data types are permitted in expressions consisting of base and exponent operands (** operator), except that it is illegal to raise an **INTEGER** base to a **COMPLEX** or **DOUBLE PRECISION COMPLEX** exponent.
- When base and exponent operands are of the same type, the result is also of that data type.
- When the base and exponent are of differing data types, the resultant value will be of the higher data type, except in the case of raising a **COMPLEX** base to a **DOUBLE PRECISION** exponent. The result of this operation is **DOUBLE PRECISION COMPLEX.**

Hollerith (string) constants may appear wherever integers are permitted in expressions. Only the first two characters of the string constant are significant, representing the ASCII value of the characters.

Logical variables and constants are treated as integers in the evaluation of arithmetic expressions.

Order of Evaluation

The following rules govern the order in which operations are evaluated within an arithmetic expression.

The arithmetic operators have precedence as shown in Table 4.3.

Operator	Precedence
**	Highest (evaluated first)
/ *	Next highest
+ −	Lowest (evaluated last)

Table 4.3 Precedence of Arithmetic Operators

- When two operators are of equal precedence, operations are evaluated from left to right in the expression.
- Parentheses are used to alter the order of operator precedence. A parenthesized expression is evaluated as an entity before further evaluation proceeds. When parenthesized expressions are nested, the innermost is evaluated first.

The rules for evaluating mixed data types are shown in tabular form in Tables 4.4 and 4.5. Follow first operand vertically and the second operand horizontally. The resulting data type appears in the box where they intersect.

OPERAND B	OPERAND A				
	Integer	**Real**	**Double Precision**	**Complex**	**Double Complex**
Integer	Integer	Real	Double Precision	Complex	Double Complex
Real	Real	Real	Double Precision	Complex	Double Complex
Double Precision	Double Precision	Double Precision	Double Precision	Complex	Double Complex
Complex	Complex	Complex	Double Precision Complex	Complex	Double Complex
Double Complex	Double Complex	Double Complex	Double Complex	Double Complex	Double Complex

Table 4.4 Evaluating mixed data types in addition, subtraction, multiplication, and division (Operators are + − * /)

BASE OPERAND	EXPONENT OPERAND				
	Integer	**Real**	**Double Precision**	**Complex**	**Double Complex**
Integer	Integer	Real	Double Precision	ILLEGAL	ILLEGAL
Real	Real	Real	Double Precision	Complex	Double Complex
Double Precision	Double Precision	Double Precision	Double Precision	Complex	Double Complex
Complex	Complex	Complex	Double Complex	Complex	Double Complex
Double Complex	Double Complex	Double Complex	Double Complex	Double Complex	Double Complex

Table 4.5 Evaluating mixed data types for exponentiation (Operator is **)

Sample Arithmetic Expressions

Some examples of legal arithmetic expressions are shown below. Table 4.6 shows the data type of each data element.

```
C*DC**C
B/(I + J)
D - B**A
D*I**J
```

Data element	Data type
I,J	Integer
A,B	Real
D	Double Precision
C	Complex
DC,CD	Double Precision Complex

Table 4.6 Types of data elements in arithmetic expressions

Relational and Logical Expressions

Relational Expressions

A relational expression consists of two arithmetic expressions separated by a relational operator. Table 4.7 lists the relational operators.

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

Table 4.7 Relational operators

Logical Expressions

A logical expression is formed with logical operators and logical or integer elements. Logical elements are those that have been given the data type **LOGICAL.** Table 4.8 lists the logical operators.

Operator	Representing
.OR.	Logical disjunction. Result is 1 if either operand has a 1 in that bit position.
.AND.	Logical conjunction. Result is 1 if, and only if, both operands have a 1 in that bit position.
.NOT.	Logical negation. Result is the bit complement of the operand.

Table 4.8 Logical operators

Evaluation of Logical and Relational Expressions

Logical and relational operations may be combined within expressions.

Logical and relational expressions can be treated as full-word operations evaluated bit by bit, as in masking, or can be considered as evaluating to truth values: **.TRUE.** or **.FALSE.**,

where: **.TRUE.** is 177777_8 (-1)

.FALSE. is 000000_8 (0)

These octal values for **.TRUE.** and **.FALSE.** are generated for literals and as the result of evaluations to a truth value. When testing for a truth value, any non-zero word is considered **.TRUE.** and a word of all zeroes is considered **.FALSE.**

The general rules of precedence in evaluating relational and logical expressions are the same as for arithmetic expressions: parenthesized expressions are evaluated first; then expressions containing operators of equal precedence are evaluated from left to right.

Arithmetic expressions are evaluated first, in accordance with the rules of arithmetic operator precedence, then relational operations, and then logical operations. Table 4.9 shows the precedence of all operators in an MP/Fortran IV expression.

Operator	Precedence
**	1 (highest)
*/	2
+ −	3
.GE., .GT., .EQ., .NE., .LT., .LE.	4
.NOT.	5
.AND.	6
.OR.	7 (lowest)

Table 4.9 Precedence of all operators in MP/Fortran IV

Tables 4.10 and 4.12 show examples of evaluation of logical and relational expressions, both for truth value results and on a full-word, bit-by-bit basis.

If Y Operand is	and Z Operand is	Then .NOT. Y is	and Y .AND. Z is	and Y .OR. Z is
.FALSE.	.FALSE.	.TRUE.	.FALSE.	.FALSE.
.FALSE.	.TRUE.	.TRUE.	.FALSE.	.TRUE.
.TRUE.	.FALSE.	.FALSE.	.FALSE.	.TRUE.
.TRUE.	.TRUE.	.FALSE.	.TRUE.	.TRUE.

Table 4.10 Truth Operations (.FALSE. = 000000_8; .TRUE. = 177777_8)

Expression	Value	Interpretation
W .LE. X	.TRUE.	True
W .LT. X .AND. W .LT. Y	.TRUE.	True and true
W .NE. X .AND. .NOT. A	.FALSE.	True and false
.NOT. A .OR. W .EQ. X	.FALSE.	False or false

Table 4.11 Logical and Relational Truth Evaluation
(Assume that: A = .TRUE., W = 2, X = 4, Y = 6)

If Operands are	Then .NOT. Y is	and .NOT. Z is	and Y .AND. Z is	and Y .OR. Z is
Y = 101010 Z = 110100	010101	001011	100000	111110

Table 4.12 Full Word Operations

Expression	Value	Interpretation
J .AND. K	117	Mask K with J.
.NOT. J .AND. K	47000	Mask K with the complement of J.
5*(-(K .EQ. L))	0	Since K does not equal L, the parenthesized expression evaluates to .FALSE. (0).
5*(-(K .EQ. ''NO''))	5	47117 is ASCII for NO. The result of the parenthesized expression is .TRUE. (-1).

Table 4.13 Logical and Relational Full Word Evaluations
(Assume that: J = 377_8, K = 47117_8, L = 200_8)

Assignment Statements

The syntax of an assignment statement is:

variable = *expression*

where: *variable* is a subscripted or non-subscripted variable name, and *expression* is any legal MP/Fortran IV expression.

The expression on the right-hand side of the equals sign is evaluated according to the rules given in the previous chapter, and the resulting value is converted to the type of the variable on the left-hand side of the assignment statement, except in the case of a complex expression.

Type of Variable	Type of Expression				
	INTEGER	**REAL**	**DOUBLE PRECISION**	**COMPLEX**	**DOUBLE PRECISION COMPLEX**
INTEGER	Evaluate and assign	Fix* and assign	Fix* and assign	Illegal	Illegal
REAL	Float and assign	Evaluate and assign	Truncate mantissa and assign	Illegal	Illegal
DOUBLE PRECISION	Float to 56 bits and assign	Extend mantissa and assign	Evaluate and assign	Illegal	Illegal
COMPLEX	Float, set imaginary part to zero, and assign	Set imaginary part to zero, and assign	Truncate mantissa, set imaginary part to zero, and assign	Evaluate and assign	Truncate and assign
DOUBLE PRECISION COMPLEX	Float, set imaginary part to zero, and assign	Extend mantissa, set imaginary part to zero, and assign	Set imaginary part to zero, and assign	Extend mantissa by 32 bits of zerores	Evaluate and assign

Table 4.14 Assignment statement conversion rules

**If the real number is within the legal integer range, it is "fixed": the fractional part is truncated, and the whole number portion is assigned.*

Chapter 5
Formatted I/O

In MP/Fortran IV, the specification of format serves two basic purposes:

1. For input (Read), formatting designates how the computer will interpret and store the input data.
2. For output (Write), formatting allows precise control of the data as it will appear on the printed page.

Formatting specifications allow the programmer to control the field width allotted to any datum, the spacing between fields, the assignment of data to particular records or lines, and the notation in which the data will be represented externally.

The specification of format can be given in a **FORMAT** statement, or it can be contained in an array that is read at run-time.

FORMAT Statement Syntax

The generalized form of the **FORMAT** statement is:
n **FORMAT** *(f_1 [s_1] f_2 [s_2]...f_m [s_m])*

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

You must label all **FORMAT** statements.

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

Field Descriptors

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

where: *r* is a repeat count indicating the number of times you want MP/Fortran IV to interpret this descriptor.

q is a format code.

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

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

For further information on the repeat count, see "Group Repeat Specifications", page 5-16.

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

The field separators are slashes and commas. A slash indicates the end of a record and a comma indicates the end of a datum within the record. For more information, see "Field Separators" on page 5-17.

Following is a list of field descriptors:

Numeric

I*w*	Integer Data
F*w.d*	Real Data
E*w.d*	Explicit Exponent Real Data
D*w.d*	Double Precision Real Data
G*w.d*	Generalized Real Data (may substitute for F, E, or D)

O*w*	Octal Data (Base 8)
O$_8$*w[.d]*	Octal Modifier
*n***P**	Scale Factor

Logical

L*w*	Logical Data

String and Editing

A*w*	Alphanumeric Data
S*w*	String Data
*n***H**	Hollerith Data
"..."	String Data
'...'	String Data
*n***X**	Column Positioning Control
T*n*	Tabulation Control

Carriage Control Characters

' '	Print after New-line
'0'	Print after two New-lines
'1'	Print after Form Feed
Z	Suppress a New-line
/	Field separator

Records and Data Association

MP/Fortran IV handles formatted I/O on a per-record basis, a record being a string of characters terminated by a data sensitive delimiter. (Data sensitive delimiters are New-line, Carriage Return, Form Feed, and Null.) The maximum number of characters in a record is 136. MP/Fortran IV reads or writes one record per I/O statement unless field separators (/) are included in the **FORMAT** statement.

MP/Fortran IV scans the format specification as well as the corresponding input or output list from left to right. It associates the field descriptors with the entities of the input or output list in the order in which you write them. For further information on the interaction between these statements, see "Format and I/O Statement Association" on page 5-17.

Numeric Field Descriptors

I Format Code

Transmits numeric data in integer format.

Format

*[r]*I*w*

Where: *r* is a repeat count.

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

Input

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

An input field containing all blanks has a value of zero. MP/Fortran IV ignores leading blanks; embedded and trailing blanks have a value of zero.

If you read in a real value in integer format, MP/Fortran IV signals an error and truncates from the first "bad" character (probably the decimal point) onward. The characters before the decimal point form the integer value.

Output

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

Examples

Input Characters	Format	Internal Value
50	I2	+50
50□	I3	+500
−5□□3	I5	−5003
□□□□	I4	0
□1234	I3	+12
+981□□	I6	Run-time error. (Reads in +981, then tries to pad the field with zeroes until overflow occurs.)

Input

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

Output

F Format Code

Transmits numeric data in real format.

Format

*[s][r]***F***w.d*

Where: *s* is a scale factor.

r is a repeat count.

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

. is a decimal point.

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

Two **F** descriptors cause the datum to be interpreted as **COMPLEX**. (See "Complex Data in I/O Lists" later in this chapter.)

Input

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

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

If an input datum is unsigned, MP/Fortran IV interprets the datum as a positive number. An input field containing all blanks has a value of zero. MP/Fortran IV ignores leading blanks; embedded and trailing blanks have a value of zero.

Output

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

The field width (*w*) must be large enough to accommodate a minus sign, if any (MP/Fortran IV suppresses plus signs on output), and a decimal point. If *w* is not large enough, MP/Fortran IV prints an asterisk(*), followed by as many digits as will fit in the field.

Examples

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

Input

Internal Value	Format	Output Characters
−98.7	F7.2	□−98.70
+100.67	F8.1	□□□ 100.7
+123.54	F5.2	*123.
+42.35	F9.3	□□□ 42.350

Output

E Format Code

Transmits numeric data in exponential format.

Format

*[s][r]***E***w.d*

Where: *s* is a scale factor.

r is a repeat count.

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

. is the decimal point.

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

Two **E** descriptors cause the datum to be interpreted as **COMPLEX**. (See "Complex Data in I/O Lists" later in this chapter.)

Input

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

- A signed integer constant, or
- An **E** followed by a signed or unsigned integer constant.

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

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

If an input datum is unsigned, MP/Fortran IV interprets the datum as a positive number. An input field containing all blanks has a value of zero. MP/Fortran IV ignores leading blanks; embedded and trailing blanks have a value of zero.

Output

MP/Fortran IV right justifies the output characters within the external field, outputs leading blanks if the datum's length is less than *w*, and signs the datum if it is negative. The decimal point is followed by *d* digits. The letter **E** and a decimal exponent follow the number. If the number of fraction digits is greater than *d*, MP/Fortran IV rounds the fraction at the *d* position.

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

Examples

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

Input

Internal Value	Format	Output Characters
+12.34	E10.3	0.123E□□2
−19.54	E6.2	****** (format not wide enough to include exponent)
+31.987	E10.3	0.320E□□2

Output

D Format Code

Transmits numeric data in double precision format.

Format

*[s][r]***D***w.d*

Where: *s* is a scale factor.

r is a repeat count.

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

. is a decimal point.

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

Input

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

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

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

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

If an input datum is unsigned, MP/Fortran IV interprets the datum as a positive number. An input field containing all blanks has a value of zero. MP/Fortran IV ignores leading blanks; embedded and trailing blanks have a value of zero.

Output

MP/Fortran IV right justifies the output characters within the external field, outputs leading blanks if the datum's length is less than *w*, and signs the datum if negative. The decimal point is followed by *d* digits. The letter **D** and a decimal exponent follow the number. MP/Fortran IV signs the exponent if it is negative. If the number of fraction digits is greater than *d*, MP/Fortran IV rounds the fraction at the *d* position.

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

Examples

Input Characters	Format	Internal Value
1234.56789012345	D16.0	1234.56789012345
1234567890123456	D16.5	1234567890.12346
1234567890123456	D20.0	Undefined Results (format larger than input field)
−12345□□□□□□□□□□	D16.5	−1234500000.
□□123.45D5	D10.7	12345000.

Input

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

Output

G Format Code

Transmits data using a generalized format code.

Format

[*s*][*r*]G*w*.*d*

Where: *s* is a scale factor.

r is a repeat count.

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

. is a decimal point.

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

You can use this code with real and double precision data, where it is equivalent to using the **F**, **D**, or **E** field descriptors. The equivalent form used depends on the type of the corresponding element in the I/O list.

Input

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

If an input datum is unsigned, MP/Fortran IV interprets the datum as a positive number.

An input field containing all blanks has a value of zero. MP/Fortran IV ignores leading blanks; embedded and trailing blanks have a value of zero.

Output

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

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

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

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

Examples

See the examples for **F** and **E** format codes (input).

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

Output

O Format Code

Transmits numeric data in unsigned octal format.

Format

*[r]*O*w*

Where: *r* is a repeat count.

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

Input

MP/Fortran IV reads an unsigned octal value from the input field.

An input field containing all blanks has a value of zero. MP/Fortran IV ignores leading blanks; embedded and trailing blanks have a value of zero.

If a character other than an octal digit is read, MP/Fortran IV signals an error and truncates from the first "bad" character onward. The characters before the "bad" character form the octal value.

Output

MP/Fortran IV converts the internal value of the datum to unsigned octal integer format, right justifies it according to *w* with leading zeros if needed. If the field width is not wide enough to output the datum, MP/Fortran IV outputs *w* asterisks (*).

Examples

Input Characters	Format	Internal Value
50	O2	+40
10□	O3	+64
177776	O6	-2
	O4	0
□50	O3	+40
17787	O5	+127

Input

Internal Value	Format	Output Characters
+40	O2	50
+40	O3	050
-2	O6	177776
+256	O2	** (format too small)

Output

O Format Modifier

Converts any datum to octal format.

Format

*[r]***O***qw*

Where: *r* is a repeat count.

q is an **I**, **F**, **D**, **E** or **G** field descriptor.

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

Input

Data will be represented in octal, according to the **I**, **F**, **D**, **E** or **G** descriptors, in the same form as they would be in decimal.

Note that exponents will be octal in all cases for the **D**, **E**, and **G** descriptors.

An input field containing all blanks has a value of zero. MP/Fortran IV ignores leading blanks; embedded and trailing blanks have a value of zero.

Output

If the field contains less than *w* digits, MP/Fortran IV right justifies it and blank pads the field on the left. If the digits require more than the width specified by the field descriptor (*w*), MP/Fortran IV outputs the leftmost *w* characters.

Examples

Input Characters	Format	Internal Value
31	OI2	+25
200000	OI6	Overflow error
25□	OI3	+168
1.11	OF4.2	+1.141
32.5	OD4.1	.21125D2

Input

Internal Value (Decimal)	Format	Output Characters
123	OI3	173
10.5	OF4.1	12.4
1.25	OF3.1	1.2
.123	OD10.5	□.11656D□1
130	OD10.5	□.20200D□3

Output

P Scale Factor

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

Format

*n***P**

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

Rules

You may affix a scale factor to field descriptors **F**, **E**, **G**, or **D** in the format specification, such as 2**PE**15.5. Once you specify a scale factor, it remains in effect for all subsequent **D**, **E**, **F**, and **G** field descriptors within the same format specification until MP/Fortran IV encounters another scale factor. (When MP/Fortran IV encounters the rightmost parenthesis, rescanning the format specification does not reset the scale factor to zero.) To reset the scale factor to zero, specify **OP**.

Input

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

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

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

The internal representation of the number is equivalent to the external representation with the decimal point shifted to the left *n* places if the scale factor is positive, and to the right *n* places if the scale factor is negative. (You will see that on output you can simply undo this process by specifying the same scale factor.) If the number entered contains an explicit exponent, MP/Fortran IV ignores the scale factor. For example, if you enter the number 3.**E**02 and apply the field descriptor 2**PE**8.2, the stored result is 300, i.e., the scale factor has no effect.

Output

The conversion formula for scale factoring on output is:

External number = *Integer value* x 10** *n*

You can specify the scale factor for all real numbers on output. With an **F** field descriptor, the effect is the same as for input except that MP/Fortran IV moves the decimal point in the opposite direction: it shifts the decimal point to the right *n* places if

the scale factor is positive, and to the left *n* places if the scale factor is negative.

For example, if you want to output the internally stored value 12.9876 as 1298.76, use the format code 2**PF** 7.4. With an **E** or **D** field descriptor, the scale factor given alters the position of the decimal point, adjusting the exponent to account for this change. The result is that no change occurs in the value of the number.

The binary system of numbers expresses integers precisely but approximates most decimal fractions. In accounting, for example, decimal fractions must be as precise as possible. Use of the scale factor allows you to retain more precise decimal values.

For example, given the entry $8.59, the dollars and cents are of equal importance during processing. To achieve decimal precision, the field descriptor of the number on input might be -2**PF**7.2. The internal representation is then 859., making the number an integer. After performing all calculations, you output the number using the same format, -2**PF**7.2, resulting in a number that reflects precise dollars and cents.

Examples

Input Characters	Format	Internal Value
−25.44 345.71	FORMAT (2PF6.2,F7.2)	−.2544 +3.4571
12.345	3PE8.3	+.012345
12.345	−3PE8.3	+12345.
3.E02	2PE8.2	+300

Input

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

Output

Logical Field Descriptor

L Format Code

Transmits data in logical format.

Format

*[r]***L***w*

Where: *r* is a repeat count.

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

Input

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

If you read in a value other than a capital **T** or **F** in logical format, MP/Fortran IV signals an error and returns **FALSE**.

Output

If the internal value is **true**, MP/Fortran IV outputs *w* − 1 blanks followed by a **T**. If the internal value is **false**, it outputs *w* − 1 blanks followed by an **F**.

Examples

Input Characters	Format	Internal Value
□□ TRUE	L6	True
□ F120	L5	False
FALSE	L10	False
12345	L5	True. Error (nonlogical characters)
FT	L2	False

Input

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

Output

String Field Descriptors

A Format Code

Transmits alphanumeric data in character (ASCII) format.

Format

*[r]***A***w*

Where: *r* is a repeat count.

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

Input

MP/Fortran IV reads *w* characters for storage in an input list element from the input field. If the end of input line is reached before *w* characters are read, the remaining space is filled with blanks. MP/Fortran IV will store no more than two characters in an input list element. If *w* is greater than two, the last two characters input are stored in the list element.

Output

MP/Fortran IV outputs characters to an external field *w* characters wide.

If *w* is greater than two, MP/Fortran IV outputs *w*-2 blanks followed by the first two characters in the input list element.

Examples

(The symbol # represents a null byte.)

Input String	Format	Internal String
ABCD	A1	A□
ABCD	A2	AB
ABCD	A4	CD
A	A2	A#

Input

Internal String	Format	Output String
AB	A2	AB
AB	A1	A
AB	A3	□AB
ABCD	A3	□AB

Output

S Format Code

Transmits alphanumeric data in character (ASCII) format.

Format

*[r]*S*w*

Where: *r* is a repeat count.

w is an unsigned integer constant that specifies the field's width in characters. A maximum field width of 132 characters is allowed. (Using a width greater than 132 characters causes MP/Fortran IV to report a run-time error.

Input

MP/Fortran IV reads *w* characters, stores them starting at the location of the input list element, and appends a null byte. The input list element must be a single variable, not an array name. (Array elements are single variables.)

S format ignores the type of the input list element. Because of this, if a field width that is too wide for the type is specified, other data in memory will be overwritten after the space in the input list element is exhausted. The number of characters that can be stored in each type is defined in the table below:

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

To read a long string into memory, name the first element of an array as the input list element and use the length of the array multiplied by the number of character its type can hold as the field width.

Output

If the length of the string is *n* characters, characters will be written out as follows:

$w = n$ Entire string is written out.

$w > n$ Entire string is written out, followed by $w - n$ spaces.

$w < n$ First w characters are written out.

Examples

(The symbol # represents a null byte.)

Input String	Format	Data Type	Number of elements	Internal String
ABCDE	S5	Integer	3	ABCDE#
ABCDEFG	S4	Integer	3	ABCD#

Input

Internal String	Format	Output
ABCDEFG#	S7	ABCDEFG
ABCDEFG#	S5	ABCDE
ABCDEFG#	S8	ABCDEFG□

Output

Hollerith and String Constants

Write an ASCII character string directly from a **FORMAT** statement.

Formats
"string"
'string'
*n***H***string*

Where: " and ' are valid delimiters containing the specified character string.

n is an unsigned integer constant that specifies the number of characters in the string.

H is the designation for a left-justified Hollerith constant.

Output
No entity in an I/O list corresponds to the string data. If the internal string is in **H** format, MP/Fortran IV outputs *n* characters. If the internal string is delimited by apostrophes or quotation marks, it outputs the enclosed characters.

Example

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

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

□ OUTPUT□ IS□ 3.4□□ 5.6
```

Output

Editing Field Descriptors

X Format Code

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

Format
*n***X**

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

Input
On input of the external field, MP/Fortran IV skips *n* character positions. You must specify *n* even if you want to skip only one character position.

Output
In the external output field, MP/Fortran IV inserts *n* ASCII blank characters. You must specify *n* even if you want to insert only one blank character.

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

T Format Code

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

Format

T*n*

Where: *n* is an unsigned integer constant that specifies a character or print position.

Input

Tabbing occurs to the character position in the record specified by *n*. MP/Fortran IV will read the character(s) beginning at position *n*. If *n* is less than the current position in the input record, the tab field is ignored.

Output

Tabbing occurs to the character position in the record specified by *n*. However, if there is data at the beginning of the record, MP/Fortran IV will use the first character for carriage control and will not print it. If *n* is less than the current position in the output record, the tab is ignored.

Examples

The following compares **X** and **T** format codes on output.

Format Items	Resulting Output
5X, 'A'	□□□□□ A
T3, 2X, 'A'	□□□□□ A
5X, T3, 'A'	□□□□□ A (Note that the tab is ignored.)

Comparison of X and T format codes

Group Repeat Specifications

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

Format

r(field descriptor(s))

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

Rules

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

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

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

Examples

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

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

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

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

Carriage Control

MP/Fortran IV does not print the first character of each output record; it uses this character for vertical carriage control. However, MP/Fortran IV treats the first character of a record in an input file as data.

The printing control characters are:

0	Print after double new line
1	Print after form feed
Blank	Print after new line

For examples, see Table 5.1.

Suppressing a New-line Character (Z)

Normally, a New-line character is the last character in a record output to a file. You use the **Z** format code to suppress the New-line character on output of the current record. It must appear in the format specification before the slash that indicates the end of that record, or before the terminating parenthesis of the format specification.

For examples, see Table 5.1.

Program Statement	Resulting Output
Given Datum = 125	
1. FORMAT (I4)	125 (after one line feed)
FORMAT (1X, I3)	
FORMAT ("□ ", I3)	
FORMAT ('□ ', I3)	
FORMAT (1H□ , I3)	
2. FORMAT (I3)	25 (after one form feed)
3. Given I = 249 and J = 133	
WRITE (12, 9) I	
WRITE (12, 10) J	
9 FORMAT ("□ ", I6,Z)	
10 FORMAT (I6)	□□□ 249
	□□ 133
4. WRITE (12.9) I	
WRITE (12.9) J	
9 FORMAT ('□ ', I6,Z)	□□□ 249□□□ 133

Table 5.1 Carriage Control

Field Separators

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

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

is equivalent to:

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

Use of repeated slashes allows you to either bypass input records or output blank records. The first slash terminates input or output of the current record and the remaining slashes indicate blank or skipped records. However, if repeated slashes appear at the beginning or end of a format specification, each slash indicates a blank or skipped record; the left-hand parenthesis of the format specification initiates a new record and the right-hand parenthesis terminates the current record.

For example, the following statements:

```
      WRITE (12, 100)
100 FORMAT (1X, T21, "CHAPTER6"///"□ ",
     1"TASKS"//"□ ", "TEXT"//)
```

result in the output:

```
      CHAPTER 6
(blank line)
(blank line)
(blank line)
TASKS
(blank line)
TEXT
(blank line)
(blank line)
```

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

Although you may separate all field descriptors, separation is not mandatory if MP/Fortran IV can identify two field descriptors clearly.

For example:

```
9    FORMAT (I4 "DATA□ ARE:" E15.5)
```

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

FORMAT:and I/O Statement Association

Each non-**COMPLEX** data element in a formatted input or output statement list corresponds to a single entity in the associated format specification. Each **COMPLEX** and **DOUBLE PRECISION COMPLEX** datum corresponds to two format items (see discussion below). Except for the effect of repeat specification, MP/Fortran IV scans the format specification and the corresponding I/O list from left to right. It associates the field descriptors of the format specification with the entities of the I/O list in the order in which you wrote them.

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

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

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

Whenever MP/Fortran IV encounters an **I, F, E, D, G, L, A,** or **S** field descriptor in a format specification, it determines if there is a corresponding entity in the I/O list (each descriptor corresponds to one I/O list entity except for complex data). If it finds one, it performs the conversion and passes control to the next field descriptor. If there is no corresponding entity, format control terminates.

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

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

Complex Data in I/O Lists

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

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

On output, MP/Fortran IV outputs the internal value according to a repeated field descriptor (such as 2**F**10.2) or two or more successive field descriptors (you may specify string constants and **H, X**, and **T** field descriptors between the two field descriptors that govern the structure of the complex datum).

Multiple-Record Formatting

MP/Fortran IV can output more than one record using a single **FORMAT** statement if either of the following conditions exists:

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

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

For example:

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

can be written as:

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

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

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

For example:

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

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

Examples

Table 5.2 (at the end of this chapter) lists examples showing how format control processes the output of various I/O statements.

Run-Time Format Specifications

I/O statements can reference an array containing a formatting specification, rather than a **FORMAT** statement. This allows formatting information to be read in at run-time and changed for different data.

The formatting array contains a format specification, including the zero-level left and right parentheses, but not the word **FORMAT**. The character string that is the format specification can be stored in an array by use of a formatted **READ** that uses a format containing **A***w* or **S***w* descriptors.

To use run-time formatting, you must:

1. Determine how large an array will be needed for the largest incoming format specification.
2. Dimension the array in a **DIMENSION**, **COMMON**, or type declaration statement.
3. Include an appropriate storage statement or statements. Most commonly, this will be a **READ** statement and a **FORMAT** statement that will read the format specification into array storage using **A***w* or **S***w* descriptors.
4. Reference the format array in the **READ** statement used for input of data.
5. Supply formatting information to be read into the array at run-time.

For example, using the **A***w* format:

```
      INTEGER IFT(12)
2     FORMAT (12A2)
      READ (11, 2)(IFT(I), I = I, 12)
      READ (11, IFT) J, W, X, Y, Z, (C(I), I = 1, 7)
```

Or using the **S***w* format:

```
      DIMENSION FT(12)
2     FORMAT (S47)
      READ (11, 2) FT (1)
      READ (11, FT) J, W, X, Y, Z, (C(I), I = 1, 7)
```

The information supplied at run-time might be:

```
(I3, 4E15.6/7F10.3)
```

The number of characters stored in the example, including the terminating null, is 20, a total below the 24-character maximum in the integer array IFT and well below the 48-character maximum allowed in the real array FT.

The elements in the formatting array must be contiguous. Exercise caution when you input the formatting array with **A***w* specifications, as only two characters per element will be stored even though the element (e.g., a real variable) may be more than one word long.

Summary of Rules for Formatted I/O

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

Input

- MP/Fortran IV ignores leading blanks input to a numeric field. Embedded and trailing blanks have a value of zero. An input field containing all blanks has a value of zero.
- A decimal point entered in a real input field overrides the decimal point position specified by *d* of the field descriptor.
- If the field width (*w*) specified in the field descriptor is greater than the field you input, the field is treated as if the rest of the characters were blanks.
- If the field width (*w*) specified in the field descriptor is smaller than the input field, *w* characters are input.
- An exponent in an input field does not override the decimal point implied by its field descriptor (*d*); however, the exponent does override any given scale factor.

Output

- The field width specifier (*w*) in a field descriptor must be large enough to indicate (where applicable) a sign, a decimal point, and an exponent; if it is not, MP/Fortran IV will print *w* asterisks.
- In field descriptors where you specify the field width (*w*) and the decimal point position (*d*), *w* must be greater than *d*, or MP/Fortran IV will signal an error by filling the field with asterisks.
- If the number of digits following the decimal point in the internal representation of a real

number is greater than d in the field descriptor, MP/Fortran IV rounds the number.

- The first character of each record in the specification is a carriage control character; MP/Fortran IV does not print it.
- If the number of non-blank characters to output is less than that specified in the field descriptor (w), MP/Fortran IV right justifies the characters, and precedes them by the appropriate number of blanks.

```
10 FORMAT (3E10.3, (I2, 2(F12.4, F10.3)), D20.12)
           ↑     ↑  ↑        ↑↑       ↑
           0     1  2        21       0
```

Figure 5.1 Nested parentheses example

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

Format control examples
(Given A=2.75, B=51.70, C=(1.0, 2.0), I=22, J=39, K=6, and L=132),2

Chapter 6
Classification of MP/Fortran IV Instructions

This chapter is designed to serve as a quick reference. If you want to know what commands are available for I/O or the name of a particular run-time routine, a quick perusal of these tables should enable you to find what you need. Each table contains all the MP/Fortran IV elements that perform one kind of operation, along with a simple definition and a syntax description for each one.

If you need more information, the next chapter, the MP/Fortran IV Dictionary, provides complete information on all of the MP/Fortran IV language elements. It explains each command in more detail, describes alternate or optional syntax components, and suggests special uses for some commands. Each dictionary entry includes an example of the MP/Fortran IV command used in a program statement.

Control Statements

Control statements allow the programmer to change the flow of program logic. The MP/Fortran IV control statements are summarized in Table 6.1.

Syntax	Meaning
(unconditional) **GO TO** statement-label	Causes transfer to a specified statement label.
(assigned) **GO TO** variable	Causes transfer to the address which is the current value of the specified variable.
(computed) **GO TO** (statement-label$_1$, statement-label$_2$,... statement-label$_n$ variable)	Causes possible transfer to one of several statement labels depending on the value of the specified variable.
(assigned) **GO TO** variable (statement-label$_1$, statement- label$_2$,..., statement-label$_n$)	Causes transfer to one of several possible statement labels depending on the value of the specified variable after the last execution of an ASSIGN statement.
ASSIGN statement-label TO variable	Causes a subsequent assigned GO TO statement to transfer control to the statement label specified within the ASSIGN statement.
IF (logical-expression) statement	Causes either execution or bypassing of the specified statement depending on the truth value of the specified logical expression
IF (expression) statement-label$_1$, statement-label$_2$, statement-label$_3$	Causes transfer to one of the three statement labels depending on the value of the specified expression.
CALL subroutine (argument, argument,..., argument)	References a specified subroutine, replacing dummy arguments with actual arguments.
CALL subroutine	References a specified subroutine.
RETURN [variable]	Indicates the logical end of a subprogram, causing a normal return when executed. Optionally, the user may cause an abnormal return by specifying a variable.
CONTINUE	Causes continuation of the normal execution sequence.
PAUSE [string]	Causes the program to cease execution with an optional message printed at the console.
STOP [string]	Causes an unconditional termination of a program's (or a task's) execution, and optionally causes a message to be printed at the console.
DO statement-label variable = integer, integer [,integer]	Sets up a programming loop.
CALL EXIT	Terminates the executing program.

Table 6.1 Syntax of control statements and routines

I/O Statements and Routines

Syntax	Meaning
READ (channel) [list-of-variables] **READ** (channel, format) [list-of-variables]	Reads from a device or file the data associated with the variables in the list; formatting may be preset (unformatted I/O) or in accordance with a FORMAT specified by the user.
READ (channel, [format,] {ERR} = statement-label) [list] {END} **READ** (channel, [format,] {ERR} = statement-label, {END} {ERR} = statement-label) [list] {END}	Reads information (as in READ description above) and also allows the user to gain control after an end-of-file or an I/O error at the driver level has been detected.
WRITE (channel) [list-of-variables] **WRITE** (channel, format) [list-of-variables]	Writes to a device or file the data associated with the variables in the list; formatting may be preset (unformatted I/O) or in accordance with a FORMAT specified by the user.
WRITE (channel, [format,] {END} = statement-label) [list] {ERR} **WRITE** (channel, [format,] {END} = statement-label, {END} = statement-label) [list] {ERR} {ERR}	Writes to a device or file the data associated with the variables in the list, and also allows the user to gain control after an end-of-file or I/O error has been detected.
statement-label **FORMAT** (specification)	Allows for the formatting of input and output data according to a specification.
ACCEPT list	Values for variables appearing within the list of the ACCEPT statement are read from the console.
(list, controlvar = lowerbound, upperbound [,increment])	DO-IMPLIED list specifies a DO-loop with respect to all, or a portion, of the list of variables associated with an I/O statement.
READ BINARY (channel) list	Transfers binary data from an external medium.
WRITE BINARY (channel) list	Transfers data in binary to an external medium.
REWIND channel	Causes the file associated with the specified channel to be positioned at the initial record.
ENDFILE channel	Closes the file associated with the specified channel.
CALL FSEEK (channel, recordnumber)	Positions a random file to a given record.
CALL CHSAV (channel, start-word)	Saves the status of a channel to enable rereading or rewriting of records.
CALL CHRST (channel, start-word)	Restores previously saved channel status to enable rereading and rewriting of records.
CALL RDBLK (channel, sblock, array, nblock, error [,iblk])	Reads a series of blocks from a file.
CALL READR/RDRW (same) (channel, srec, array, nrec, error [,nbyte])	Reads a series of records from a file into an array.
CALL WRITR/WRTR (same) (channel, srec, array, nrec, error [,nbyte])	Writes a series of records to a file.
CALL WRBLK (channel, sblock, array, nblock, error [,iblk])	Writes a series of blocks into a disc file from an integer array.

Table 6.2 Syntax of I/O statements and routines

I/O statements and routines allow externally recognizable ASCII or binary characters to be read and converted to their internal computer representation and vice versa.

The programmer may choose to control the internal and external representations with formatting information (described more fully in the Chapter 5).

These MP/Fortran IV elements also control the positioning of devices and the actions taken on detecting the ends of input and output. Table 6.2 summarizes I/O statements .

Program Structure

Table 6.3 summarizes the statements that define simple program and array structures.

Syntax	Meaning
SUBROUTINE name (argument,..., argument	Defines a subroutine subprogram unit.
type **FUNCTION** name (argument,... , argument	Defines a function subprogram.
functionname(dummyargument, ..., dummyargument) = expression	Defines a single-statement, user- written function, internal to the program unit.
EXTERNAL subprogram-name, ..., subprogram-name	Specifies subprograms as external to the program unit in which the specification is made.
BLOCK DATA	Defines a subprogram which contains only DIMENSION, DATA, COMMON, data-type, and EQUIVALENCE statements.

Table 6.3 Syntax of program structure statements

Data Declarations and Initialization

Data-type declarations are non-executable statements that provide the MP/Fortran IV compiler with information about the names, storage allocation, and data-types of simple variables and arrays.

The MP/Fortran IV data initialization statements define initial values for variables and array elements stored in a labeled **COMMON** area.

Table 6.4 summarizes the data declarations and initialization statements.

Syntax	Meaning
INTEGER variable, variable,... variable	Specifies integer variables and arrays. The arrays may be dimensioned in the statement.
REAL variable,..., variable	Specifies real variables and arrays. The arrays may be dimensioned in the statement.
DOUBLE PRECISION variable, variable,..., variable	Specifies DOUBLE PRECISION variables and arrays. The array may be dimensioned in the statement.
COMPLEX variable, variable,..., variable	Specifies single precision complex variables and arrays. The arrays may be dimensioned in the statement.
DOUBLE PRECISION COMPLEX variable, variable,..., variable	Specifies DOUBLE PRECISION COMPLEX variables and arrays. The arrays may be dimensioned in the statement.
LOGICAL variable, variable,... variable	Specifies LOGICAL variables and arrays. The arrays may be dimensioned in the statement.
COMPILER DOUBLE PRECISION variable, variable..., variable	Forces all REAL variables and and constants to DOUBLE PRECISION and all COMPLEX to DOUBLE PRECISION COMPLEX.
COMMON name ... name	Specifies names of variables and arrays to be placed in unlabeled COMMON. The arrays may be dimensioned in the statement.
COMMON block-name/list of names .../block-name/list of names	Specifies lists of arrays and variables to be placed in labeled COMMON areas defined by block names.
EQUIVALENCE (list-of-names), (list-of-names),...(list-of-names)	Determines shared storage for variables and arrays.
PARAMETER variable = constant, ... variable = constant	Assigns values to symbolic names, which may then be used as constants throughout the program.
DATA variable-list/constant-list/ ... variable-list/constant-list/	Defines initial values for variables and array elements.
DIMENSION arrayname (subscript bounds),...,arrayname (subscript bounds)	Specifies the subscript bounds of arrays for allocation of storage to the arrays.
COMPILER NOSTACK	Causes all non-COMMON variables and arrays to be placed in a fixed location in memory rather than on a run-time stack.

Syntax of data declarations and initialization

Built-In Functions

MP/Fortran IV includes many built-in library functions, which are supplied with the compiler. For the summary tables, we have divided them into three categories:

- Trigonometric operations (Table 6.5)
- Arithmetic and conversion operations (Table 6.6)
- Bit/word manipulation and testing (Table 6.7)

Name	Definition	Type of Argument	Function
ATAN(x)		Real	Real
DATAN(x)	Arctangent	Double	Double
ATAN2(x)		Real	Real
DATAN2(x)	Arctangent	Double	Double
COS(x)	Trigonometric	Real	Real
DCOS(x)	Cosine	Double	Double
CCOS(x)		Complex	Complex
DCCOS(x)		DP Complex	DP Complex
SIN(x)	Trigonometric	Real	Real
DSIN(x)	Sine	Double	Double
CSIN(x)		Complex	Complex
DCSIN(x)		DP Complex	DP Complex
SINH(x)	Hyperbolic Sine	Real	Real
TAN(x)	Trigonometric	Real	Real
DTAN(x)	Tangent	Double	Double
TANH(x)	Hyperbolic	Real	Real
DTANH(x)	Tangent	Double	Double

Table 6.5 Trigonometric operations

Name	Definition	Type of Argument	Type of Function
ABS(x)	Absolute	Real	Real
IABS(x)	value	Integer	Integer
DABS(x)		Double	Double
CABS(x)	Complex	Complex	Real
DCABS(x)	modulus	DP Complex	Double
AIMAG(x)	Obtain imaginary	Complex	Real
DAIMAG(x)	part of complex argument	DP Complex	Double
DINT(x)	Truncation	Double	Double
AINT(x)		Real	Real
INT(x)		Real	Integer
IDINT(x)		Double	Integer
ALOG(x)	Natural	Real	Real
DLOG(x)	logarithm	Double	Double
CLOG(x)		Complex	Complex
DCLOG(x)		DP Complex	DP Comple
$ALOG_{10}(x)$	Common	Real	Real
$DLOG_{10}(x)$	logarithm	Double	Double
AMAX0(x1,...xn)	Choosing	Integer	Real
AMAX1(x1,...xn)	largest	Real	Real
MAX0(x)1,...xn)	value	Integer	Integer
MAX1(x)1,...xn)		Real	Integer
DMAX1(x1,...xn)		Double	Double
AMIN0(x1,...xn)	Choosing	Integer	Real
AMIN1(x1,...xn)	smallest	Real	Real
MIN0(x)1,...xn)	value	Integer	Integer
MIN1(x)1,...xn)		Real	Integer
DMIN1(x1,...xn)		Double	Double
AMOD(x1,x2)	Modulus	Real	Real
MOD(x1,x2)		Integer	Integer
DMOD(x1,x2)		Double	Double
CMPLX(x1,x2)	Express two real	Real	Complex
DCMPLX(x)	arguments in complex form	Double	DP Complex
CONJG(x)	Obtain conjugate of	Complex	Complex
DCONJG(x)	complex argument	DP Complex	DP Complex

Arithmetic and conversion operations

Name	Definition	Type of Argument	Type of Function (cont'd)
DBLE(x)	Express single precision argument in double precision form	Real	Double
DIM(x)	Positive	Real	Real
IDIM(x)	difference	Integer	Integer
EXP(x)	Exponential	Real	Real
DEXP(x)		Double	Double
CEXP(x)		Complex	Complex
DCEXP(x)		DP Complex	DP Complex
FLOAT(x)	Convert from	Integer	Real
DFLOAT(x)	integer to real	Integer	Double
IFIX(x)	Convert from real to integer by truncation	Real	Integer
REAL(x)	Obtain real part of	Complex	Real
DREAL(x)	complex argument	DP Complex	Double
SIGN(x)	Transfer of sign	Real	Real
ISIGN(x)		Integer	Integer
DSIGN(x)		Double	Double
SNGL(x)	Obtain most significant part of double precision argument	Double	Real
SQRT(x)	Square root	Real	Real
DSQRT(x)		Double	Double
CSQRT(x)		Complex	Complex
DCSQRT(x)		DP Complex	DP Complex
CALL OVERFLOW ($statement_label, $statement_label, [< 'S' \ 'N' >])	Checks for floating point overflow.		

Table 6.6 Arithmetic and conversion operations (continued)

Function or Routine	Definition	Type of Argument	Type of Function
IAND(x1,x2)	Bit-by-bit ANDing	Integer	Integer
IOR(x1,x2)	Bit-by-bit ORing	Integer	Integer
NOT(x)	Logical Complement	Integer	Integer
IEOR(x1,x2)	Bit-by-bit Exclusive OR	Integer	Integer
ISHFT(x1,x2)	Shift arg_1 by the number of bits given in arg_2, where: $arg_2 < 0$ right shift $arg_2 = 0$ no shift $arg_2 > 0$ left shift	Integer	Integer
ITEST(x1,x2) BTEST(x1,x2)	Test a bit within the word given by arg_1. The bit tested is specified by arg_2. The result returned is: .FALSE. if tested bit = 0 .TRUE. if tested bit = 1	Integer	Logical
CALL BSET (word, position)	Sets a single bit in a		
CALL ISET(word, position)	word to zero.		
CALL BCLR (word, position)	Sets a single bit in a		
CALL ICLR (word, position)	word to one.		

Table 6.7 Bit/word manipulation and testing

System, Directory, and Device Control

Table 6.8 summarizes the MP/Fortran IV calls that interface to system directory commands. These commands are described in Chapter 7, but if you need more information on the MP/OS system and on the operation of these commands, see Part 3, *MP/OS Assembly Language Programmer's Reference*, (DGC No. 093-40001).

Syntax	Meaning
CALL BOOT (pathname, error)	Perform a disc bootstrap.
CALL DIR (directoryname, error)	Changes the current working directory.
CALL CDIR (directoryname, error)	Creates a subdirectory.
CALL GDIR (array, error)	Returns the name of the current default directory/device.
CALL FINTD (device-code, dct)	Specifies a device which is capable of generating interrupt requests.
CALL FINRV (device-code)	Removes a user interrupt device from the system interrupt vector table.
CALL SYS (call-number, option, ac0, ac1, ac2, error)	Executes any MP/OS system call from the MP/Fortran IV environment.
WORDADDR (identifier)	Returns the address of the first word of any identifier.
BYTEADDR (identifier)	Returns the address of the first byte of any identifier.

Table 6.8 System, directory and device control calls

File Maintenance and I/O Control

Table 6.9 summarizes the MP/Fortran IV calls that interface to system file I/O and file management commands.

Syntax	Meaning
CALL CFILW (pathname, type, [size,] error)	Creates a disc file.
CALL DFILW (pathname, error)	Deletes a disc file.
CALL DELETE (pathname)	Deletes a disc file.
CALL RENAM (oldpathname, newpathname, error)	Renames a disc file.
CALL OPEN (channel, pathname, mode, error, [size])	Assigns a specified channel to a disc file and opens the file.
CALL FOPEN (channel, pathname [,"B"] [,recordbytes])	Assigns a specified channel to a file (device) and opens that file or device.
CALL APPEND (channel, pathname, < mode, | array, > error [,size])	Opens a file for appending.
CALL CLOSE (channel,error)	Closes a file, and frees its associated channel.
CALL FCLOS (channel)	Closes a file on a specified channel and frees the channel.
CALL RESET	Closes all open files.

Table 6.9 File maintenance and I/O control calls

Tasking

Table 6.10 summarizes the MP/Fortran IV calls that interface to system multitask programming facilities.

Syntax	Meaning
TASK taskname	Assigns a name to a task program unit.
CHANTASK number-of-channels, number-of-tasks	Specifies the number of channels that may be open at one time (maximum: 16), and the number of tasks which can be simultaneously active at one time.
CALL FTASK (taskname, $error-return, priority-number [,IASM])	Activates a task by task name.
CALL ITASK (taskname, id, priority-number, error [,IASM])	Activates a task and associates an id number with the task name.
CALL PRI (priority-number)	Changes the priority number of the executing task.
CALL CHNGE (id, priority-number, error)	Changes the priority number of a specified task.
CALL KILL	Terminates the calling task.
CALL ABORT /DESTR(id, error)	Terminates the task with the specified identification number.
CALL XMT (message-key, message-source, $error-return)	Transmits a one-word message between active tasks.
CALL REC (message-key, message-destination)	Receives a one-word message.
CALL XMTW (message-key, message source, $error-return)	Transmits a one-word message between active tasks and waits until the message has been received.
CALL SINGL	Disables scheduling of all other tasks.
CALL MULTI	Re-enables scheduling of all other tasks.

Table 6.10 Syntax for tasking calls

Swapping, Chaining, and Overlays

Table 6.11 shows the MP/Fortran IV calls the programmer can use to subdivide a large program that would exceed the limits of memory if allowed to reside in memory in its entirety.

Syntax	Meaning
CALL SWAP (filename, error)	Saves the current program's memory image on disc, and loads another program into memory from disc.
CALL FSWAP (filename)	Saves the current program's memory image on disc, and loads another program from disc.
CALL BACK	Restores to memory the last program that was swapped out to disc.
CALL FBACK	Restores to memory the last program that was swapped out to disc.
CALL EBACK (error)	Returns the last-swapped program to memory; if it is the CLI, the error is typed on the console.
CALL CHAIN (filename, error)	Causes the current program's memory image to be overwritten by another program from disc.
CALL FCHAN (filename)	Causes current program's memory image to be overwritten by another program from disc.
OVERLAY overlayname	Defines an overlay.
CALL OVLOD/FOVLD (channel, overlayname, conditional-flag, error)	Loads an overlay.
CALL OVREL/FOVRL (overlay, error)	Releases an overlay.

Table 6.11 Swapping, chaining and overlay calls

Real Time Clock and Calendar

Table 6.12 lists the MP/Fortran IV calls that interface to the system real time clock and calendar, and allow real-time control of program tasks.

Syntax	Meaning
CALL TIME (time-array, error)	Gets the current time of day.
CALL STIME (array, error)	Sets the time of day.
CALL FGTIM (hour, minute, second)	Gets the current time.
CALL FSTIM (hour, minute, second)	Sets the real time clock.
CALL DATE (date-array, error)	Gets the current date.
CALL SDATE (array, error)	Sets the date.
CALL WAIT (time, units, error)	Suspends an executing task for specified amount of time.
CALL FDELY (milliseconds)	Suspends a task for a specified amount of time.

Table 6.12 Real time clock calendar calls

Chapter 7
Command Dictionary

ABORT (subroutine)

Terminates a task specified by its identification number.

Syntax

CALL ABORT*(id, error)*

where: *id* is an integer variable, constant, or array element specifying the identification number of the task. (This number was previously assigned in an ITASK call.)

error is an integer variable that returns a MP/Fortran IV or run-time error code on completion of the call.

Example

```
CALL ABORT(127, IER)        TERMINATE TASK 127
```

Notes

This routine is the same as the routine **DESTR**.

ABS (function)

Computes the absolute value of any real number. Returns a real value.

Syntax

ABS*(e)*

where: *e* is any real constant, variable, or expression.

Examples

```
W = 6.0
X = 6.0
Y = ABS(W)        ; Y = 6.0
Z = ABS(X)        ; Z = 6.0
```

Notes

ABS does not generate error messages.

ACCEPT (statement)

Reads values from the standard input channel (normally open to **?inch**.)

Syntax

ACCEPT *list*

where: *list* is a list of any combination of variables, arrays, array elements, Hollerith strings, and implied-DO loops (see DO-IMPLIED LIST), separated by commas.

Example

```
ACCEPT "INITIAL RANDOM NUMBER =", RN1
```

On execution of this line, the program will type

```
INITIAL RANDOM NUMBER =
```

to the standard output channel (**?ouch**), and the user must respond with a value and a newline character.

Notes

On execution of the **ACCEPT** statement, any string constants given in the I/O list of the **ACCEPT** statement will be typed one at a time at the console and can serve as prompts for the required input values. The program then reads in the values when they are typed by the programmer. The values may be separated by commas or newlines.

In the I/O list, a newline, if used, must be the last character in a Hollerith string, since it causes the operating system to terminate output.

When Hollerith strings are interspersed with variables, the program will type the first string, and then wait until values have been typed in for all the variables up until the next string in the list. The user must then type a newline character to force output of the next string.

ACCEPT will convert integers to real or double precision if the data type of the internal variable requires it.

ACCEPT statements can transfer entire arrays, or array elements with variable or constant subscripts:

```
ACCEPT "PLEASE TYPE IN DATA: ", RARRAY
```

causes the program to wait for values for every element of the array RARRAY. Perhaps more reasonably:

```
ACCEPT "PLEASE TYPE IN LIMIT, DATA: ", N,
(RARRAY(I), I = 1, N)
```

asks the programmer to fill the first N elements of the array RARRAY.

The **ACCEPT** statement reads from channel 11.

ACCEPT uses channel 11 no matter what it is open to. For example:

```
CALL CLOSE (II,IER) ; CLOSE PRE-OPENED CHANNEL
CALL OPEN(11, "TFILE", . . .)
ACCEPT "SHALL I GO ON?", IANS
```

ACCEPT goes to **"TFILE"**, rather than to the console, to get a value for **IANS**. The call to **CLOSE** is required, since channel 11 is preopened to **?Inch**.

AIMAG (function)

Returns the imaginary part of a single precision complex number as a real result.

Syntax

AIMAG*(e)*

where: *e* is a single precision complex constant, variable, or expression.

Example

```
SCIENCE = (8.76, .125)
DREAM = AIMAG(SCIENCE) ; DREAM = .125
```

Notes

No error messages are generated.

AINT (function)

Truncates the fractional part of a single precision real number, returns a real result.

Syntax

AINT*(e)*

where: *e* is a single precision real constant, variable, or expression.

Example

```
X = 87.65
WHOLE = AINT(X) ; WHOLE = 87.00
```

You can "round off" a real number by:

```
WHOLE = AINT(X + 0.5) ; WHOLE = 88.00
```

Notes

Result returned is a real number with the value:

*sign of argument * largest integer <= |argument|*

AINT does not generate error messages.

ALOG (function)

Computes the single precision real natural logarithm of a single precision real positive argument.

Syntax

ALOG*(x)*

where: *x* is a positive single precision real constant, variable, or expression.

Example

```
TANG = EXP(U)
T = ALOG(TANG) ; T = U
```

Notes

In the case of a zero argument, **ALOG** returns an error message and returns the largest possible real number, $7.2*10^{75}$, as a result.

In the case of a negative argument, **ALOG** returns an error message and computes the logarithm of **ABS***(argument)*.

ALOG10 (function)

Computes the single precision real base 10 logarithm of a single precision real positive argument.

Syntax

ALOG10*(x)*

where: *x* is a positive single precision real constant, variable, or expression.

Example

```
A = 10 ** B
C = ALOG10(A) ; B = C
```

Notes

In the case of a zero argument, **ALOG10** generates an error message, and returns the largest possible real number, $7.2 * 10^{75}$, as an answer.

In the case of a negative argument, **ALOG10** returns an error message and computes the logarithm of **ABS***(argument)*.

AMAX0 (function)

Selects the largest member from a set of integers, expressing the selection as a single precison real value.

Syntax

AMAX0$(i_1, i_2, \ldots i_n)$

where: i is an integer constant, variable, array element, or expression.

Example

```
X = AMAX0(11,34,22,12,27,29) ;X will be set equal to
34.0 .
```

Notes

AMAX0 does not generate error messages.

AMAX1 (function)

Selects the largest member from a set of single precision real numbers, expressing the selection as a single precison real number.

Syntax

AMAX1$(x_1, x_2, \ldots x_n)$

where: x is a single precision real constant, variable, array element, or expression.

Example

```
X = AMAX1(11.5, 11.376, 12.0, 11.1, 11.8) ; X will
be set equal to 12.0 .

UPPER = (CURRENT, BOUND)
```

Notes

AMAX1 does not generate error messages.

AMIN0 (function)

Selects the smallest member from a set of integers, expressing the selection as a single precison real value.

Syntax

AMIN0$(i_1, i_2, \ldots i_n)$
where: i is an integer constant, variable, array element, or expression.

Example

```
SMALL = (LIST1,LIST2)
X = AMIN0(11,34,22,12,27,29) ;X will be set equal to
11.0 .
```

Notes

AMIN0 does not generate error messages.

AMIN1 (function)

Selects the smallest member from a set of single precision real numbers, expressing the selection as a single precison real number.

Syntax

AMIN1$(x_1, x_2, \ldots x_n)$
where: x is a single precision real constant, variable, array element, or expression.

Example

```
SMALL = AMIN1(S1,S2)
X = AMIN1(11.5, 11.376, 12.0, 11.1, 11.8)
   ;X will be set equal to 11.1 .
```

Notes

AMIN1 does not generate error messages.

AMOD (function)

Returns the real remainder in the quotient of two single precision real arguments, x/y.

Syntax

AMOD(*x,y*)

where: *x* and *y* are single precision real constants, variables, array elements, or expressions.

Example

```
START = 10.0 * AMOD(T1,T2)
```

If **T1**=3.0 and **T2**=5.0, **START** receives the value 30 .

NOTES

AMOD is defined as:

arg_1 - $[arg_1/arg_2]$ * arg_2,

where: $[arg_1/arg_2]$ is the truncated value of that quotient.

If the quotient causes overflow or underflow, **AMOD** generates an error message and will not return a meaningful result.

APPEND (subroutine)

Opens a file for appending.

Syntax

CALL APPEND(*channel, pathname, < array | mode >, error* [,*size*])

where: *channel* is an integer variable or constant whose value specifies the number of the channel ($0 - 63_{10}$) on which *pathname* receives the appended material.

pathname is the name of the file to be opened for appending.

mode and *array* (alternate arguments in the command line), are ignored by MP/Fortran IV. They appear in the command line to maintain program portability among AOS, RDOS, and MP/OS. (*Mode* is an integer constant or variable, while *array* is a three-element integer array.)

error is an integer variable which will return one of the MP/Fortran IV or run-time error codes on completion of the call.

size is an integer constant or variable specifying the number of bytes that make up a record of a randomly organized file. You must give *size* if the file is not accessed sequentially.

Examples

```
CALL APPEND(5, "SQRT", 2, IERR, ISIZ)
```

Notes

When a call to **APPEND** is executed, the end of the file *pathname* is located and *pathname* is opened on the specified channel.

Only one channel at a time may be open to an input character device such as the console keyboard input.

ASSIGN (statement)

Gives a variable the value of a specified line number.

Syntax

ASSIGN *n* **TO** *v*

where: *n* is a statement number.

v is a non-subscripted integer variable name that appears in an assigned **GO TO** statement.

Example

```
ASSIGN 50 TO VOLUME
   .
   .
   .
IF (HGT .GT. 0) GO TO VOLUME
   .
   .
   .
50 CONTINUE
VOL = HGT * XLEN * WDTH
```

Notes

If *n* is a statement number that does not label any statement in the program, the compiler will produce "undefined label" errors.

ATAN (function)

Computes the real arctangent of a real argument. The value returned will be in the range: -pi <= **ATAN**(*x*) <= pi.

Syntax

ATAN(*x*)

where: *x* is a real constant, variable, array element, or expression.

Example

```
E = ATAN(F)
G = TAN(E)  ; E = G
```

Notes

The result returned is in radians.

ATAN2 (function)

Computes the real arctangent of the quotient of two arguments, e_2/e_1. The value returned is in the range:
-pi <= **ATAN2**(e_1, e_2) <= pi

Syntax
ATAN2(e_1, e_2)
where: e_1 and e_2 are real variables, expressions, or array elements, and e_2 is not equal to zero.

Example
```
ANGLE1 = ATAN2(XNUM, 1)
ANGLE2 = ANGLE1 + ATAN2(X + PI, Y)
```

Notes
Overflow is possible as the divisor s_2 approaches zero. In the case of overflow, **ATAN2** returns + or - pi.

If $e_2 = 0$, the compiler returns an error message.

The result returned is in radians.

BACK (subroutine)

Returns to the next lower program level.

Syntax
CALL BACK

Notes
BACK, **FBACK** and **EXIT** are identical. They all perform an assembly language **?RETURN** call, with all arguments equal to zero. See the *MP/OS Assembly Language Programmer's Reference* for details of program management. **EBACK** and **STOP** provide other ways of terminating the program.

BACK does not pass **COMMON**.

BLOCK DATA (subprogram)

Defines initial values for variables in labeled **COMMON**. (See also **DATA** statement in this dictionary.)

Syntax

The subprogram begins with the statement **BLOCK DATA** and terminates with an **END**.

The subprogram may contain only **DIMENSION, DATA, COMMON**, data type specification, and **EQUIVALENCE** statements.

Example

```
BLOCK DATA
COMMON /ELN/C,A,B/RMC/Z,Y
DIMENSION B(4), Z(3)
DOUBLE PRECISION Z
COMPLEX C
DATA B(1),B(2)/2*1.1/C/2.4,3.769/Z(1)/7.649D5/
END
```

Notes

If labeled **COMMON** is used in a **BLOCK DATA** subprogram, its **COMMON** statement must include *all* the variables from that labeled **COMMON** block, even if only some of its variables are initialized to values in a **DATA** statement.

To include your **BLOCK DATA** subprogram in your program, treat it as you would any other subprogram, i.e:

Give the subprogram file a name, such as BD.FR. Compile it to produce BD.OB. Then include the name BD in your **BIND** command line. The binder will combine the subprogram with your main program, other subprograms you may have supplied, and the MP/Fortran IV library, to produce the program.PR, which you may then execute.

BLOCK DATA and **DATA** statements may be used if you are placing your program in ROM or PROM, but will involve extra overhead.

BOOT (subroutine)

Shuts down the current system; bootstraps a new system if a non-null file name is supplied.

Syntax

CALL BOOT*(pathname, error)*

where: *Pathname* is the name of a device or a file (for a stand alone program).

Error is in integer variable that returns one of the error codes on completion of the call.

Example

```
CALL BOOT("@DPX1", IER)
```

Notes

If *pathname* is null, this call simply shuts down the system in an orderly fashion.

This routine calls the MP/OS system call, **?BOOT**. See Part 3 *MP/OS System Assembly Language Programmer's Reference*, (DGC 093-400001) for more information.

BYTEADDR (function)

Returns the byte address of any identifier.

Syntax

BYTEADDR*(identifier)*

where: *identifier* is any legal subprogram or variable identifier.

Example

```
NADDR = BYTEADDR(CALCTASK)
```

Notes

The function **BYTEADDR** takes an argument of any data type. The address returned is always an integer value.

This function is especially useful for providing needed information for system call routines.

CABS (function)

Computes the absolute value of a single precision complex number. Returns a single precision real number.

Syntax

CABS*(arg)*

where: *arg* is any single precision complex constant, variable, or expression.

Examples

```
COMPLEX SC, CX
.
.
.
SC = CABS(CX)
```

Notes

The function may generate stack overflow messages.

CABS is defined as a complex modulus operation:

```
SQRT( REAL(arg) ** 2. + IMAG(arg) ** 2. )
```

CALL (statement)

References the specified subroutine.

Syntax

CALL *subr*

CALL *subr*(a_1, a_2, . . . , a_n)

where: *subr* is the name of a subroutine or of a dummy variable, and

a_1, a_2, ..., a_n are actual argument names that replace the dummy argument names in the subroutine.

Examples

```
CALL OPTIONS

CALL QUAD(9.73, Q/R, 5, R-S**2.0, X1, X2)
```

Notes

The statement references the designated subroutine, which is executed. Control is returned to the statement after the **CALL** statement when execution of the subroutine is completed, unless the subroutine makes an abnormal return.

In an abnormal return, the **RETURN** statement in the subprogram specifies that control will return to some line in the calling program. This line can be an executable statement anywhere in the calling program. (See **RETURN**.)

You may declare a user-defined function as external to your program, and then pass the function to a subprogram via a dummy argument. For example:

```
REAL ROOT
EXTERNAL ROOT
.
.
.
CALL MULT(A,B,ROOT)          ;These are the actual
                             ;arguments that will replace the
                             ;dummy arguments in the
                             ;subroutine subprogram.
-----------------            --------------------
SUBROUTINE MULT(Q,R,S)       ;Q, R, S are dummy arguments.
.
.
.
Q = S(Q,R)                   ;This generates a call to the
                             ;function ROOT, passed via
                             ;dummy argument S.
```

CCOS (function)

Computes the complex cosine of a single precision complex number.

Syntax

CCOS*(cx)*

where: *cx* is a single precision complex constant, variable, array element, or expression.

Examples

```
COM = CCOS((0.86, 0.05))
TAC = CCOS(T) + A
```

Notes

The function generates error messages on underflow or overflow.

For any x expressed in radians, cosine(x) = cosine($y + 2n$pi), where y is in the range $0 >= y < 2$pi; hence, if you evaluate the cosine of a large x, some of the information is wasted in specifying multiples of 2pi radians (360 degrees), which have no effect on the result. You will see this "waste" as a loss of significant digits in the answer.

CDIR (subroutine)

Creates a subdirectory.

Syntax

CALL CDIR*(name, error)*

where: *name* is the name of the subdirectory.

error is an integer variable that returns one of the MP/Fortran IV or run-time error codes on completion of the call.

Example

```
CALL CDIR(''SDIR'', IER)
```

CEXP (function)

Computes the complex value of e^x.

Syntax

CEXP*(x)*

where: *x* is any complex constant, variable, array element, or expression.

Example

```
COMPLEX C,B
C = CEXP(B)
```

Notes

If either underflow or overflow occurs, an error message is typed at the console, and the routine returns zero or the greatest possible real value ($7.2 * 10^{75}$), as a result.

If *x* is the input argument, the routine performs the following calculation:

$$e^x = x * \log_e 2 = 2^{(I + F)}$$

where *I* and *F* are the integral and fractional portions of the power whose base is 2. The argument *x* of e^x must be selected so that $I <= 175_8$.

In the case of very large *I* values, where $I > n*2^{16}$, the routine generates an error message.

CFILW (subroutine)

Creates a disc file.

Syntax

CALL CFILW*(pathname, type, [size,] error)*

where: *pathname* is the name the new file will have.

type is an integer constant or variable whose value indicates the type of file to be created:

1. Sequentially organized file
2. Randomly organized file
3. Contiguously organized file.

size is an integer constant or variable giving the size in number of blocks (256 words) of a contiguously organized file. *This argument is used only for type 3 (contiguous) files.*

error is an integer variable which returns one of the MP/Fortran IV or run-time error codes on completion of the call.

Example

```
CALL CFILW(''Y10'',3,20,IER)
```

Notes

The name *pathname* must be unique with respect to all other file names in the system.

CHAIN (subroutine)

Causes the current program's memory image to be overwritten by another program from disc.

Syntax

CALL CHAIN *(filename, error)*

where: *filename* is the name of the file to be executed next. Its execution level will be the same as that of the calling program.

error is an integer variable which will return one of the MP/Fortran IV or run-time error messages on completion of the call.

Example

```
CALL CHAIN("AA", IER)
```

Notes

There is no limit on the number of chains performed. Program chaining can be used to subdivide a program so large that it would exceed the limits of memory if it were to reside in memory in its entirety. Each chained-to program must contain a complete program.

Refer to the *MP/OS Assembly Language Programmer's Reference* for more information on chaining.

This routine does not pass **COMMON** under MP/Fortran IV.

CHANTASK (specification statement)

Specifies the maximum number of FORTRAN tasks which may be executing at any one time.

Syntax

CHANTASK *c,t*

where: *c* is the maximum number of channels, up to 16_{10}.

t is the maximum number of concurrently active FORTRAN tasks.

Example

```
CHANTASK 16, 10
```

Notes

The **CHANTASK** statement must precede all other statements in the main MP/Fortran IV program, except for the statements **COMPILER DOUBLE PRECISION, COMPILER NOSTACK,** and **OVERLAY**.

This statement must be used to prepare a multi-task program for execution. It must appear in the first MP/Fortran IV program unit.

If assembly tasks are used with a FORTRAN program, a **.TSK** statement should be included in the assembler code, to specify the total number of concurrently active tasks.

CHNGE (subroutine)

Modifies the priority number of the executing task.

Syntax

CALL CHNGE*(id, priority-number, error)*

where: *id* is the task's identification number (previously assigned by a call to **ITASK**).

priority-number gives the new priority of the task. This number must be in the range 0 - 255_{10}, where zero is the highest priority.

error is an integer variable that returns a MP/Fortran IV or run-time error code on completion of the call.

Example

```
CALL CHNGE(127, 59, IER)
```

Notes

The lower the number, the higher the priority.

You may change the priority of a task any number of times while it is active.

More than one task may have the same priority number. Tasks with the same priority are scheduled in a round robin fashion: their relative priorities are determined by how long each has been waiting for processing. Each time a task relinquishes control to the task scheduler, it is moved to the end of the queue of active tasks.

CHRST (subroutine)

Restores the previously saved status of a MP/Fortran IV I/O channel.

Syntax

CALL CHRST*(channel, start-word)*

where: *channel* is an integer constant or variable specifying the number of the channel to be used (within the range 0 to 63_{10}).

start-word is the first element of the three-word block in which the previously saved channel status is stored.

Example

```
CALL CHSAV(25, I(1))
.
. scan data in file
.
CALL CHRST (25, I(I))
.
. rescan data in file
.
```

Notes

You must restore the same channel that was saved by the call to **CHSAV**.

If the specified channel is not open, **CHRST** returns a non-fatal error message. Also, if **CHRST** detects an attempt to restore channel information that has not been saved (using **CHSAV**), **CHRST** returns a non-fatal error message.

Using **CHSAV** with the library routine **CHRST** gives the user a powerful means of returning to reprocess any record within a given disc file.

CHSAV (subroutine)

Saves the status of a MP/Fortran IV I/O channel.

Syntax

CALL CHSAV*(channel, start-word)*

where: *channel* is an integer constant or variable specifying the number of the channel to be used (within the range 0 to 63_{10}).

start-word is an element of an integer array specifying the start of a three-word block. This block is used to save the current channel status.

Example

```
    INTEGER NSAVE(6), CHNUM
    .
    .
    .
C   THE FOLLOWING CALL SAVES
    STATUS INFO IN NSAVE(4),
    NSAVE(5), AND
C   NSAVE(6):
    CALL CHSAV(CHNUM, NSAVE(4))
```

Notes

If the specified channel is not open, **CHSAV** returns a non-fatal error message.

The status on more than one channel may be saved with an appropriate two-dimensional integer array. An array declared as:

```
I(3, 25)
```

can be used to save up to 25 blocks of channel status information. You can save information about a second channel, J, with:

```
CALL CHSAV(J, I(1,J))
```

Using **CHSAV** with the library routine **CHRST** gives the user a powerful means of returning to reprocess any record within a given disc file.

CLOG (function)

Computes the single precision complex natural logarithm of a single precision complex argument.

Syntax

CLOG*(cx)*

where: *cx* is a single precision complex constant, variable, or expression.

Example

```
COMPLEX L, G
L = CLOG(G)
```

Notes

Error messages are generated on overflow or underflow.

CLOSE (subroutine)

Closes a file, and frees its associated channel.

Syntax

CALL CLOSE*(channel, error)*

where: *channel* is an integer variable or constant whose value specifies the channel number associated with the file you want to close.

error is an integer variable that returns one of the MP/Fortran IV or run-time error codes upon completion of the call.

Example

```
CALL CLOSE(14, IER)
```

Notes

This routine is preferred over **FCLOS**, which is similar.

CMPLX (function)

Constructs a single precision complex number from two single precision real numbers.

Syntax

CMPLX*(x_1, x_2)*

where: each x is a single precision real constant, variable, array element, or expression.

Example

```
COMPLEX CR
CR = CMPLX(R, D)
```

Notes

CMPLX does not generate any error messages.

COMMON (specification statement)

Allocates a data storage area that MP/Fortran IV program units can share.

Syntax

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

where: each *list* is a list of names of variables and arrays, and each *block* is the optional name of the block of COMMON storage that is to contain the list following.

Examples

Unlabeled **COMMON**:

```
COMMON ALPHA(25), X, Y, Z
```

The statement above sets up a block of unlabeled **COMMON** storage.

Labeled **COMMON**:

```
COMMON/INDEX/COUNT,TLIST//MATCH, NUM
```

The **COMMON** storage block labeled **INDEX** contains the variables **COUNT** and **TLIST**. **MATCH** and **NUM** are in unlabeled **COMMON**. If this statement and the one above appear in a program unit, only one unlabeled **COMMON** block (rather than two) will be allocated, containing, in this order:

```
ALPHA(25), X, Y, Z, MATCH, NUM.
```

Notes

Arrays may be dimensioned in a **COMMON** statement.

Each program unit that uses **COMMON** storage must have a **COMMON** statement at the head of the program unit.

An empty field between two slashes indicates unlabeled **COMMON**. No slashes are needed if the unlabeled **COMMON** list appears as the first list in the **COMMON** statement.

Although no dummy arguments may appear in a **COMMON** statement, different program units do not have to refer by the same names to variables stored in **COMMON**.

COMMON storage is positional, with correspondence between variables established by lining up the first variable or array element in each list. After this, all the others must fall in place sequentially.

```
COMMON/BLK/D,E,F          ;Program unit 1
COMMON/BLK/G,H,O          ;Program unit 2
Storage in BLK:
 D E F
 G H O
```

The size of unlabeled **COMMON** in the various program units does not have to match; but blocks of labeled **COMMON** must match in size in the different program units. (The size of a **COMMON** block can be increased with an **EQUIVALENCE** as well as a **COMMON** statement.)

```
DOUBLE PRECISION W   ;Program unit 1
        COMMON /STAR/W(10)        ;These two COMMON
                                  ;declarations will
        REAL R       ;Program unit 2   ;cause an error.
        COMMON /STAR/R(10)
Storage in STAR:

  W(1)  W(2)      ...   W(5)      ..... W(10)
R(1) R(2)  R(3) R(4)  ...  R(9)
                           R(10)
```

Two real variables take up the same storage space as one double precision variable, so the array **R** only takes up half the space in **COMMON** that the array **W** does. Since you are placing both **W** and **R** in the same labeled **COMMON** block, you must make **STAR** appear the same length in both program units. You do this by adding place-holder variables to the **COMMON** statement where **R** is declared, as:

```
COMMON /STAR/R(10),DUMMY(10)
```

You may not **DATA** initialize variables in unlabeled **COMMON**.

COMPILER DOUBLE PRECISION (specification statement)

Forces all real variables and constants to double precision, and all complex variables and constants to double precision complex.

Syntax
COMPILER DOUBLE PRECISION

Notes
COMPILER DOUBLE PRECISION must be the first statement in a program. It overrides any succeeding **REAL** or **COMPLEX** specification statements.

The compiler recognizes those single precision library functions that have double precision counterparts and generates calls to the appropriate double precision functions. However, it does not override the precision of library functions passed as arguments.

Using all single precision or all double precision variables and constants reduces object program size, because the single and double precision arithmetic packages are separate. Each package requires about 600 words of storage. Use of the **COMPILER DOUBLE PRECISION** statement ensures that only the double precision arithmetic package is loaded.

COMPILER NOSTACK (specification statement)

Causes all non-common variables and arrays to be placed in a fixed location in memory rather than on the run-time stack.

Syntax
COMPILER NOSTACK

Notes
If used, this statement must be the first statement of a program unit, or the second statement if the **COMPILER DOUBLE PRECISION** statement is given.

With the **COMPILER NOSTACK** specification, a program has the following attributes:

- Non-common variables may be initialized in **DATA** statements.
- Variables within a subprogram remain defined from one invocation of the subprogram to the next.

COMPLEX (specification statement)

Specifies that all values assigned to the variable(s) named will be of data type complex.

Syntax

COMPLEX $v_1, v_2, \ldots, v_n$

where: each v is a variable name, an array name, a dimensioned array name, a function name, or a statement function argument name.

Example

```
COMPLEX IMAG, PSYCH(50)
```

Notes

A complex datum is an ordered pair of signed or unsigned real values, separated by a comma and enclosed by parentheses, for example: (3.2,1.86).

The **COMPLEX** statement must appear to specify any variables used to store complex values, and must come before any executable statements in the program unit.

Dummy arguments may appear, and arrays may be dimensioned in the **COMPLEX** statement.

It is illegal to change the data type of a variable within a program unit.

CONJG (function)

Returns the conjugate of a complex number.

Syntax

CONJG(*x*)

where: x is a single or double precision complex constant, variable, array element, or expression.

Example

```
COMPLEX NEW, CURR
NEW = CONJG(CURR)
```

Notes

The sign of the imaginary portion of the complex number is complemented, and the modified complex number is returned.

CONJG and **DCONJG** are equivalent functions.

CONTINUE (statement)

Causes continuation of the normal execution sequence; serves as a dummy executable statement.

Syntax
CONTINUE

Example
```
    S = 0
5   DO 15 I = 1,N
    IF (B(I) .GE. 0) GO TO 15
10      S = S + C(I) * B(I)
15      CONTINUE
```

Notes
CONTINUE is most commonly used as the destination of an **IF** statement or as the range limit statement of a **DO** loop. The example above shows both uses.

CONTINUE is only a dummy: it generates no code.

COS (function)

Computes the real cosine of an argument x expressed as a single precision real number.

Syntax
COS(*x*)
where: *x* is a single precision real constant, variable, array element, or expression.

Example
```
Y = 7.5 - (COS(F))
```

Notes
For any *x* expressed in radians, cosine(*x*) = cosine(*y* + 2*n*pi), where *y* is in the range 0 >= *y* < 2pi; hence, if you evaluate the cosine of a large *x*, some of the information is wasted in specifying multiples of 2pi radians (360 degrees), which have no effect on the result. You will see this "waste" as a loss of significant digits in the answer.

CSIN (function)

Computes the single precision complex sine of a single precision complex argument.

Syntax

CSIN*(cx)*

where: *cx* is a single precision complex constant, variable, array element, or expression.

Example

```
COMPLEX NOW, NEVER
NOW = CSIN(NEVER)
```

Notes

For any x expressed in radians, sine(x) = sine(y + 2$n$pi), where y is in the range $0 <= y < 2$pi; hence, if you evaluate the sine of a large x, some of the information is wasted in specifying multiples of 2pi radians (360 degrees), which have no effect on the result. You will see this "waste" as a loss of significant digits in the answer.

CSQRT (function)

Computes the single precision complex square root of a non-negative single precision complex argument.

Syntax

CSQRT*(cx)*

where: *cx* is a non-negative complex constant, variable, array element, or expression.

Example

```
COMPLEX ABX, FD
FD = CSQRT(FD)
```

Notes

If the input argument is negative, the function generates an error message and calculates the square root of the absolute value of the argument.

DABS (function)

Computes the absolute value of a double precision real number. Returns a double precision real number.

Syntax

DABS*(e)*

where: *e* is any double precision real constant, variable, or expression.

Examples

```
DW = 6.0 D0
DX = 6.0 D0
DY = ABS(W)  ; Y = 6.0
D2 = ABS(X)  ; Z = 6.0
```

Notes

The function does not generate error messages.

DAIMAG (function)

Returns the imaginary part of a double precision complex number as a double precision real result.

Syntax

DAIMAG*(e)*

where: *e* is any double precision complex constant, variable, or expression.

Example

```
X = (1.0D0, 3.0D-1)
Y = DAIMAG(X)  ; Y = 3.0D-1
```

Notes

No error messages are generated.

DATA (data initialization statement)

Defines initial values for variables and array elements stored in labeled common.

Syntax

DATA $vlist_1/clist_1/vlist_2/clist_2/. . . vlist_n/clist_n/$

where: each *vlist* is a list of names of variables, arrays, and array elements with constant subscripts, (but no dummy arguments).

each *clist* is a list of signed or unsigned constants.

Examples

```
COMPLEX CC
DATA
X,Y,I,L,S,CC/1.0,1.0,0,.TRUE.,4HCOST,3.86,3.2/
```

The constant **4HCOST** is a Hollerith string. **4H** tells the compiler to expect four characters that should be treated as literals. Other ways to say that **COST** should be treated as a string are to enclose it in single or double quotes: **'COST'** or **"COST"**.

Use of Repeat Count:

```
DATA A,B,C,AR(1,1),AR(2,2),AR(3,3)/6*1.0/ ↑ {repeat count}
```

causes the value 1.0 to be assigned to the six variables of the variable list. A repeat count cannot be used with a string constant.

Notes

Variables besides those in labeled common may be DATA initialized if the **COMPILER NOSTACK** option is used. (See **COMPILER NOSTACK.**)

Constants are assigned to variables according to their positions in the paired lists.

You must make sure the numbers of variables and constants in your lists correspond exactly. If the constant list is longer than the variable list, the constants will be placed in succeeding storage locations as long as the constants' data types agree with the type of the last variable in the list. This will cause undefined results. If the variable list is longer than the constant list, the compiler will flag the DATA statement as an error.

If an unsubscripted name of an array appears in the variable list, the name is usually assumed to refer only to the first element of the array. However, if the array name is the last name in the variable list, all remaining constants will be assigned to sequential elements of the array. A **DATA** statement packs arrays with the first subscript varying most often, then the second, and so on (exactly as the array is packed from a **DIMENSION** statement):

```
DIMENSION I(2,2)
DATA I/0,1,2,3/
    I(1,1) first
    I(2,1) second
    I(1,2) third
    I(2,2) fourth
```

In general, you initialize arithmetic and logical variables with constants that have the same data type. The exceptions: initialize complex variables with two single precision real numbers, and initialize double precision complex variables with two double precision real numbers.

Any variable may be initialized with string data. Each ASCII character of a string constant will occupy one byte (two characters per 16-bit word.) For example, an eight character string will fill four integer or logical variables, two real variables, or one double precision variable. A string constant will initialize any number of consecutive words, depending only upon the length of the string. For this reason you must be careful to include enough variables in your list to accommodate the entire string.

DATAN (function)

Returns the double precision real arctangent of a double precision argument. The result returned will be in the range:

-pi <= **DATAN**(*e*) <= pi

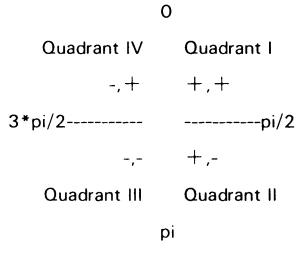
Syntax

DATAN(*e*)
where: *e* is a double precision real variable, expression, or array element.

Examples

```
DPRX = DATAN(DBLU)
ANGLE = DATAN(DP + PI)
```

If **DP** is a positive double precision real value, adding pi to it will force the function to return a value from quadrant III:



Notes

The sign of the result is the same as the sign of the input argument. The result is in radians.

DATAN2 (function)

Calculates the double precision real arctangent of the quotient of two arguments, e_2/e_1.

Syntax

DATAN2(e_1, e_2)
where: e_1 and e_2 are double precision real arguments, array elements, or expressions.

Examples

```
DY = DATAN2(DX, DEV)
DPRVAL = DT * DA/DATAN2(X(NUM), Y(NUM))
```

Notes

The sign of the result is the same as the sign of the argument quotient. The result is in radians.

You may invoke the function **DATAN2** by the name **DATN2**.

Overflow is possible as the divisor e_2 approaches zero. In the case of overflow, **DATAN2** returns + or - pi.

If $e_2 = 0$, the compiler returns an error message.

DATE (subroutine)

Gets the current date.

Syntax

CALL DATE*(date-array, error)*

where: *date-array* is the name of a three-element integer array that will be set equal to the date.

error is an integer variable which will return one of the FORTRAN or system error codes.

Example

```
CALL DATE(IAR, IER)
```

Notes

The date is expressed in the order: month, day, year; where January = 1, and year = year - 1900 (e.g., 1979 returns 79).

DBLE (function)

Expresses a single precision real argument in double precision form.

Syntax

DBLE*(x)*

where: *x* is a single precision constant, variable, array element, or expression.

Example

```
DOUBLE PRECISION YY
YY = DBLE(Y)
```

DCABS (function)

Computes the absolute value of a double precision complex number. Returns a double precision real number.

Syntax

DCABS*(arg)*

where: *arg* is any double precision complex constant, variable, or expression.

Examples

```
DOUBLE PRECISION COMPLEX DSC, DCX
.
.
.
DSC = DCABS(DCX)
```

Notes

The function may generate stack overflow messages.

DCABS is defined as a complex modulus operation: SQRT(REAL(*arg*) ** 2. + IMAG(*arg*) ** 2.)

DCCOS (function)

Computes the double precision complex cosine of a double precision complex number.

Syntax

DCCOS*(dccx)*

where: *dccx* is a double precision complex constant, variable, array element, or expression.

Example

```
DOUBLE PRECISION COMPLEX QWERTY, UIOP
QWERTY = DCCOS(UIOP)
```

Notes

The routine generates errors on underflow and overflow.

For any x expressed in radians, cosine(x) = cosine(y + 2$n$pi), where y is in the range 0 >= y < 2pi; hence, if you evaluate the cosine of a large x, some of the information is wasted in specifying multiples of 2pi radians (360 degrees), which have no effect on the result. You will see this "waste" as a loss of significant digits in the answer.

DCEXP (function)

Computes the double precision complex value of e^x.

Syntax

DCEXP*(x)*

where: *x* is any double precision complex constant, variable, array element, or expression.

Example

```
DOUBLE PRECISION COMPLEX F,G
F = DCEXP(G)
```

Notes

If either underflow or overflow occurs, an error message is typed at the console, and the routine returns zero or the greatest possible real value ($7.2 * 10^{75}$), as a result.

If x is the input argument, the routine performs the following calculation:

$$e^x = x * \log_e 2 = 2^{(I + F)}$$

where I and F are the integral and fractional portions of the power whose base is 2. The argument x of e^x must be selected so that $I <= 175_8$.

In the case of very large I values, where $I > n*2^{16}$, the routine generates an error message.

DCLOG (function)

Computes the double precision complex natural logarithm of a double precision complex argument.

Syntax

DCLOG*(dcx)*

where: *dcx* is a double precision complex constant, variable, or expression.

Example

```
DOUBLE PRECISION COMPLEX A,B
A = DCLOG(B)
```

Notes

Error messages are generated on overflow or underflow.

DCMPLX (function)

Constructs a double precision complex number from two double precision real numbers.

Syntax

DCMPLX(dx_1, dx_2)

where: each dx is a double precision real constant, variable, array element, or expression.

Example

```
DOUBLE PRECISION COMPLEX DPCX
DOUBLE PRECISION DP, CX
DPCX = DCMPLX(DP, CX)
```

Notes

DCMPLX, does not generate any error messages.

DCONJG (function)

Returns the conjugate of a complex number.

Syntax

DCONJG(x)

where: x is a single or double precision complex constant, variable, array element, or expression.

Example

```
DOUBLE PRECISION COMPLEX FOR, BACK
FOR = DCONJG(BACK)
```

Notes

The sign of the imaginary portion of the complex number is complemented, and the modified complex number is returned.

CONJG and **DCONJG** are equivalent functions.

DCOS (function)

Calculates the double precision cosine of a double precision real number.

Syntax

DCOS*(dx)*

where: *dx* is a double precision real constant, variable, array element, or expression.

Example

```
DOUBLE PRECISION OWIU, WEOX
OWIU = DCOS(WEOX)
```

Notes

For any x expressed in radians, cosine(x) = cosine(y + 2$n$pi), where y is in the range 0 >= y < 2pi; hence, if you evaluate the cosine of a large x, some of the information is wasted in specifying multiples of 2pi radians (360 degrees), which have no effect on the result. You will see this "waste" as a loss of significant digits in the answer.

DCSIN (function)

Computes the double precision complex sine of a double precision complex argument.

Syntax

DCSIN*(dcx)*

where: *dcx* is a double precision complex constant, variable, array element, or expression.

Example

```
DOUBLE PRECISION COMPLEX LEFT, RIGHT
LEFT = DCSIN(RIGHT)
```

Notes

For any x expressed in radians, sine(x) = sine(y + 2$n$pi), where y is in the range 0 <= y < 2pi; hence, if you evaluate the sine of a large x, some of the information is wasted in specifying multiples of 2pi radians (360 degrees), which have no effect on the result. You will see this "waste" as a loss of significant digits in the answer.

DCSQRT (function)

Computes the double precision complex square root of a non-negative double precision complex argument.

Syntax

DCSQRT*(dcx)*

where: *dcx* is a non-negative double precision complex constant, variable, array element, or expression.

Example

```
DOUBLE PRECISION COMPLEX SDKJH, SDLKFJ
SDLKFJ = DCSQRT(SDKJH)
```

Notes

If the input argument is negative, the function generates an error message and calculates the square root of the absolute value of the argument.

DELETE (subroutine)

Deletes a disc file.

Syntax

CALL DELETE*(pathname)*

where: *pathname* is the name, in a string constant or variable, of the file you want to delete.

Example

```
CALL DELETE("DATAFILE")
```

Notes

This routine returns no error message if the file does not exist before the **DELETE** call is executed.

DESTR (subroutine)

Aborts a task specified by its identification number.

Syntax

CALL DESTR*(id, error)*

where: *id* is an integer variable, constant, or array element specifying the identification number of the task.

error is an integer variable that returns a MP/Fortran IV or run-time error code on completion of the call.

Example

```
CALL DESTR(127, IERR)  ; Terminate task 127
```

Notes

This routine is the same as the routine **ABORT**.

DEXP (function)

Computes the double precision real value of e^x.

Syntax

DEXP*(x)*

where: x is any double precison real constant, variable, array element, or expression.

Example

```
DOUBLE PRECISION X,Y,Z
Z = DLOG(X)
Y = DEXP(Z)  ; Y equals Z
```

Notes

If either underflow or overflow occurs, an error message is typed at the console, and the routine returns zero or the greatest possible real value ($7.2 * 10^{75}$) as a result.

If x is the input argument, the routine performs the following calculation:

$$e^x = x * \log_e 2 = 2^{(I + F)}$$

where I and F are the integral and fractional portions of the power whose base is 2. The argument x of e^x must be selected so that $I <= 175_8$.

In the case of very large I values, where $I > n*2^{16}$, the routine generates an error message.

DFILW (subroutine)

Deletes a disc file.

Syntax

CALL DFILW*(pathname, error)*

where: *pathname* is the name, in a string constant or function, of the file you want to delete.

error is an integer variable which returns a MP/Fortran IV or run-time error code on completion of the call.

Example

```
CALL DFILW("DATA12", IER)
```

Notes

DFILW and **DELETE** are similar, except that **DFILW** returns a value for the *error* argument, instead of terminating the task if an error occurs.

This routine returns no error message if the file *pathname* does not exist before the DFILW call is executed.

DFLOAT (function)

Converts an integer to a double precision real number.

Syntax

DFLOAT*(i)*

where: *i* is an integer constant, variable, array element, or expression.

Example

```
DOUBLE PRECISION DX
M = 6.0E24
DX = DFLOAT(M) ; DX = 6.0D24
```

DIM (function)

Computes the positive difference of two single precision real numbers, returns a real result.

Syntax

DIM(e_1, e_2)

where: each e is a single precision real constant, variable, array element, or expression.

Example

```
HOWTALLER = DIM(TALL,SHORT)
```

Notes

If e_1 - e_2 <= 0.0, the result is zero; otherwise, the result is the difference: e_1 - e_2.

That is, the function is defined as:

e_1 - **AMIN1**(e_1, e_2)

DIMENSION (specification statement)

Gives the subscript bounds (dimensions) of arrays for allocation of storage to the arrays.

Syntax

DIMENSION $a_1(i_1)$, $a_2(i_2)$, ..., $a_n(i_n)$

where: each a is the name of an array, and each i represents the subscript bounds of an array.

The general form of a set of subscript bounds is:

$sb_1, sb_2, \ldots sb_n$

where: each *sb* is an integer constant, dummy integer variable, or a (possibly mixed) pair of these separated by a colon (:). An array may have up to 128 dimensions.

Notes

The bounds of a given array may be set only once, in a **DIMENSION, COMMON**, or data-type specification statement. These statements, if used, must appear before any executable statements in the program unit.

When a pair of values or variables separated by a colon gives the dimensions of an array, the first value or variable gives the lower bound, and the second value or variable gives the upper bound.

When a subscript bound is a single integer, a lower bound of 1 is implied. For example:

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

is identical to:

```
DIMENSION FLOWERS(1:3, 1:5, 1:2, 1:2)
```

Array dimensions are not passed to subroutines; rather, the dimensions declared within the subroutine determine the array size and structure usable within the subroutine.

"Adjustable dimensioning" involves using variables to specify array dimensions. For example, array dimensions could be passed as arguments to a subroutine as follows:

```
SUBROUTINE ABC(I, J, K, A)
DIMENSION A(I, J, K)
    .
    .
END
```

You may create an essentially boundless array within a subroutine if you dimension the array to one. You must use caution with this method, however. For example, if the following subroutine is executed, an endless loop will result:

```
SUBROUTINE PRINT (A)
DIMENSION A(1)
WRITE(10) A
RETURN
END
```

Since, practically speaking, the array **A** has no bounds, subroutine **PRINT** will start printing the contents of core, beginning where array **A** is allocated. The subroutine should have used an implied DO loop like this one to write the contents of array **A**:

```
WRITE (10) (A(I), I = 1,10)
```

Arrays are stored in ascending storage locations, with the value of the first subscript changing most rapidly, and that of the second and succeeding subscripts changing less rapidly.

For example, elements of array C(15) are stored:

```
C(1), C(2), C(3), . . ., C(15)
```

Elements of array A(10,10) are stored:

```
A(1,1), A(2,1), . . . , A(9,1), A(10,1),
A(1,2), A(2,2), A(3,2), . . ., A(10,2),
A(1,3), A(2,3), . . .A(9,10), A(10,10)
```

Elements of array B(2,3,4) are stored:

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

DINT (function)

Truncates the fractional part of a double precision real number, returns a double precision result.

Syntax

DINT*(e)*

where: *e* is a double precision real constant, variable, or expression.

Example

```
X = 87.65D0
WHOLE = DINT(X) ; WHOLE = 87.0D0
```

You can "round off" a double precision number with the value:

```
WHOLE = DINT(X + 0.5D0) ; WHOLE = 88.0D0
```

Notes

Result returned is a double precision real number with the value:

*sign of argument * largest integer <= |argument|*

DINT generates no error messages.

DIR (subroutine)

Changes the current working directory.

Syntax

CALL DIR*(directoryname, error)*

where: *directoryname* is the pathname that is to become the new working directory.

error is an integer variable which will return one of the MP/Fortran IV or run time error codes.

Examples

```
CALL DIR (''@DPX1'', IER)
CALL DIR ('USER_DIR', IER)
CALL DIR ('@DPX0:UTIL', IER)
```

DLOG (function)

Computes the double precision real natural logarithm of a double precision real positive argument.

Syntax

DLOG*(dx)*

where: *dx* is a positive double precision real constant, variable, or expression.

Example

```
DOUBLE PRECISION Q, D, G
Q = DEXP(G)
D = DLOG(Q)  ; Q equals G
```

Notes

If the argument is zero, an error message is given and the largest possible negative number, $-7.2 * 10^{75}$, is returned.

If the argument is negative, an error message is given and the logarithm of **ABS***(argument)* is computed.

DLOG10 (function)

Computes the double precision real base 10 logarithm of a double precision real positive argument *x*.

Syntax

DLOG*(e)*

where: *e* is a positive double precision real constant, variable, or expression.

Example

```
DOUBLE PRECISION X, Y, Z
X = 10**Z
Y = DLOG10(X) ; Y = Z
```

Notes

If the argument is zero, an error message is given and the largest possible negative number is returned.

If the argument is negative, an error message is given and the logarithm of |argument| is computed.

DMAX1 (function)

Selects the largest member from a set of double precision real numbers, expressing the selection as a double precison real number.

Syntax

DMAX1*(dx*$_1$*, dx*$_2$*, . . . dx*$_n$*)*

where: *dx* is a double precision real constant, variable, array element, or expression.

Example

```
DV = DMAX1(11.5D0, 11.376D0, 1D2) ; DV equals
1D2
```

Notes

DMAX1 does not generate error messages.

DMIN1 (function)

Selects the smallest member from a set of double precision real numbers, expressing the selection as a double precison real number.

Syntax

DMIN1$(dx_1, dx_2, \ldots dx_n)$

where: dx is a double precision real constant, variable, array element, or expression.

Example

```
DV = DMIN1(11.5D.0, 11.376D0, 1D2) ; DV equals
11.376D0
```

Notes

DMIN1 does not generate error messages.

DMOD (function)

Returns the double precision real remainder in the quotient of two double precision real arguments, arg_1/arg_2

Syntax

DMOD(arg_1, arg_2)

where: arg_1 and arg_2 are double precision real constants, variables, array elements, or expressions.

Example

```
DOUBLE PRECISION START, T1, T2
START = 10.0D0 * DMOD(T1, T2)
```

If **T1** equals 3.0D0, and **T2** equals 5.0D0, **START** receives the value 3.0D0.

Notes

DMOD is defined as:

$arg_1 - [arg_1/arg_2] * arg_2,$

where: $[arg_1/arg_2]$ is the truncated value of that quotient.

If the quotient causes overflow or underflow, **DMOD** generates an error message and will not return a meaningful result.

DO (statement)

Sets up a loop. (Also see: DO-Implied List)

Syntax

DO *n i* = m_1, m_2, m_3
DO *n i* = m_1, m_2

where: *n* is a statement label, indicating the range of the loop,

i is a nonsubscripted integer variable called the control variable.

m_1, m_2, and m_3 are integer constants or non-subscripted integer variable names. They are the initial value, final value, and incremental value, respectively, of *i*. If m_3 is omitted, its default value is 1; m_3 must be greater than or equal to 1.

Examples

Simple **DO** loop:

```
      DIMENSION A(100)
      .
      .
      .
      SUMSQ = 0.0
      DO 25 I = 1,100
25      SUMSQ = SUMSQ + A(I)**2
```

Nested **DO** loops:

```
      INTEGER SUM, MATRIX(10,20)
      .
      .
      .
      SUM = 0
      DO 30 I = 1,10
      DO 30 J = 1,20
      SUM = SUM + MATRIX(I,J)
30      CONTINUE
```

The range of a nested **DO** cannot extend beyond the range of an outer **DO** loop.

Notes

The parameters of a **DO** are the values the control variable may assume within the loop (E.g., m_1, m_2, and m_3.

Control cannot be transferred into the range of a **DO**. (Control *can* be transferred out of the **DO** range: this is an abnormal exit.)

The statement terminating the range of the **DO** must be an executable statement, and cannot be a **GO TO** of any form, an arithmetic **IF, RETURN, STOP, PAUSE, DO,** or a logical **IF** containing any of these statements.

The control variable cannot be redefined within the **DO** range. After a normal exit from a **DO** loop, the control variable's value is undefined.

If a **DO** loop is exited by execution of a **GO TO** or arithmetic **IF** statement, the control variable's value is defined as its value at the time of exit from the loop via the **GO TO** or **IF**.

Extending the range of a DO Loop

The range of a **DO** loop can be extended to include additional statements or program units if:

1. A statement exists in an innermost, completely nested **DO** loop that transfers control out of that loop, and
2. A statement that might logically be executed as part of the extended range exists to transfer control back into the innermost, completely nested loop.

```
      INTEGER SUM, TOTAL,
      MATRIX(10,20)
      .
      .
      .
      SUM = 0 DO 30 I = 1,10    Completely nested
      DO 25 J = 1,20 SUM =      Innermost DO
      SUM + MATRIX(I,J) IF
      (SUM .GT. TOTAL) GO TO
      50
25    CONTINUE
30    CONTINUE
      .
      .
      .
50
      .                         Extended range of DO
      .                         containing at least one
      .                         statement transferring
      GO TO 25                  control back into the
      .                         innermost DO.
      .
      .
```

DO-Implied List

Controls repetitive action of a **READ** or **WRITE** statement on a specified list of I/O elements, usually arrays or parts of arrays.

Syntax

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

where: *list* is a list of variables to **READ** or **WRITE**.

i is the control variable, which must be an unsubscripted integer variable.

m_1 is the initial value of the control.

m_2 is the final value of the control.

m_3 is the increment of the control variable. If m_3 is omitted, the increment is +1.

m_1, m_2, and m_3 must be integer constants or unsubscripted integer variables.

Examples

```
READ (11,20) A, B, (C(I), I = 1, 3)
```

is equivalent to:

```
READ (11,20) A, B, C(1), C(2), C(3)
```

```
WRITE (12,25) (A,B,C,D, I = 1,2)
```

is equivalent to:

```
WRITE (12,25) A,B,C,D,A,B,C,D
```

```
READ (11,30) (C(I,I), I = 1,4)
```

is equivalent to:

```
READ (11,30) C(1,1), C(2,2), C(3,3), C(4,4)
```

DO-implied lists may be nested to any depth. For example:

```
READ (11,50) ((A(I,J), J = 1,4), I = 1,9,2)
```

Notes

As an extension to ANSI Fortran, MP/Fortran IV saves the current value of the control variable upon entering an implied **DO** loop, and restores the value on completing the loop.

DOUBLE PRECISION (specification statement)

Specifies that all values assigned to the variable(s) named will be of the double precision data type.

Syntax

DOUBLE PRECISION $v_1, v_2, \ldots, v_n$

where: each *v* is a variable name, an array name, a dimensioned array name, a function name, or a statement function argument name.

Example

```
DOUBLE PRECISION SCIENCE(10), METHOD
```

Notes

A double precision datum is a signed or unsigned floating point value.

The **DOUBLE PRECISION** specification statement must appear if any double precision variables are needed. This statement must come before any executable statements in the program unit.

Dummy arguments may appear, and arrays may be dimensioned in the **DOUBLE PRECISION** statement.

It is illegal to change the data type of a variable within a program unit.

DOUBLE PRECISION COMPLEX (specification statement)

Specifies that all values assigned to the variable(s) named will be of data type double precision complex.

Syntax

DOUBLE PRECISION COMPLEX $v_1, v_2, \ldots, v_n$

where: each v is a variable name, an array name, a dimensioned array name, a function name, or a statement function argument name.

Example

```
DOUBLE PRECISION COMPLEX DPLEAS, DFUN(5,8)
```

Notes

A double precision complex datum is an ordered pair of signed or unsigned double precision values, separated by a comma and enclosed in parentheses.

The **DOUBLE PRECISION COMPLEX** statement must appear to specify any variables used for storage of double precision values. This statement must come before any executable statements in the program unit.

Dummy arguments may appear, and arrays may be dimensioned in the **DOUBLE PRECISION COMPLEX** statement.

It is illegal to change the data type of a variable within a program unit.

DREAL (function)

Returns the double precision real part of a double precision complex number.

Syntax

REAL(*dcx*)

where: *dcx* is a double precision complex constant, variable, array element, or expression.

Example

```
DOUBLE PRECISION DEXNORM
DOUBLE PRECISION COMPLEX DM
DM = (8.90D1, .521D0
DEXNORM = DREAL(DM) ; DEXNORM equals 8.90D1
```

Notes

DREAL does not generate error messages.

DSIGN (function)

Transfers the sign of one double precision real number, dx_2 to another double precision real number, dx_1.

Syntax

DSIGN(dx_1, dx_2)

where: dx_1 and dx_2 are both double precision real constants, variables, array elements, or expressions.

Example

```
DOUBLE PRECISION DEND, DEAD, END
END = 1.0D0
DEAD = 8.754D2
DEND = DSIGN(DEAD, END) ; DEND equals -8.754D2
```

Notes

DSIGN does not generate error messages.

DSIGN performs the operation:
sign of dx_2 * dx_1.

DSIN (function)

Computes the double precision real sine of a double precision real argument.

Syntax

DSIN(dx)

where: dx is a double precision real constant, variable, array element, or expression.

Example

```
DOUBLE PRECISION DV,DC
DV = DSIN(DC)
```

Notes

For any x expressed in radians, sine(x) = sine(y + 2$n$pi), where y is in the range 0 <= y < 2pi; hence, if you evaluate the sine of a large x, some of the information is wasted in specifying multiples of 2pi radians (360 degrees), which have no effect on the result. You will see this "waste" as a loss of significant digits in the answer.

DSQRT (function)

Computes the double precision square root of a non-negative double precision real argument.

Syntax

DSQRT*(dx)*

where: *dx* is a non-negative double precision real constant, variable, array element, or expression.

Example

```
DOUBLE PRECISION DEF, INIT, BASE
INIT = BASE**2
DEF = DSQRT(INIT) ; DEF = BASE
```

Notes

If the input argument is negative, the function generates an error message and calculates the square root of the absolute value of the argument.

DTAN (function)

Computes the double precision real tangent of a double precision real argument.

Syntax

DTAN*(dx)*

where: *dx* is a double precision real constant, variable, array element, or expression.

Example

```
DOUBLE PRECISION ANDR, C3PO
ANDR = DTAN(C3PO)
```

Notes

For any x expressed in radians, tangent(x) = tangent(y + $n$pi), where y is in the range $0 >= y < \text{pi}$; hence, if you evaluate the tangent of a large x, some of the information is wasted in specifying multiples of pi radians (180 degrees), which have no effect on the result. You will see this "waste" as a loss of significant digits in the answer.

DTANH (function)

Computes the double precision real hyperbolic tangent of a double precision real argument.

Syntax

DTANH*(x)*

where: *x* is a double precision real constant, variable, array element, or expression.

Example

```
DOUBLE PRECISION THUNDER, THOR
THUNDER = DTANH(THOR)
```

Notes

The routine generates error messages on underflow and overflow.

EBACK (subroutine)

Returns the last-swapped program to memory; if it was the CLI, the error message is typed at the console.

Syntax

CALL EBACK*(error)*

where: *error* is an integer variable which will return one of the MP/Fortran IV or run time error codes on completion of the call.

Notes

EBACK does an assembly language call to **?RETURN**, using the error code passed in *error*. For more details on program management, see the *MP/OS Assembly Language Programmer's Reference.*

BACK, FBACK, EXIT and **STOP** provide other ways of terminating the program.

EBACK does not pass **COMMON**.

ENDFILE (statement)

Closes the file associated with the specified channel number.

Syntax

ENDFILE *channel*

where: *channel* is the channel number ($0\text{-}63_{10}$) associated with the file you want to close.

Examples

```
OPEN (25, "XREF", 2, IER, 128)
.
.
.
ENDFILE 25
END
```

EQUIVALENCE (specification statement)

Specifies that all variables named within a given list in this statement share the same storage area.

Syntax

EQUIVALENCE *(list$_1$),(list$_2$),. . .,(list$_n$*

where: each list is a list of names of variables, arrays, and array elements having constant subscripts.

Example

```
DIMENSION B(6), D(4)
COMMON D
EQUIVALENCE (B(1), D(2))
```

Storage in blank common area:

```
D(1)   D(2)   D(3)   D(4)
       B(1)   B(2)   B(3)   B(4)   B(5)   B(6)
                     } Extended Common {
```

Notes

If used, **EQUIVALENCE** statements must appear before any executable statements in the program unit.

Dummy argument names of arrays cannot appear in **EQUIVALENCE** statements.

An array name with no subscript causes equivalencing to begin with the first element of that array.

Array elements in **EQUIVALENCE** statements may be referenced by complete subscripts or by a single subscript equal to the element's position in the storage sequence. For example, array X(3,2) is stored:

X(1,1), X(2,1), X(3,1), X(1,2), X(2,2), X(3,2)

The element X(2,2) is fifth in this sequence: therefore it can also be referenced in an **EQUIVALENCE** statement as X(5).

Because common storage is positional, when an element of one array is equivalenced with an element of another array, that determines storage correspondence for all elements of the arrays.

Common may only be extended beyond the last assignment of storage made in a **COMMON** statement; no core storage is left empty to provide

for **EQUIVALENCE** extensions in the other direction.

A variable or array not declared in common may appear in only one equivalence list.

Equivalencing storage should not be used to equate entities mathematically. For example, if a **REAL** variable is equivalenced with a **DOUBLE PRECISION** variable, the **REAL** variable will share storage only with the first storage unit of the two-unit **DOUBLE PRECISION** variable.

EXIT (subroutine)

Returns to the next lower program level.

Syntax
CALL EXIT

Notes
BACK, FBACK and **EXIT** are identical. They all perform an assembly language call to **?RETURN**, with all arguments equal to zero. See the *MP/OS Assembly Language Programmer's Reference* for details of program management.

EBACK and **STOP** provide other ways of terminating the program.

EXP (function)

Computes the real value of e^x.

Syntax

EXP(*x*)

where: *x* is any single precison real constant, variable, array element, or expression.

Example

```
R = LOG(RB)
RV = EXP(R) ; RV equals RB
```

Notes

If either underflow or overflow occurs, an error message is typed at the console, and zero, or the greatest possible real value, replaces *x* on the stack.

If *x* is the input argument, the routine performs the following calculation:

$$e^x = x * \log_e 2 = 2^{(I + F)}$$

where *I* and *F* are the integral and fractional portions of the power whose base is 2. The argument **x** of e^x must be selected so that $I <= 175_8$.

In the case of very large *I* values, where $I > n*2^{16}$, the routine generates an error message.

EXTERNAL (specification statement)

Specifies subprograms as external to the program unit in which the specification is made.

Syntax

EXTERNAL $s_1, s_2, \ldots, s_n$

where: each *s* is the name of a function subprogram or subroutine subprogram.

Example

```
        REAL ROOT
    EXTERNAL ROOT   ;Subroutine MULT is called with
           .        ;REAL function ROOT as the last
           .        ;argument.
           .
CALL MULT(A, B, ROOT)
SUBROUTINE MULT(Q, R,
                S)
           .
           .
           .
  Q = S(Q,R)   ;This generates a call to the
               ;function passed via dummy
               ;argument S.
```

Notes

In order for the the compiler to recognize names of functions, subroutines, and tasks that appear in the program unit as arguments to be passed to another program unit, their names must be specified as **EXTERNAL**. The **EXTERNAL** statement causes the argument to be recognized as a subprogram, rather than as an array or a variable. An address for the subprogram argument can then be passed to the called program unit.

The **EXTERNAL** statement must appear before any executable statements in the program unit.

The data type of an external function subprogram may appear in a data type specification statement in the calling program.

Each overlay or task name must be declared **EXTERNAL** in any program unit in which it is referenced. (See **OVERLAY** and **TASK** in this dictionary.)

FBACK (subroutine)

Returns to the next lower program level.

Syntax
CALL FBACK

Notes
BACK, FBACK and **EXIT** are indentical. They all perform an assembly language call to **?RETURN**, with all arguments equal to zero. See the *MP/OS Assembly Language Programmer's Reference* for details of program management.

EBACK and **STOP** provide other ways of terminating the program.

FBACK does not pass **COMMON**.

FCHAN (subroutine)

Causes the current program's memory image to be overwritten by another program from disc.

Syntax
CALL FCHAN*(filename)*
where: *filename* is the name of the file to be executed next. Its execution level will be the same as that of the calling program.

Example
```
CALL FCHAN(''ABC'')
```

Notes
CHAIN and **FCHAN** are the same, except that **FCHAN** does not provide an error return location.

There is no limit to the number of chains you can perform. You can use program chaining to subdivide a program so large that it would exceed the limits of memory if it were to reside in memory in its entirety. Each chained-to program must contain a complete program.

Refer to the *MP/OS Assembly Language Reference Manual* for more information on chaining.

This routine does not pass **COMMON** under MP/OS.

FCLOS (subroutine)

Closes a file, and frees its associated channel.

Syntax

CALL FCLOS*(channel)*

where: *channel* is an integer variable or constant whose value specifies the channel number associated with the file you want to close.

Example

```
CALL FCLOS(10)
```

Notes

The routine **CLOSE** is preferred over **FCLOS**, since **CLOSE** returns error codes.

FDELY (subroutine)

Suspends a task for a specified amount of time.

Syntax

CALL FDELY*(milliseconds)*

where: *milliseconds* is an integer constant, variable, array element, or expression giving the number of milliseconds for which the task will be suspended.

Example

```
CALL FDELY(50)
```

FGTIM (subroutine)

Gets the time of day.

Syntax

CALL FGTIM*(hour, minute, second)*

where: *hour, minute,* and *second* are integer variables which will return the correct hour, minute and second.

Example

```
CALL FGTIM(IHR, IMIN, ISEC)
```

Notes

The real-time clock is a 24-hour clock.

The subroutine **TIME** also returns the time of day.

FINRV (subroutine)

Removes a previously added user interrupt device handler from the system interrupt vector table.

Syntax

CALL FINRV*(devicecode)*

where: *devicecode* is an integer variable or constant which must be the device code of a previously identified (via **FINTD**) user interrupt device.

Example

```
CALL FINRV(23)
```

Notes

If you attempt to remove a **SYSGEN**ed device, or the *devicecode* argument is not within the legal range of user interrupt devices (0 to 63) a fatal run-time error occurs, and execution is terminated.

FINRV performs the system call **?IRMV**.

FINTD (subroutine)

Specifies a routine which is capable of handling interrupt requests from a user device.

Syntax

CALL FINTD*(devicecode, dct)*

where: *devicecode* is an integer variable or constant in the range 0 to 63, which is the code of the user device.

dct is the name of a two or more word device control table, which may be a dimensioned array or an externally defined variable. The *dct* entries are defined in the Part 3, *MP/OS System Assembly Language Programmer's Reference* (DGC No. 093-400001).

Example

```
CALL FINTD(62, IDDCT)
```

where **IDDCT** is defined to the program as either **EXTERNAL IDDCT** or **DIMENSION IDDCT(2)**.

Note that if **IDDCT** is an array, it must always be in labeled or unlabeled **COMMON**, or in program module with **COMPILER NOSTACK**, so that it is always accessible in the case of an interrupt.

Notes

Those devices that were not identified to the system at **SYSGEN** time must be made known to the system by the **FINTD** routine. **FINTD** causes an entry for the specified device code to be placed in the system interrupt vector table. (See the *MP/OS System Assembly Language Programmer's Reference* (DGC No. 093-400001).

Interrupt requests from special (non-**SYSGEN**ed) devices do not, for the most part, change the status of tasks in a MP/Fortran IV multitask environment. Instead, such interrupts freeze the environment until servicing of the interrupt is completed, and the multitask environment is unfrozen. Likewise, all other tasks will resume their former states when the environment is unfrozen, unless the user transmits a message to one of them by means of the assembly language transmit command, **?IUNPEND**.

FLOAT (function)

Converts an integer to a single precision real number.

Syntax

FLOAT*(i)*

where: *i* is an integer constant, variable, array element, or expression.

Example

```
M = 50
X = FLOAT(M)  ; X equals 50.0
```

FOPEN (subroutine)

Opens a device or disc file by assigning it a specified channel number.

Syntax

CALL FOPEN*(channel, pathname [, "B"] [, recordbytes])*

where: *channel* is an integer constant or variable with a value between 0 and 63_{10}.

pathname is a string constant or variable

"B" is ignored by the MP/Fortran IV compiler. You may use the argument for program portability among RDOS, AOS, and MP/OS.

recordbytes, if used, implies that the file is made up of records. *Recordbytes* is an integer constant or variable that specifies the length in bytes of a record. Only files made up of records may be accessed randomly.

Examples

```
CALL FOPEN(ICH, ''RFILE'', ''B'', 200)
CALL FOPEN(3, 'DATAFILE', 40)
CALL FOPEN(4, '$PTR', 'B')
```

Notes

If you attempt to open a nonexistent file, it will be created as a randomly organized file.

Only one channel at a time may be open to a character device such as the console or the lineprinter.

OPEN is preferred to **FOPEN** for opening files, since **OPEN** returns error codes.

FORMAT (statement)

Describes how data to be read are organized, or describes how data should be written.

Syntax

statement label **FORMAT***(specification)*

where: *statement label* is the label by which a **READ** or **WRITE** statement references the **FORMAT** statement.

specification is the list of carriage control characters, field descriptors, field separators, Hollerith strings, repetition constants, etc., that together define the organization of the data for input or output.

Notes

Chapter 5 covers the FORMAT statement in more detail.

FSEEK (subroutine)

Positions a random access file to a given record for reading or writing.

Syntax

CALL FSEEK *(channel, recordnumber)*

where: *channel* is an integer constant or variable specifying the assigned channel number of the file.

recordnumber is an integer constant or variable. It contains the number of the next record to be read or written.

Example

```
    CALL OPEN(21, "TXFL", IAR, IER, 4)
    CALL FSEEK (21, INUM)
    READ (21, 50) A(INUM)
50      FORMAT(E14.6)
```

Notes

The file must always be opened (through a call to **OPEN**) before **FSEEK** can be executed.

Note that MP/Fortran IV numbers file records starting at 0. If you are working with an array whose lower bound is not zero (see **DIMENSION**), the record number and the array element subscript will not be the same. In the example above, if **A** is dimensioned **A**(25), then the **READ** statement should probably be:

```
READ(21, 50) A(INUM + 1)
```

FSTIM (subroutine)

Sets the real time clock.

Syntax

CALL FSTIM *(hour, minute, second)*

where: *hour* is an integer variable or constant in the range 0 to 23.

minute is an integer variable or constant in the range 0 to 59.

second is an integer variable or constant in the range 0 to 59.

Example

```
CALL FSTIM(7, 25, 11)
```

Notes

If an attempt is made to set a time outside the specified legal range, a run-time error occurs.

The routine **STIME** also sets the time of day.

FSWAP (subroutine)

Saves the current program's memory image on disc, and loads another program into memory from disc.

Syntax
CALL FSWAP *(filename)*
where: *filename* is the name of the file to be executed next.

Example
```
CALL FSWAP('A2')
```

Notes
FSWAP is the same as **SWAP**, except that **FSWAP** does not provide an error return.

Program swapping allows memory images of programs to be saved and called for execution more than once during a program's execution. Each time it is called, the program will resume execution from the point of suspension. If the execution level of the calling program is *n*, *filename* executes at level $n + 1$.

This routine does not pass **COMMON** under MP/Fortran IV.

If an attempt is made to nest swaps to a level deeper than eight, an error will result.

Each program swapped to must contain a complete program.

For more information on program management, refer to the *MP/OS Assembly Language Programmer's Reference.*

FTASK (subroutine)

Activates a task.

Syntax
CALL FTASK*(taskname,***$***error-return, priority-number [, stack_base])*

where: *taskname* is the name of the task you want to activate. *Taskname* must first be defined in an **EXTERNAL** statement.

error-return is the line number of a statement in the calling program to which control returns if the task cannot be activated. This number must be preceded by the symbol **$.**

priority-number is an integer constant in the range 0 - 255_{10}, specifying the priority assigned to the new task.

stack_base is an optional parameter. Its use indicates that the task you want to activate is written in assembly language. It must be set to the address of the base of the stack. Conversely, if *stack_base* does not appear in the **FTASK** call, or it has a value of 0, the task *taskname* must be written in MP/Fortran IV.

Example
```
    EXTERNAL PROG, PRSTK
CALL FTASK (PROG, $14, 3, PRSTK)
.
.
.
14  STOP ERROR ON ACTIVATING PROGRAM
```

Notes
All tasks except the main MP/Fortran IV program unit are activated by executing a call either to **FTASK** or **ITASK**.

The task *taskname* must be declared EXTERNAL by the calling program.

If you are using an assembly language task, put a label on a pointer to the base of the stack and define the label as an entry point. Then, in the calling MP/Fortran IV program, define the label **EXTERNAL**, and use the label as the actual parameter *stack base*:

```
   .ENT PRSTK
   .NREL 1
PRSTK:  .PRST
   .NREL
.PRSTK:  .BLK ?STKMIN + 10
```

Initially, MP/Fortran IV assumes that the stack at *stack_base* is greater than or equal to the MP/OS minimum stack size. Once the program is executing, you may then change the size of the stack.

FUNCTION (statement)

Defines a function subprogram.

Syntax

[type] **FUNCTION** *name* (a_1, a_2, . . ., a_n)

where: *type*, if specified, is **INTEGER, REAL, COMPLEX, DOUBLE PRECISION, LOGICAL,** or **DOUBLE PRECISION COMPLEX.**

name is the name of the function subprogram. Within the first five characters, the name must be unique to the program.

each *a* is a dummy (or "formal") argument. It may be a variable name, array name, or an external subprogram name (function or subroutine). The dummy argument is replaced by an actual argument when the function subprogram is referenced. The argument list may not be blank.

Example

```
   FUNCTION SWITCH (X)
   IF (X .LE. 0.) GO TO 5
   IF (X .LE. 1.) GO TO 10
20    SWITCH = 1.
   RETURN
10    SWITCH = X
   RETURN
5  RETURN
   END
```

Notes

You reference a function subprogram in the same way as any function: you specify in an expression the name of the function and its actual arguments.

A function subprogram is external (separately compiled). An external function name that is used as an actual argument in the referencing program unit must appear in an **EXTERNAL** statement in the referencing program unit.

The function returns a value that is of the data type specified in the **FUNCTION** statement. If no data type is given, the function returns an **INTEGER** or **REAL** value, depending on the beginning letter of the function name (IJKLMN convention).

Function subprograms can execute abnormal returns. (See **RETURN**.)

The name of the function subprogram must appear on the left-hand side of an assignment statement at least once in the function subprogram. Except for the **FUNCTION** statement itself, the name of the function subprogram cannot appear in any non-executable statement in the function subprogram.

We do not recommend that you directly reference the same variable as two different parameters of a function. For example:

```
FUNCT(A,A)
```

If you want to pass the same variable twice, enclose its name in parentheses:

```
FUNCT((A), (A))
```

These parentheses tell the compiler to copy the argument, and pass the address of the copy. In this case the function will not affect the value of the variable in the calling program. For example:

```
FUNCT(X, (Y))
```

means that the routine may return a new value for **X**, but cannot change the value of **Y** in the calling program.

Dummy argument names cannot appear in **DATA, COMMON,** or **EQUIVALENCE** statements in the function subprogram.

The function subprogram cannot contain statements that define other program units, e.g., it cannot contain another **FUNCTION, TASK, BLOCK DATA** or **SUBROUTINE** statement.

When a program unit references a function subprogram, actual argument names replace dummy argument names of given structures, as shown below:

Dummy Argument	Actual Argument
Variable name	Variable name, array element name, or any expression*
Array name	Array name, or array element name**
Name that can be used as a function call***	External function name
Name that can be used as a subroutine name in a CAll statement***	External subroutine name

**When the actual argument is an expression, its value is passed.*

***When an array name is passed:*
dummy length <= actual array length.
When an array element name is passed:
dummy length <= actual array length + 1 - the actual array's subscript.

****A dummy argument that stands for a function or subroutine name cannot be defined or redefined in the function subprogram.*

See also **STATEMENT** function in this dictionary.

Statement (function)

Defines an internal function.

Syntax

$f(a_1, a_2, \ldots, a_n) = e$

where: f is the name given by the programmer to the function. Within a program unit, statement function names must be uniquely distinguishable by their first five characters.

each a is a dummy (or formal) argument name.

e is an expression.

Example

The **STATEMENT** function is:

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

This statement function might be referenced by:

```
VAL = ROOT(D(6), 122.6, ABS(X-Y)) + Z ** 3
```

In the example, D(6) replaces A, 122.6 replaces B, the absolute value of A-Y replaces C, and the expression:

```
(-122.6 + SQRT(122.6 ** 2 - 4. * D(6) * ABS(X-Y))) /
2. * D(6))
```

is evaluated and its value is returned to the assignment statement. Z**3 is added to the returned value, and the total is assigned to location VAL.

Notes

The expression on the right-hand side of the **STATEMENT** function is evaluated, and its result is assigned to the function name on the left-hand side.

Function names and dummy arguments can be explicitly data-typed through data type specification statements; otherwise they are implicitly typed according to the first letters of their names (IJKLMN convention).

Besides the dummy arguments, the expression e may contain:

1. Constants of any type,
2. Variables stored in a **COMMON** area,
3. References to previously defined functions, to MP/Fortran IV library functions, and to external functions.

To use a **STATEMENT** function, the programmer places a reference to the function in an expression. The reference contains the function name and actual arguments to replace the dummy arguments. The actual arguments are passed to the **STATEMENT** function, e is evaluated, and its value is returned to the reference point.

The actual arguments in a **STATEMENT** function must agree in order, number and type with the corresponding dummy arguments. Actual arguments in a reference may be any expression of the same type as the corresponding dummy argument.

See also **FUNCTION**.

GDIR (subroutine)

Returns the name of the default directory or device.

Syntax

CALL GDIR*(array, error)*

where: *array* is the name of the array that stores the name of the current default directory or device.

error is an integer variable which returns one of the MP/Fortran IV or run-time error codes on completion of the subroutine call.

Example

```
CALL GDIR(IAR, IER)
```

GO TO (assigned; statement)

Causes control to transfer to the statement whose label was last assigned to v in an **ASSIGN** statement (see **ASSIGN**.)

Syntax

GO TO $v[, (n_1, n_2, \ldots, n_m)]$

where: v is a non-subscripted integer variable name appearing in a previously executed **ASSIGN** statement. Note that v does not evaluate to an integer, but to a statement label.

$n_1, n_2, \ldots, n_m$ are optional statement labels. (These are actually ignored by the compiler but existing programs using this form of the **GO TO** will not be rejected.)

Examples

```
    ASSIGN 100 TO SORTX
    .
    .
    .
    GO TO SORTX
    .
    .
    .
C   SORTX STARTS HERE
100   DO 200 I = 1, N
```

Notes

If you use the optional statement label list, the compiler will check to see that each statement label is defined somewhere in the program, and will generate an appropriate error message if it is not. Otherwise, the statement label list is ignored.

GO TO (Computed; Statement)

Control transfers to the statement pointed to by the arguments of the **GO TO.**

SYNTAX

GO TO$(n_1, n_2, \ldots, n_m)[,]v$
where:$n_1, n_2, \ldots, n_m$ *are statement labels*

v is a non-subscripted integer variable name.

EXAMPLE

```
GO TO(10, 100, 40, 25, 9), K
```

K must evaluate to 1, 2, 3, 4, or 5. If K is equal to 3, control transfers to the statement labeled 40.

NOTES

The computed GO TO transfers control to the statement n_i, where i is the value of v.

If $v>m$ or $v<1$, the GO TO statement does not execute, and a fatal run time error results.

Unconditional GO TO (statement)

Transfers control to the statement specified by the programmer.

Syntax

GO TO n
where: n is a statement label.

Example

```
GO TO 25   .   .   . 25 CONTINUE
```

Notes

The **GO TO** statement can also be written as **GOTO**. The compiler will recognize it either way.

IABS (function)

Computes the absolute value of an integer argument. Returns an integer result.

Syntax

IABS*(e)*

where: *e* is an integer constant, variable, or expression.

Example

```
I = -6
J = 6
K = ABS(I) ; K = 6
L = ABS(J) ; L = 6
```

Notes

No error messages are generated.

IAND (function)

Performs a bit-by-bit AND (logical conjunction) on two integer arguments.

Syntax

IAND*(i_1, i_2)*

where: i_1 and i_2 are integer constants, variables, array elements, or expressions.

Example

```
J = IAND(J, 15)
```

This clears all but rightmost 4 bits

ICLR (subroutine)

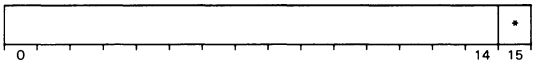
Sets a single bit in a word to zero.

Syntax

CALL ICLR*(word, position)*

where: *word* is an integer variable or array element.

position is an integer constant, variable, array element, or expression whose value specifies the position in *word* of the bit you are setting to zero:



**0* = *least significant bit*
15 = most significant bit

Example

```
CALL ICLR (IX, 10)
```

Bit 10 of IX will be cleared.

Notes

Note that the bits in a word are numbered in reverse order to DGC assembler conventions.

BCLR is a subroutine identical to **ICLR**.

IDIM (function)

Computes the positive difference of two integer numbers, returns an integer result.

Syntax

IDIM*(e_1, e_2)*

where: each *e* is an integer constant, variable, array element, or expression.

Example

```
IDIFF = IDIM(IFIRST,ISEC)
```

Notes

If e_1 - e_2 <= 0, the result is zero; otherwise, the result is the difference: e_1 - e_2.

That is, **IDIM** is defined as:

e_1 - MIN0(e_1, e_2)

IDIM does not generate error messages.

IDINT (function)

Converts a double precision real argument to an integer by truncation.

Syntax

IDINT*(e)*

where: *e* is a double precision real constant, variable, or expression.

Example

```
D = 6.01D2
I = IDINT(D)  ; I equals 601
```

Notes

An error message is generated if the truncated argument is greater than 2**15 - 1 or less than -(2**15 - 1).

*Result = Sign of argument * largest integer <=* **ABS***(argument).*

IEOR (function)

Performs a bit-by-bit logical exclusive OR on two integer arguments.

Syntax

IEOR*i_1, i_2)*

where: i_1 and i_2 are integer constants, variables, array elements, or expressions.

Example

```
N = IEOR(N, 1)
```

This flips the rightmost (least significant) bit.

IF (arithmetic; statement)

Transfers control to a specified statement, depending on the value of an expression relative to zero.

Syntax

IF *(e)* n_1, n_2, n_3

where: *e* is an integer, real, or double precision expression.

n_1, n_2, and n_3 are statement labels.

Examples

```
IF (A(J,K)-B) 10, 4, 30
IF (Q*R) 5, 5, 2
```

Notes

The expression is evaluated, and if:

- $e < 0$, control transfers to statement label n_1,
- $e = 0$, control transfers to statement label n_2,
- $e > 0$, control transfers to statement label n_3.

If a given statement label is not defined in the program, its use in an **IF** statement will be flagged as an error.

IFIX (function)

Truncates the fractional part of a single precision real number and expresses the result as an integer.

Syntax

IFIX*(x)*

where: *x* is a single precision real constant, variable, array element, or expression.

Example

```
INDEX = IFIX(XMEAN)
```

IF (logical; statement)

Controls program flow, according to the logical value of an expression.

Syntax

IF *(le) s*

where: *le* is a logical expression.

s is any executable statement (assignment statement, control statement, or I/O statement) except a **DO**.

Examples

```
IF (A .AND. B) F=SIN(R)
IF (I .GT. 0) GO TO 25
```

Notes

The logical expression is evaluated. If the expression is true, statement *s* is executed. Control then passes to the statement following the logical **IF** unless statement *s* transfers control.

If the expression is false, statement *s* is bypassed and control passes to the next statement.

INT (function)

Converts a real argument to the nearest integer by truncation.

Syntax

INT*(real)*

where: *real* is a single precision constant, variable, or expression.

Example

```
J = INT(26.7685)
```

J will have the integer value 26 after execution.

Notes

An error message is generated if the truncated real is greater than 2**15 - 1 or less than -(2**15 - 1).

*Result = Sign of argument * largest integer <=* **ABS***(argument).*

INTEGER (specification statement)

Specifies that all values assigned to the variable(s) named will be of data type **INTEGER**.

Syntax

INTEGER $v_1, v_2, \ldots, v_n$

where: each v is a variable name, an array name, a dimensioned array name, a function name, or a statement function argument name.

Example

```
INTEGER GOOD, BETTER, BEST(10)
```

Notes

An integer value is a signed or unsigned whole number without a decimal point. It is stored in one word (16 bits), and can have a value between -32,767 and 32,767.

The **INTEGER** specification statement may be used to override implicit data typing. The **INTEGER** statement, if used, must appear before any executable statements in the program unit.

Dummy arguments may appear, and, arrays may be dimensioned in the **INTEGER** statement.

The data type of a variable may not be changed within a program unit.

IOR (function)

Performs a bit-by-bit inclusive OR on two integer arguments.

Syntax

IOR(i_1, i_2)

where: i_1 and i_2 are integer constants, variables, array elements, or expressions.

Example

```
J = IOR(J, 15)
```

This sets rightmost four bits to 1, and leaves leftmost twelve bits as they were.

ISET (subroutine)

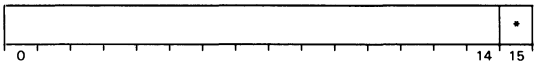
Sets a single bit in a word to one.

Syntax

CALL ISET*(word, position)*

where: *word* is an integer variable or array element.

position is an integer constant, variable, array element, or expression whose value specifies the position in *word* of the bit you are setting to one:



**0 = least significant bit*
15 = most significant bit

Example

```
CALL ISET (K, 0) Bit 0 of K will be set to one.
```

Notes

Note that the bits in a word are numbered in reverse order to the DGC assembler convention.

BSET is a subroutine identical to **ISET**.

ISHFT (function)

Shifts a word a number of bits left or right.

Syntax

ISHFT*(word, bits)*

where: *word* is an integer variable, constant, array element, or expression to be shifted.

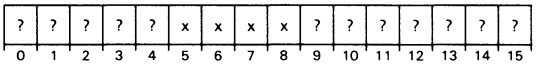
bits is an integer constant, variable, array element, or expression whose value specifies the number of bit positions the word will be shifted as follows:
A negative value represents a right shift.
A zero value represents no shift.
A positive value represents a left shift.

Example

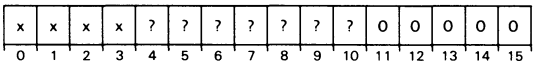
```
100    K = ISHFT (J, -5)
.
.
.
.
120        N = ISHFT(ISHFT(J,5), -12)
```

Line 100 stores into K the value of J shifted right by five bit positions.

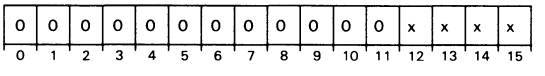
Value of J after line 100 is executed:



Word after first shift performed by ISHFT function in line 120:



Word after second shift performed by ISHFT in line 120 (this is the value of N; the value of J has not changed):



Notes

The **ISHFT** routine is often used to extract or set bit fields within individual words, as shown above.

ISIGN (function)

Transfers the sign of one integer, i_2, to another integer, i_1.

Syntax

ISIGN(i_1, i_2)

where: i_1 and i_2 are both integer constants, variables, array elements, or expressions.

Example

```
I = -6
J = ISIGN(20, I)  ; J equals -20
```

Notes

ISIGN does not generate error messages.

ISIGN performs the operation: sign of i_2 * i_1.

ITASK (subroutine)

Activates a task, assigns it an identification number.

Syntax

CALL ITASK*(taskname, identification, priority-number, error [, stack_base])*

where: *taskname* is the name of the task you want to activate.

identification is the task identification number. This is either an integer constant or variable in the range 0 - 255_{10}; zero is the default value of the identifier.

priority-number is an integer constant in the range 0 - 255_{10}, specifying the priority assigned to the newly activated task. (A priority of 0 indicates that the task will have the same priority as the calling program.)

error is an integer variable that returns one of the MP/Fortran IV or run-time error codes on completion of the call.

stack_base is an optional parameter. Its use indicates that the task you want to activate is written in assembly language. It must be set to the address of the base of the stack. Conversely, if *stack_base* does not appear in the **ITASK** call, or if it has a value of 0, the task *taskname* must be written in MP/Fortran IV.

Example

```
.
.
.
EXTERNAL OUT, OUTSTK
CALL ITASK (OUT, 10, 127, IER, OUTSTK) .
.
.
```

Notes

You activate all tasks except the main MP/Fortran IV program unit by executing a call either to **FTASK** or **ITASK**.

If you are using an assembly language task, put a label on a pointer to the base of the stack and define the label as an entry point. Then, in the calling MP/Fortran IV program, define the label **EXTERNAL,** and use the label as the actual parameter *stack_base*.

```
    ENT PRSTK
    NREL 1
PRSTK: PRST
    NREL
PASTK   BLK ?STKMIN + 10
```

Initially, MP/Fortran IV assumes that the stack at *stack_base* is greater than or equal to the MP/OS minimum stack size. Once the program is executing, you may then change the size of the stack.

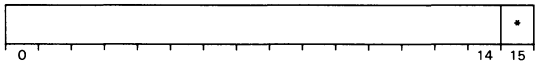
ITEST (function)

Tests a single bit in a word.

Syntax

ITEST*(word, position)*

where: *word* is an integer variable, or array element, and *position* is an integer constant, variable, array element, or expression whose value specifies the position in *word* of the bit you are testing.



**0* = *least significant bit*
15 = most significant bit

Example

```
LOGICAL ITEST
.
.
.
IF (ITEST(I,J)) GO TO 120
```

Notes

ITEST returns the integer value -1 (true) if the tested bit is one, and the integer 0 (false) if the tested bit is zero. If you wish the function to return the logical values T or F, you may declare it to be type **LOGICAL,** as in the example above.

Note that the bits in a word are numbered in reverse order to DGC assembler convention.

BTEST is a function identical to **ITEST**.

KILL (subroutine)

Terminates the executing task.

Syntax
CALL KILL

Notes
KILL can be thought of as a **STOP** statement for tasks.

LOGICAL (specification statement)

Specifies that all values assigned to the variable(s) named will be of data type logical.

Syntax
LOGICAL $v_1, v_2, \ldots, v_n$

where: each v is a variable name, an array name, a dimensioned array name, a function name, or a statement function argument name.

Example
```
LOGICAL THAT, SOUNDS(10,5)
```

Notes
A logical datum is stored in one word. The value .TRUE. is stored as octal 177777 (decimal -1), and .FALSE. as octal 000000. When testing for a logical value, any non-zero word will be treated as the value .TRUE.

Variables used for storage of logical data must be specified with the **LOGICAL** statement. It must appear in the program unit before any executable statements.

Dummy arguments may appear, and arrays may be dimensioned in the **LOGICAL** statement.

The data type of a variable may not be changed within a program unit.

MAX0 (function)

Selects the largest member from a set of integers, expressing the selection as an integer.

Syntax

MAX0$(i_1, i_2, \ldots i_n)$

where: i is an integer constant, variable, array element, or expression.

Example

```
ITOP = MAX0(INDEX, 100)
N = MAX0(11,34,22,12,27,29)
```

N will be set equal to 34.

MAX1 (function)

Selects the largest member from a set of single precision real numbers, expressing the selection as an integer.

Syntax

MAX1$(x_1, x_2, \ldots x_n)$

where: x is a single precision real constant, variable, array element, or expression.

Example

```
LARGE = MAX1(GREAT, 10.0)
N = MAX1(11.5, 11.376, 12.0, 11.11 11.8
```

N will be assigned the value 12 .

Notes

MAX1 generates an error message if the truncated real number exceeds 2 ** 15 -1, or is less than -(2 ** 15 -1).

MIN0 (function)

Selects the smallest member from a set of integers, expressing the selection as an integer.

Syntax

MIN0$(i_1, i_2, \ldots i_n)$

where: i is an integer constant, variable, array element, or expression.

Example

```
LIMIT = MINO(N(I),N(I+1)
N = MINO(11,34,22,12,27,29)
```

N will be set equal to 11 .

MIN1 (function)

Selects the smallest member from a set of single precision real numbers, expressing the selection as an integer.

Syntax

MIN1$(x_1, x_2, \ldots x_n)$

where: x is a single precision real constant, variable, array element, or expression.

Example

```
L = MIN1(Y,7.0)
N = MIN1(11.5, 11.376, 12.0, 11.1, 11.8)
```

N will be assigned the value 11

Notes

MIN1 generates an error message if the truncated real number exceeds 2 ** 15 -1, or is less than -(2 ** 15 -1).

MOD

Returns the integer remainder in the quotient of two integer arguments, i_1/i_2.

Syntax

MOD(i_1, i_2)

where: i_1 and i_2 are integer constants, variables, array elements, or expressions.

Example

```
INIT = 10 * MOD(IN1,IN2)
```

If IN1=8 and IN2=5, INIT receives the value 30 .

Notes

MOD is defined as:

arg_1 - $[arg_1/arg_2]$ * arg_2, where $[arg_1/arg_2]$ is the truncated value of that quotient.

If i_2 is equal to 0 or to 2**15, **MOD** generates an error message, and returns a value of 0.

MULTI (subroutine)

Re-enables scheduling of all other tasks.

Syntax

CALL MULTI

Notes

After you issue a call to **SINGL,** you should use this routine to reenable scheduling of other tasks.

The routine **REC** will perform an implied **CALL MULTI** if there are no data available when the call is issued; **CALL XMTW** will always perform an implied **CALL MULTI.**

The latest call to **MULTI** overrides all prior calls to **SINGL.**

NOT (function)

Complements (changes) each bit of its integer argument.

Syntax

NOT*(i)*

where: *i* is an integer constant, variable, array element, or expression.

Example

```
MASKS = NOT(MASKS)  ;masks has all but the rightmost 4 bits set
```

OPEN (subroutine)

Opens a device or disc file.

Syntax

CALL OPEN *(channel, pathname, < array | mode >, error [,size])*

where: *channel* is an integer variable or constant whose value specifies the number of the channel ($0 - 63_{10}$) on which *pathname* is opened. If the channel has an assigned device, this association is temporarily suspended until **CLOSE, FCLOS,** or **RESET** is called. No more than 16 channels may be open at any time.

pathname is the name of the file to be opened,

mode and *array* (alternate arguments in the command line) are ignored by the MP/Fortran IV compiler. They are included to provide compatibility among AOS, RDOS, and MP/OS. (*Mode* is an integer constant or variable, while *array* is a three-element integer array.)

error is an integer variable which will return one of the MP/Fortran IV or run-time error codes on completion of the call.

size is an integer constant or variable specifying the number of bytes that make up a record of a randomly organized file. You must give *size* if the file is not accessed sequentially.

Examples

```
CALL OPEN(3, "TEST", 2, IER, 128)
CALL OPEN(5, "X45", IAR, IER)
```

The following call will produce an error message:

```
CALL OPEN (1, "@TTI", IAR, IER)
```

because channel 10 is automatically opened to @TTI when MP/Fortran IV begins executing.

Notes

When you execute a call to **OPEN,** the file with the specified *pathname* will be opened on the channel specified by the value of *channel.* If *pathname* is a sequentially organized file to which information will be written, all previous information contained in the file will be

overwritten.

Only one channel at a time may be open to an input character device, such as the console keyboard.

OVERFLOW (subroutine)

Checks for floating point overflow.

Syntax

CALL OVERFLOW(*$$s_1$, $$s_2$, [< 'S' | 'N' >]*)

where: **$** is a symbol that must precede each of the first two arguments.

s_1 and *s_2* are statement labels.

'N' is an alternate optional argument. If you specify *'N'*, the routine will write all overflow error messages to **?ouch**.

'S' is an alternate optional argument. If you specify *'S'*, or if you do not specify a third argument for this routine, MP/Fortran IV will suppress error messages associated with floating point overflow or underflow.

Example

```
CALL OVERFLOW($999, $100, 'N')
```

Notes

OVERFLOW checks a system flag to determine whether or not non-integer arithmetic overflow has occurred since the last call to **OVERFLOW,** or since the start of execution, whichever was more recent.

If overflow has occurred, control returns to the statement labeled *s_1*.

If overflow has not occurred, control returns to the statement labeled *s_2*.

OVERLAY (specification statement)

Defines an overlay.

Syntax

OVERLAY *overlayname*

where: *overlayname* is the name of an overlay.

Example

```
OVERLAY VIRGINIA        ; Overlay program unit
.
.
END

OVERLAY WOOLF           ; Overlay program unit
.
.
END

EXTERNAL VIRGINIA,      ; Main program
WOOLF
REAL LIGHTHOUSE,
DALLOWAY
DIMENSION
DALLOWAY(23)
.
.
```

Notes

An **OVERLAY** statement must be the first statement (except for possible **COMPILER DOUBLE PRECISION, COMPILER NOSTACK,** or **CHANTASK** statements) in one of the program units belonging to an overlay. If a single overlay is created from two or more object files, each of which contains an **OVERLAY** statement, each *overlayname* specified in these statements is associated with that overlay. The overlay can then be referrred to by any one of the names.

An overlay name is an external symbol (like the names of subprograms) and must be unique within its first five characters from all other external symbols and all reserved words. You refer to overlays by their names when loading or releasing them. You must declare each overlay name **EXTERNAL** in any program unit which uses it. Do not declare it **EXTERNAL** in the program unit which defines it with an **OVERLAY** statement.

Overlays are sophisticated programming tools. We recommend that before you use them, you read the material on overlays in Part 3 of *MP/OS System Assembly Language Programmer's Reference,* (DGC No. 093-400001).

OVLOD (Subroutine)

Loads an overlay.

Syntax

CALL OVLOD*(channel, overlayname, flag, error)*

where: *(channel* is an integer variable ignored by MP/Fortran IV. This argument was retained to provide compatibility with RDOS and AOS programs, where it specifies the channel on which the overlay file was opened.

overlayname is the name of the overlay to be loaded. (This name is a symbol defined by an **OVERLAY** statement or by an **.ENTO** statement in assembler.)

flag is an integer variable or constant ignored by MP/Fortran IV.

error is an integer variable which will return one of the standard MP/Fortran IV or run-time error codes on completion of the call.

Example

```
INTEGER FLAG
EXTERNAL OTHELLO
.
.
.
CALL OVLOD(63, OTHELLO, FLAG, IER)
```

Notes

The subroutine **FOVLD** takes the same arguments and has the same results as **OVLOD**.

All overlay loads are conditional (i.e., if the requested overlay is currently in the node--perhaps called by another task--it will not be reloaded.) This implies that overlays may not be self-modifying. If the requested node contains another overlay that is in use, this call pends until the node is available.

Overlays are sophisticated programming tools. We recommend that before you use them, you read the material on overlays in Part 3 of *MP/OS System Assembly Language Programmer's Reference,* (DGC No. 093-400001).

OVREL (subroutine)

Releases an overlay.

Syntax

CALL OVREL*(overlay, error)*

where: *overlay* is the name of the overlay you wish to release.

error is an integer variable that returns one of the MP/Fortran IV or run-time error codes on completion of the call.

Example

```
EXTERNAL OROMEO
CALL OVREL(OROMEO, IER)
```

Notes

The subroutine **FOVRL** is the same as **OVREL**.

You must release the overlay occupying a given node before you can load another overlay into the node.

Overlays are sophisticated programming tools. For information on overlay nodes and overlay handling in general, please read Part 3, *MP/OS System Assembly Language Programmer's Reference,* (DGC No. 093-00001).

PARAMETER (specification statement)

Assigns a symbolic name to a constant.

Syntax

PARAMETER $v_1 = c_1, v_2 = c_2, \ldots, v_n = c_n$

where: each v is a variable name, and each c is a numerical or logical constant.

Examples

Example of **PARAMETER** statement:

```
PARAMETER PI=3.141592653, Q1=.1731523D-7
```

Examples of use of parameter K:

```
   PARAMETER K = 8
   COMMON / COMLABEL / IC1(K), IC2(K)
   DATA IC1/K*K/
   DO 100 I=1,K
   C1(I) = I*K
100   CONTINUE
   CALL SUBROUT(IC1, AB, K)
```

Notes

A parameter has the data type of its constant value. A parameter may be used anywhere that a constant of the same data type may be used.

The **PARAMETER** statement, if used, must appear before any executable statement in the program unit.

String constants cannot be associated with parameters.

PAUSE (statement)

Causes the task to cease executing until a go-ahead signal is received from the user.

Syntax

PAUSE

PAUSE *s*

where: *s* is a string of ASCII characters to be typed out at the console.

Example

```
PAUSE AT STATEMENT 75
```

Notes

When the **PAUSE** statement is reached, the program will type PAUSE, followed by the message, if any, and the literal:

```
STRIKE ANY KEY TO CONTINUE.
```

The program will then cease execution until the user types in any character.

Examples

```
      READ (13,5) G, B(1), C, B(2), D, B(3)
5     FORMAT(6F9.3)
      DIMENSION A(3,4)
      READ(11,6) A
      READ(11,6) A(1,1), A(2,1), . . . A(3,4)
6     FORMAT(12F8.2)
```

Both of these statements both read in the entire array.

For punctuation purposes, any portion of an I/O list can be enclosed in parentheses, except within the loop specification of a **DO-IMPLIED LIST**. For example, these four statements are all equivalent:

```
READ (12) M, O, P(I,J)
READ (12) (M), O, P(I,J)
READ (12) (M, O), (P(I,J))
READ (12) ((M,O), (P(I,J)))
```

Notes

In unformatted **READ** statements, the I/O list order completely prescribes the order in which data are read.

On input, the programmer delimits individual elements of data by separating them with commas or end-of-record indicators (carriage returns or newlines.) For example, to fill array A, which has six elements:

```
READ (11) A
```

will read from the console (channel 11). You can satisfy the **READ** by typing:

```
1,2,3,4,5,6 <newline>
```

or by typing:

```
1,2,3 <newline>
4,5 <newline>
6 <newline>
```

or by typing:

```
1, 2, 3, 1, -5E2, 0, .1E-3 <newline>
```

The **READ** will convert data types from integer to floating point or vice versa, if required by the internal data types of the variables specified in the I/O list.

Two channels are pre-opened at the beginning of MP/Fortran IV execution:

10	**?ouch**	Standard output channel
11	**?inch**	Standard input channel

A third channel is special: if you write to channel 12 without opening it, MP/Fortran IV will open it to **@LPT** for you.

To open a file, an I/O statement must reference one of the above channels, or you must use **OPEN** or **FOPEN** to open a channel to the file.

READ BINARY (statement)

Reads binary data from an external device or file.

Syntax

READ BINARY *(channel) list*

where: *channel* is an I/O channel number associated with the file or device. There are 64 channels (0 to 63_{10}.)

list is a list of names of variables, including arrays and array elements, which are to be read. You must specify the subscripts of array elements with unsubscripted integer variables or constants. *list* can include an implied **DO** loop (see **DO-IMPLIED LIST** in this dictionary).

End-of-File or Error Transfer of Control

READ BINARY
(channel, [format,] **ERR**$=n_1$*) [list]*

READ BINARY
(channel, [format,] **END**$=n_2$*) [list]*

READ BINARY
(channel, [format,] **ERR**$=n_1$, **END**$=n_2$*) [list]*

READ BINARY
(channel, [format,] **END**$=n_2$*)* **ERR**$=n_1$*) [list]*

where: n_1 is the return statement number for an I/O error.

n_2 is the return statement number for an end-of-file.

If an end-of-file is encountered during execution of a **READ** statement, execution of the program is terminated unless the end-of-file was prepared for in the **READ** statement.

With the transfer of control branches specified, the user can regain control after an end-of-file is encountered or an I/O error at the driver level (i.e., parity, record size) has been detected.

Example

```
READ BINARY (11) MASKA, FLAGA
```

Notes

Data is handled strictly in the order that variables are listed in the I/O list.

Data is transferred at two bytes per word, where the number of words transferred depends on the internal data representation:

- One word for integer data,
- Two words for real data,
- Four words for double precision and complex data,
- Eight words for double precision complex data.

The high or logical order or left byte is transferred first.

Two channels are pre-opened at the beginning of MP/Fortran IV execution:

10	**?ouch**	Standard output channel
11	**?inch**	Standard input channel

A third channel is special: if you write to channel 12 without opening it, MP/Fortran IV will open it to **@LPT** for you.

To open a file, an I/O statement must reference one of the above channels, or you must use **OPEN** or **FOPEN** to open a channel to the file.

PRI (subroutine)

Modifies the executing task's priority.

Syntax

CALL PRI*(priority-number)*

where: *priority-number* is an integer variable or constant, in the range 0 - 255_{10}, that gives the new priority of the task.

Example

```
CALL PRI(37)
```

Notes

The lower the number, the higher the priority.

You may change the priority of a task any number of times while it is active.

More than one task may have the same priority number. Tasks with the same priority are scheduled in a round robin fashion: their relative priorities are determined by how long each has been waiting for processing. Each time a task relinquishes control to the task scheduler, it is moved to the end of the queue of active tasks.

RDBLK (subroutine)

Reads a series of blocks from a file, without using a system buffer.

Syntax

CALL RDBLK*(channel, sblock, array, nblock, error [, iblk])*

where: *channel* is an integer constant or variable whose value specifies the number of the channel on which the file to be read from is opened.

sblock is an integer constant or variable whose value specifies the number of the first block to be read.

array is the name of an integer array that receives the blocks that are read. The array must be *nblock* * 256 words in length. (No error check is made on the adequacy of the array length.)

nblock is an integer constant or variable whose value specifies the number of consecutive blocks to be read.

error is an integer variable which will return one of the MP/Fortran IV or run-time error codes upon completion of the call.

iblk is an optional integer variable that is set to return the number of blocks read on encountering an end of file marker.

Example

```
CALL RDBLK(10, 100, IARR, 15, IER, IBLK)
```

Execution of this call causes 15 blocks to be read, starting from the 100th block, into array IARR. The IARR array must previously have been dimensioned to a length of 3840 words.

RDRW (subroutine)

Reads a series of records from a randomly organized file into an integer array.

Syntax

CALL RDRW*(channel, srec, array, nrec, error [, nbyte])*

where: *channel* is an integer constant or variable whose value specifies the number of the channel on which the randomly organized file to be read from is opened,

srec is an integer constant or variable whose value specifies the number of the first record to be read,

array is the name of an integer array that receives the records that are read. (No error check is made on the adequacy of the array length.)

nrec is an integer constant or variable whose value specifies the number of successive random records to be read,

error is an integer variable which will return one of the MP/Fortran IV or run-time error codes upon completion of the call,

nbyte is an optional integer variable that is set to return the number of bytes read if an EOF or Disk Full is encountered.

Example

```
CALL RDRW(15, 0, IARR, 20, IERR)
```

Notes

A call to **RDRW** has exactly the same syntax, and the same results, as a call to **READR**.

READ (statement)

Reads data from an external device or file.

Syntax

READ*(channel) [list]*

READ*(channel, format) [list]*

where: *channel* is an I/O channel number associated with the file or device. There are 64 channels (0 to 63_{10}.)

format is the label of the referenced **FORMAT** statement, or the name of an array containing the format specifications on formatted I/O).

list is a list of names of variables, including arrays and array elements, which are to be given values. You must specify the subscripts of array elements with unsubscripted integer variables or constants. If *list* is not given, *READ* will read and ignore an entire record. *List* can include an implied *DO* loop (see *DO-IMPLIED LIST* in this dictionary).

End-of-File or Error Transfer of Control

READ *(channel, [format,]* **ERR**$=n_1$*) [list]*

READ *(channel, [format,]* **END**$=n_2$*) [list]*

READ *(channel, [format,]* **ERR**$=n_1$, **END**$=n_2$*) [list]*

READ *(channel, [format,]* **END**$=n_2$, **ERR**$=n_1$*) [list]*

where: n_1 is the return statement label for an I/O error.

n_2 is the return statement label for an end-of-file.

If an end-of-file is encountered during execution of a **READ** statement, execution of the program is terminated unless the end-of-file was prepared for in the **READ** statement.

With the transfer of control branches specified, the user can regain control after an end-of-file is encountered or an I/O error at the driver level (i.e., parity, record size) has been detected.

READR (subroutine)

Reads a series of records from a randomly organized file into an integer array.

Syntax

CALL READR*(channel, srec, array, nrec, error [, nbyte])*

where: *channel* is an integer constant or variable whose value specifies the number of the channel on which the randomly organized file to be read from is opened,

srec is an integer constant or variable whose value specifies the number of the first record to be read,

array is the name of an integer array that receives the records that are read. (No error check is made on the adequacy of the array length.)

nrec is an integer constant or variable whose value specifies the number of successive random records to be read,

error is an integer variable which will return one of the MP/Fortran IV or run-time error codes upon completion of the call,

nbyte is an optional integer variable that is set to return the number of bytes read if an EOF or Disk Full is encountered.

Example

```
CALL READR(15, 0, IARR, 20, IERR)
```

Notes

A call to **RDRW** has exactly the same syntax, and the same results, as a call to **READR**.

REAL (specification statement)

Specifies that all values assigned to the variable(s) named will be of the floating point data type.

Syntax

REAL $v_1, v_2, \ldots, v_n$

where: each v is a variable name, an array name, a dimensioned array name, a function name, or a statement function argument name.

Example

```
REAL IDLE, LAZY, NOGOOD(10,5)
```

Notes

A real datum is a signed or unsigned floating point value stored in two 16-bit words.

The **REAL** specification statement may be used to override implicit data typing. The **REAL** statement must appear before any executable statements in the program unit.

Dummy arguments may appear, and arrays may be dimensioned in the **REAL** statement.

The data type of a variable may not be changed within a program unit.

REAL (function)

Returns the real part of a single precision complex number.

Syntax

REAL*(cx)*

where: *cx* is a single precision complex constant, variable, array element, or expression.

Example

```
COMPLEX IMAG
IMAG = (1.25, .625)
XNORM = REAL(IMAG) ; XNORM equals 1.25
```

Notes

REAL does not generate error messages.

REC (subroutine)

Receives a synchronously transmitted one-word message from another active task.

Syntax

CALL REC*(message-key, message-destination)*

where: *message-key* is an integer variable common to both transmitting and receiving tasks.

message-destination is an integer variable accessible by the receiving task.

Example

```
TASK SEG1
   COMMON KEY
   .
   .
   CALL REC(KEY, MDEST)
   .
   .
   END
   TASK SEG2
   COMMON KEY
   .
   .
   MSRCE = IRESULT
   CALL XMT(KEY, MSRCE, $17)
   .
   .
17    WRITE(10) "KEY ALREADY SET"
   .
   .
   END
```

Notes

A message transmitted by **XMT** or **XMTW** is received by execution of a call to the **REC** routine.

In the transmission of a message using corresponding **CALL XMT** and **CALL REC** statements, the statements may be executed in either order.

If **CALL XMT** is executed first, the value of its parameter *message-source* is assigned to *message-key*. When **CALL REC** is subsequently executed, *message-destination* is assigned the value of *message-key*, and *message-key* is assigned the value 0.

If **CALL REC** is executed before the corresponding **CALL XMT,** the receiving task is suspended until **CALL XMT** is executed. When **CALL XMT** is executed, *message-destination* is assigned the value of XMT variable

message-source, and the receiving task is placed in the ready state.

RENAM (subroutine)

Renames a disc file.

Syntax

CALL RENAM*(oldpathname, newpathname, error)*

where: *oldpathname* is the file name that you want to change.

newpathname is the new name you want to assign to the file.

error is an integer variable that returns one of the MP/Fortran IV or run-time error codes on completion of the call.

Example

```
CALL RENAM(''TEST'', ''SORT'', IER)
```

The following call will produce an error message:

```
CALL RENAM('@DPD0:AB', '@DPA0:B', IER)
```

because the path from the old file name to the new file name must cross from one device to another.

Notes

The **RENAM**ing cannot occur across devices.

The file you wish to **RENAM** cannot be open.

RESET (subroutine)

Closes all open files.

Syntax
CALL RESET

Notes
The MP/Fortran IV call to **RESET** does not affect overlays, as the MP/OS call does.

RETURN (statement)

Returns control from the logical end of a subprogram to the calling procedure.

Syntax
RETURN

RETURN *v*

where: *v* is a dummy integer variable whose value represents a statement number in the calling program.

Example
```
SUBROUTINE SUB(DUM, I, R1, Q, K)
INTEGER Q
.
.
.
RETURN Q
.
.
.
RETURN
END
```

When the subroutine **SUB** is referenced, the calling procedure passes a statement label to replace the dummy integer argument. The statement label must be preceded by a dollar sign ($), as in:

```
CALL SUB(A, K1, K2, $25, K3)
```

Notes
Execution of a **RETURN** without *v* is a normal return.

Control is returned to the calling procedure as follows:

1. Return from a subroutine is made to the statement following the **CALL** statement, or, in an abnormal return, to the statement with the line number specified by the **RETURN**.
2. Return from a function is made to the statement containing the function reference, and a value is substituted for the function in that statement.

An abnormal return must return to the immediately calling procedure.

REWIND (statement)

Causes the file to be positioned at its initial record.

Syntax

REWIND *channel*

where: *channel* is the channel number ($0\text{-}63_{10}$) associated with the file.

Examples

```
CALL OPEN (0, "FILE", 2, IER)
.
.
.
WRITE (0,1000) A,B,C
.
.
.
REWIND 0
READ (0,1000) D,E,F
```

SDATE (subroutine)

Sets the current date.

Syntax

CALL SDATE*(date-array, error)*

where: *date-array* is the name of a three-element integer array that will be set equal to the date.

error is an integer variable which will return one of the FORTRAN or system error codes.

Example

```
CALL SDATE(IAR, IER)
```

Notes

The date is expressed in the order: month, day, year; where January = 1, and year = year - 1900 (e.g., 1979 is returned as 79).

SIGN (function)

Transfers the sign of one single precision real number, x_2 to another single precision real number, x_1.

Syntax

SIGN(x_1, x_2)

where: x_1 and x_2 are both single precision real constants, variables, array elements, or expressions.

Example

```
X = 25.0
Y = -6.0
XNEW = SIGN(X,Y)  ; XNEW equals -25.0
```

Notes

SIGN does not generate error messages.

SIGN performs the operation: sign of x_2 * x_1.

SIN (function)

Computes the real sine of a single precision real argument.

Syntax

SIN(x)

where: x is a single precision real constant, variable, array element, or expression.

Example

```
V = SIN(T)
```

Notes

For any x expressed in radians, sine(x) = sine(y + 2$n$pi), where y is in the range 0 <= y < 2pi; hence, if you evaluate the sine of a large x, some of the information is wasted in specifying multiples of 2pi radians (360 degrees), which have no effect on the result. You will see this "waste" as a loss of significant digits in the answer.

SINGL (subroutine)

Disables scheduling of all other tasks.

Syntax
CALL SINGL

Notes
You should use this routine if you wish a task to complete an operation without interruption from other tasks. The routine **MULTI** reenables scheduling of other tasks.

The routine **REC** will perform an implied **CALL MULTI** if there are no data available when the call is issued; **CALL XMTW** will always perform an implied **CALL MULTI**.

The latest call to **SINGL** overrides all prior calls to **MULTI**.

SINH (function)

Computes the real hyperbolic sine of a single precision real number.

Syntax
SINH(*x*)
where: *x* is a single precision real constant, variable, array element, or expression.

Example
```
SW = SINH(SN)
```

Notes
SINH does not generate error messages.

SNGL (function)

Expresses the most significant part of a double precision real argument as a single precision real number.

Syntax

SNGL*(dr)*

where: *dr* is a double precision real constant, variable, array element, or expression.

Example

```
DOUBLE PRECISION FD
.
.
.
SUM = A + X + Y(I) + SNGL(FD)
```

SQRT (function)

Computes the single precision square root of a non-negative single precision real argument.

Syntax

SQRT*(x)*

where: *x* is a non-negative real constant, variable, array element, or expression.

Example

```
POWER = BASE**2
ROOT = SQRT(POWER)  ; BASE equals POWER
```

Notes

If the input argument is negative, the function generates an error message and calculates the square root of the absolute value of the argument.

STIME (subroutine)

Sets the time of day.

Syntax

CALL STIME*(array, error)*

where: *array* is a three-element integer array, specifying the time to be set in the order hours (range: 0 to 23), minutes (0 to 59), seconds (0 to 59).

error is an integer variable which will return one of the error codes on completion of the call.

Example

```
CALL STIME(IAR, IER)
```

Notes

The subroutine **FSTIM** also sets the time of day.

STOP (statement)

Causes unconditonal termination of program execution.

Syntax

STOP

STOP *s*

where: *s* is a string of ASCII characters to be printed out at the console when execution is terminated.

Example

```
.
.
.
STOP AND HAVE A GOOD DAY
```

Notes

The word **STOP** and the text string, if any, will be printed at the console.

BACK, FBACK, EBACK and **EXIT** will also terminate program execution.

SUBROUTINE (statement)

Defines a subroutine.

Syntax

SUBROUTINE *name* (a_1, a_2, . . ., a_n)

where: *Name* is the name of the subroutine. The name must be uniquely distinguishable within its first five characters. *Name* may appear in the subroutine only in the **SUBROUTINE** statement, immediately following the word **SUBROUTINE**.

Each *a* is a dummy (or "formal") argument. It may be a variable name, array name, or an external subprogram name (function or subroutine). The dummy argument is replaced by an actual argument when the subroutine is referenced. The argument list may be blank.

Example

```
    SUBROUTINE REV(ARRAY, N1, N2)
    DIMENSION ARRAY(100)
    N12 = N1 + N2
    MID = N12 / 2
    DO 50 N = N1, MID
    J = N12 - N
C USE TEMPORARY TO REVERSE
C ELEMENTS OF ARRAY
    TEMP = ARRAY(N)
    ARRAY(N) = ARRAY(J)
    ARRAY(J) = TEMP
50     CONTINUE
    RETURN
    END
```

Notes

A subroutine subprogram begins with a **SUBROUTINE** statement.

A subroutine subprogram is referenced by a **CALL** statement (see **CALL**).

A subroutine is external (separately compiled). An external subroutine name that is used as an actual argument in the referencing program unit must appear in an **EXTERNAL** statement in the referencing program unit.

A subroutine returns values to the calling program unit only through actual-dummy argument correspondence. Control returns to the calling program unit at the statement following the subroutine call, unless the subroutine has executed an abnormal return (see **RETURN**).

A subroutine returns control to the referencing program when it executes a **RETURN** or **END** statement.

Dummy argument names cannot appear in **DATA, COMMON,** or **EQUIVALENCE** statements in the function subprogram.

The subroutine cannot contain statements that define other program units, e.g., it cannot contain another **SUBROUTINE** statement, a **BLOCK DATA** statement, a **TASK** statement, or a **FUNCTION** statement.

When a program unit calls a subroutine, actual argument names replace dummy argument names of given structures, as shown below:

Formal Argument	Actual Argument
Variable name	Variable name, array element name, or any expression*
Array name	Array name, or array element name**
Name that can be used as a function call***	External function name
Name that can be used as a subroutine name in a CAll statement***	External subroutine name

**When the actual argument is an expression, its value is passed.*

***When an array name is passed:*
formal length <= actual array length.
When an array element name is passed:
formal length <= actual array length + 1 - the actual array's subscript.

A formal argument that stands for a function or subroutine name cannot be defined or redefined in the subroutine.

We do not recommend that you directly reference the same variable as two different parameters of a subroutine, as for example:

```
CALL SUB(A,A).
```

If you want to pass the same variable twice, enclose its name in parentheses:

```
CALL SUB((A), (A))
```

These parentheses tell the compiler to copy the argument, and pass the address of the copy. In this case the subroutine will not affect the value of the variable in the calling program. For example:

```
CALL SUB(X, (Y))
```

means that the subroutine may return a new value for **X,** but cannot affect the value of **Y** in the calling program.

SWAP (subroutine)

Saves the current program's memory image on disc, and loads another program into memory from disk.

Syntax

CALL SWAP *(filename, error)*

where: *filename* is the name of the file to be executed next.

error is an integer variable which will return one of the MP/Fortran IV or run-time error messages on completion of the call.

Example

```
CALL SWAP("ABC", IER)
```

Notes

FSWAP is the same as **SWAP**, except that **FSWAP** does not provide an error return.

Program swapping allows memory images of programs to be saved and called for execution more than once during a program's execution. Each time it is called, the program will resume execution from the point of suspension. If the execution level of the calling program is *n*, *filename* executes at level $n + 1$.

If an attempt is made to nest swaps to a level deeper than eight, an error will result.

Each program to which you swap must contain a complete program.

This routine does not pass **COMMON** under MP/Fortran IV.

For more information on program management, refer to the *MP/OS Assembly Language Programmer's Reference.*

SYS (subroutine)

Executes any MP/OS system call from the MP/Fortran IV environment.

Syntax

CALL SYS*(call-number, option, ac0, ac1, ac2, error)*

where: *call-number* is an integer constant, variable, or expression specifying the number of the MP/OS system call, or the name of a MP/OS library call which has been defined as **EXTERNAL**.

option is an integer constant, variable or expression specifying any selected option.

ac0, ac1, and *ac2* are integer constants, variables, or expressions whose values correspond to Accumulators 0, 1, and 2 in the Assembly language call. Their significance varies with the particular system call used.

error is an integer variable that returns a MP/Fortran IV or run-time error on completion of the call.

Example

```
SCHAR = 26
CNAS = 1
CALL SYS(SCHAR, 0, BYTEADDR(''@TTI''), CNAS, 0,
IER)
```

This call informs the system that your input device is non-ANSI standard, e.g., a teletype.

Notes

The MP/OS system calls and the options available are described in Part 3, *MP/OS System Assembly Language Programmer's Reference,* (DGC No. 093-400001).

The following externals are provided for library calls. Please refer to the *MP/OS Assembly language Programmer's Reference* (093-40001) for details of the library calls.

```
CDAY
CTOD
DELAY
ERMSG
GDAY
GNFN
GTOD
MSEC
SLIST
TMSG
```

TAN (function)

Computes the single precision real tangent of a single precision real argument.

Syntax

TAN*(x)*

where: *x* is a single precision real constant, variable, array element, or expression.

Example

```
ROBT = TAN(R2D2)
```

Notes

For any *x* expressed in radians, tangent(*x*) = tangent(*y* + *n*pi), where *y* is in the range 0 >= *y* < pi; hence, if you evaluate the tangent of a large *x*, some of the information is wasted in specifying multiples of pi radians (180 degrees), which have no effect on the result. You will see this "waste" as a loss of significant digits in the answer.

TANH (function)

Computes the single precision real hyperbolic tangent of a single precision real argument.

Syntax

TANH*(x)*

where: *x* is a single precision real constant, variable, array element, or expression.

Example

```
HYP = TANH(BOL) ,
```

Notes

The routine generates error messages on underflow and overflow.