EXPANDED DESCRIPTION OF FORTRAN 90 / 95 INTRINSIC PROCEDURES

This document contains an expanded description of the intrinsic procedures built into the Fortran 90 and Fortran 95 languages, and provides some suggestions for their proper use. It is , and is intended as a supplement to Appendix B of ***Introduction to Fortran 90/95***. This document may be freely used in conjunction with that text.

All of the intrinsic procedures that are present in Fortran 90 are also present in Fortran 95, although some have additional arguments. Those procedures which are only in Fortran 95 and those procedures which have additional arguments in Fortran 95 are highlighted in the tables and discussions below.

The majority of Fortran intrinsic procedures are functions, although there are a few intrinsic subroutines.

B.1. Classes of Intrinsic Procedures

Fortran 90 / 95 intrinsic procedures can be broken down into three classes: elemental, inquiry, or transformational. An **elemental function**[1] is one that is specified for scalar arguments, but which may also be applied to array arguments. If the argument of an elemental function is a scalar, then the result of the function will be a scalar. If the argument of the function is an array, then the result of the function will be an array of the same shape as the input argument. If there is more than one input argument, all of the arguments must have the same shape. If an elemental function is applied to an array, the result will be the same as if the function were applied to each element of the array on an element-by-element basis.

An **inquiry function** or **inquiry subroutine** is a procedure whose value depends on the properties of an object being investigated. For example, the function `PRESENT(A)` is an inquiry function that returns a true value if the optional argument `A` is present in a procedure call. Other inquiry functions can return properties of the system used to represent real numbers and integers on a particular processor.

A **transformational function** is a function that has one or more array-valued arguments or an array-valued result. Unlike elemental functions which operate on an element-by-element basis, transformational functions operate on arrays as a whole. The output of a transformational function will often not have the same shape as the input arguments. For example, the function `DOT_PRODUCT` has two vector input arguments of the same size, and produces a scalar output.

B.2. Alphabetical List of Intrinsic Procedures

Table B-1 contains an alphabetical listing of the intrinsic procedures included in Fortran 90 and Fortran 95. The table is organized into 5 columns. The first column of the

[1] One intrinsic subroutine is also elemental.

table contains the *generic name* of each procedure, and its calling sequence. The calling sequence is represented by the keywords associated with each argument. Mandatory arguments are shown in roman type, and optional arguments are shown in italics. The use of keywords is optional, but they must be supplied for optional arguments if earlier optional arguments in the calling sequence are missing, or if the arguments are specified in a non-default order (see Section 8.3). For example, the function SIN has one argument, and the keyword of the argument is X. This function can be invoked either with or without the keyword, so the following two statements are equivalent.

```
result = sin(X=3.141593)
result = sin(3.141593)
```

Another example is the function MAXVAL. This function has one required argument and two optional arguments:

```
MAVXAL ( ARRAY, DIM, MASK )
```

If all three calling values are specified in that order, then they may be simply included in the argument list without the keywords. However, if the MASK is to be specified without DIM, then keywords must be used.

```
value = MAVXAL ( array, MASK=mask )
```

The types of the most common argument keywords are as shown below (any kind of the specified type may be used):

A	Any
BACK	Logical
DIM	Integer
I	Integer
KIND	Integer
MASK	Logical
STRING	Character
X, Y	Numeric (Integer, real, or complex)
Z	Complex

For the types of other keywords, refer to the detailed procedure descriptions below.

The second column contains the *specific name* of an intrinsic function, which is the name by which the function must be called if it is to appear in an INTRINSIC statement and be passed to another procedure as an actual argument. If this column is blank, then the procedure does not have a specific name, and so may not be used as a calling argument. The types of arguments used with the specific functions are:

c, c1, c2, ...	Default Complex
d, d1, d2, ...	Double Precision Real
i, i1, i2, ...	Default Integer

`r, r1, r2, ...`	Default Real
`l, l1, l2, ...`	Logical
`str1, str2, ...`	Character

The third column contains the type of the value returned by the procedure if it is a function. Obviously, intrinsic subroutines do not have a type associated with them. The fourth column is a reference to the section of this document in which the procedure is described, and the fifth column is for notes which are found at the end of the Table.

Those procedures which are only present in Fortran 95 are shown with a shaded background.

Table B-1: Specific and Generic Names for All Fortran 90/95 Intrinsic Procedures

Generic name, keyword(s), and calling sequence	Specific name	Function type	Section	Notes

`ABS(A)`	`ABS(r)`
	`CABS(c)`
	`DABS(d)`
	`IABS(i)`
		Argument type
		Default real
		Default real
		Double Prec.
		Default integer
			B.3	2

`ACHAR(I)`		Character(1)	B.7	
`ACOS(X)`	`ACOS(r)` `DACOS(d)`	Argument type Default Real Double Prec.	B.3	
`ADJUSTL(STRING)`		Character	B.7	
`ADJUSTR(STRING)`		Character	B.7	
`AIMAG(Z)`	`AIMAG(c)`	Real	B.3	
`AINT(A, KIND)`	`AINT(r)` `DINT(d)`	Argument type Default Real Double Prec.	B.3	
`ALL(MASK, DIM)`		Logical	B.8	
`ALLOCATED(ARRAY)`		Logical	B.9	
`ANINT(A, KIND)`	`ANINT(r)` `DNINT(d)`	Argument type Real Double Prec.	B.3	
`ANY(MASK, DIM)`		Logical	B.8	

`ASIN(X)`	`ASIN(r)`
	`ASIN(r)`
	`DASIN(d)`
		Argument type
		Real
		Double Prec.		

`ASSOCIATED(POINTER,TARGET)`		Logical	B.9	
`ATAN(X)`	`ATAN(r)` `DATAN(d)`	Argument type Real Double Prec.	B.3	
`ATAN2(Y,X)`	`ATAN2(r2,r1)` `DATAN2(d2,d1)`	Argument type Real Double Prec.	B.3	
`BIT_SIZE(I)`		Integer	B.4	
`BTEST(I,POS)`		Logical	B.6	
`CEILING(A,KIND)`		Integer	B.3	4
`CHAR(I, KIND)`		Character(1)	B.7	

CMPLX(X, *Y*, *KIND*)		Complex	B.3	
CONGJ(X)	CONJG(c)	Complex	B.3	
COS(X)	 CCOS(c) COS(r) DCOS(d)	Argument type Complex Real Double Prec.	B.3	
COSH(X)	 COSH(r) DCOSH(d)	Argument type Real Double Prec.	B.3	
COUNT(MASK, *DIM*)		Integer	B.8	
CPU_TIME(TIME)		Subroutine	B.5	5
CSHIFT(ARRAY, *SHIFT*, *DIM*)		Array type	B.8	
DATE_AND_TIME(*DATE*, *TIME*, *ZONE*, *VALUES*)		Subroutine	B.5	
DBLE(A)		Double Prec.	B.3	
DIGITS(X)		Integer	B.4	
DIM(X,Y)	 DDIM(d1,d2) DIM(r1,r2) IDIM(i1,i2)	Argument type Double Prec. Real Integer	B.3	
DOT_PRODUCT(VECTOR_A, VECTOR_B)		Argument type	B.3	
DPROD(X,Y)	DPROD(x1,x2)	Double Prec.	B.3	
EOSHIFT(ARRAY, SHIFT, *BOUNDARY*, *DIM*)		Array type	B.8	
EPSILON(X)		Real	B.4	
EXP(X)	 CEXP(c) DEXP(d) EXP(r)	Argument type Complex Double Prec. Real	B.3	
EXPONENT(X)		Integer	B.4	
FLOOR(A,*KIND*)		Integer	B.3	4
FRACTION(X)		Real	B.4	
HUGE(X)		Argument type	B.4	
IACHAR(C)		Integer	B.7	
IAND(I,J)		Integer	B.6	
IBCLR(I,POS)		Argument type	B.6	
IBITSI,POS,LEN)		Argument type	B.6	
IBSET(I,POS)		Argument type	B.6	
ICHAR(C)		Integer	B.7	
IEOR(I,J)		Argument type	B.6	
INDEX(STRING, SUBSTRING, *BACK*)	INDEX(str1,str2)	Integer	B.7	
INT(A, *KIND*)	 IDINT(i) IFIX(r)	Integer Integer Integer	B.3	 1 1
IOR(I,J)		Argument type	B.6	
ISHFT(I,SHIFT)		Argument type	B.6	
ISHFTC(I,SHIFT,*SIZE*)		Argument type	B.6	
KIND(X)		Integer	B.4	
LBOUND(ARRAY, *DIM*)		Integer	B.8	

LEN(STRING)	LEN(str)	Integer	B.7	
LEN_TRIM(STRING)		Integer	B.7	
LGE(STRING_A, STRING_B)		Logical	B.7	
LGT(STRING_A, STRING_B)		Logical	B.7	
LLE(STRING_A, STRING_B)		Logical	B.7	
LLT(STRING_A, STRING_B)		Logical	B.7	
LOG(X)		Argument type	B.3	
	ALOG(r)	Real		
	CLOG(c)	Complex		
	DLOG(d)	Double Prec.		
LOG10(X)		Argument type	B.3	
	ALOG10(r)	Real		
	DLOG10(d)	Double Prec.		
LOGICAL(L, *KIND*)		Logical	B.3	
MATMUL(MATRIX_A, MATRIX_B)		Argument type	B.3	
MAX(A1,A2,*A3, ...*)		Argument type	B.3	
	AMAX0(i1,i2, ...)	Real		1
	AMAX1(r1,r2, ...)	Real		1
	DMAX1(d1,d2,...)	Double Prec.		1
	MAX0(i1,i2,...)	Integer		1
	MAX1(r1,r2,...)	Integer		1
MAXEXPONENT(X)		Integer	B.4	
MAXLOC(ARRAY, *DIM*, *MASK*)		Integer	B.8	6
MAXVAL(ARRAY, *DIM*, *MASK*)		Argument type	B.8	
MERGE(TSOURCE,FSOURCE,MASK)		Argument type	B.8	
MIN(A1,A2,*A3, ...*)		Argument type	B.3	
	AMIN0(i1,i2, ...)	Real		1
	AMIN1(r1,r2, ...)	Real		1
	DMIN1(d1,d2,...)	Double Prec.		1
	MIN0(i1,i2,...)	Integer		1
	MIN1(r1,r2,...)	Integer		1
MINEXPONENT(X)		Integer	B.4	
MINLOC(ARRAY, *DIM*, *MASK*)		Integer	B.8	6
MINVAL(ARRAY, *DIM*, *MASK*)		Argument type	B.8	
MOD(A,P)		Argument type	B.3	
	AMOD(r1,r2)	Real		
	MOD(i,j)	Integer		
	DMOD(d1,d2)	Double Prec.		
MODULO(A,P)		Argument type	B.3	
MVBITS(FROM, FROMPOS, LEN, TO, TOPOS)		Subroutine	B.6	
NEAREST(X,S)		Real	B.3	
NINT(A, *KIND*)		Integer	B.3	
	IDNINT(i)	Integer		
	NINT(x)	Integer		
NOT(I)		Argument type	B.6	
NULL(*MOLD*)		Pointer	B.8	5
PACK(ARRAY, MASK, *VECTOR*)		Argument type	B.8	
PRECISION(X)		Integer	B.4	
PRESENT(A)		Logical	B.9	

PRODUCT(ARRAY, *DIM*, *MASK*)		Argument type	B.8	
RADIX(X)		Integer	B.4	
RANDOM_NUMBER(HARVEST)		Subroutine	B.3	
RANDOM_SEED(*SIZE*, *PUT*, *GET*)		Subroutine	B.3	
RANGE(X)		Integer	B.4	
REAL(A, *KIND*)	 FLOAT(i) SNGL(d)	Real Real Real	B.3	 1 1
REPEAT(STRING, NCOPIES)		Character	B.7	
RESHAPE(SOURCE,SHAPE,*PAD*, *ORDER*)		Argument type	B.8	
RRSPACING(X)		Argument type	B.4	
SCALE(X, I)		Argument type	B.4	
SCAN(STRING, SET, *BACK*)		Integer	B.7	
SELECTED_INT_KIND(R)		Integer	B.4	
SELECTED_REAL_KIND(*P,R*)		Integer	B.4	3
SET_EXPONENT(X, I)		Argument type	B.4	
SHAPE(SOURCE)		Integer	B.8	
SIGN(A,B)	 DSIGN(d1,d2) ISIGN(i1,i2) SIGN(r1,r2)	Argument type Double Prec. Integer Real	B.3	
SIN(X)	 CSIN(c) DSIN(d) SIN(r)	Argument type Complex Double Prec. Real	B.3	
SINH(X)	 DSINH(d) SINH(r)	Argument type Double Prec. Real	B.3	
SIZE(ARRAY, *DIM*)		Integer	B.8	
SPACING(X)		Argument type	B.4	
SPREAD(SOURCE, DIM, NCOPIES)		Argument type	B.8	
SQRT(X)	 CSQRT(c) DSQRT(d) SQRT(r)	Argument type Complex Double Prec. Real	B.3	
SUM(ARRAY, *DIM*, *MASK*)		Argument type	B.8	
SYSTEM_CLOCK(COUNT, COUNT_RATE, COUNT_MAX)		Subroutine	B.5	
TAN(X)	 DTAN(d) TAN(r)	Argument type Double Prec. Real	B.3	
TANH(X)	 DTANH(d) TANH(r)	Argument type Double Prec. Real	B.3	
TINY(X)		Real	B.4	
TRANSFER(SOURCE, MOLD, *SIZE*)		Argument type	B.8	
TRANSPOSE(MATRIX)		Argument type	B.8	
TRIM(STRING)		Character	B.7	
UBOUND(ARRAY, *DIM*)			B.8	

UNPACK(VECTOR, MASK, FIELD)		Argument type	B.8	
VERIFY(STRING, SET, *BACK*)		Integer	B.7	

1. These intrinsic functions cannot be passed to procedures as calling arguments.
2. The result of function CABS is real with the same kind as the input complex argument.
3. At least one of P and R must be specified in any given call.
4. Argument KIND is only available in Fortran 95 for this function.
5. These procedures are only available in Fortran 95.
6. The argument *DIM* is only available in the Fortran 95 version of functions MAXLOC and MINLOC.

These intrinsic procedures are divided into broad categories based on their functions. Refer to Table B-1 to determine which of the following sections will contain a description of any particular function of interest.

The following information applies to all of the intrinsic procedure descriptions:

1. All arguments of all intrinsic functions have INTENT(IN). In other words, all of the functions are pure. The intent of subroutine arguments are specified in the description of each subroutine.
2. Optional arguments are shown in italics in all calling sequences.
3. When a function has an optional KIND dummy argument, then the function result will be of the kind specified in that argument. If the KIND argument is missing, then the result will be of the default kind. If the KIND argument is specified, it must correspond to a legal kind on the specified processor, or the function will abort. The KIND argument is always an integer.
4. When a procedure is said to have two arguments of the same type, it is understood that they must also be of the same kind. If this is not true for a particular procedure, the fact will be explicitly mentioned in the procedure description.
5. The lengths of arrays and character strings will be shown by an appended number in parentheses. For example, the expression

   Integer(*m*)

   implies that a particular argument is an integer array containing *m* values.

B.3. Mathematical and Type Conversion Intrinsic Procedures

ABS(A)

- Elemental function of the same type and kind as A
- Returns the absolute value of A, |A|.
- If A is complex, the function returns $\sqrt{\text{real}^2 + \text{imag}^2}$.

ACOS(X)

- Elemental function of the same type and kind as X
- Returns the inverse cosine of X
- Argument is Real of any kind, with $|X| \le 1.0$, and $0 \le \text{ACOS}(X) \le \pi$.

`AIMAG(Z)`

- Real elemental function of the same kind as `Z`
- Returns the imaginary part of complex argument `Z`

`AINT(A,KIND)`

- Real elemental function
- Returns `A` truncated to a whole number. `AINT(A)` is the largest integer which is smaller than `|A|`, with the sign of `A`. For example, `AINT(3.7)` is 3.0, and `AINT(-3.7)` is -3.0.
- Argument `A` is Real; optional argument *`KIND`* is Integer

`ANINT(A,KIND)`

- Real elemental function
- Returns the nearest whole number to `A`. For example, `ANINT(3.7)` is 4.0, and `AINT(-3.7)` is -4.0.
- Argument `A` is Real; optional argument *`KIND`* is Integer

`ASIN(X)`

- Elemental function of the same type and kind as `X`
- Returns the inverse sine of `X`
- Argument is Real of any kind, with $|X| \le 1.0$, and $-\pi/2 \le \texttt{ASIN(X)} \le \pi/2$.

`ATAN(X)`

- Elemental function of the same type and kind as `X`
- Returns the inverse tangent of `X`
- Argument is Real of any kind, with $-\pi/2 \le \texttt{ATAN(X)} \le \pi/2$.

`ATAN2(Y,X)`

- Elemental function of the same type and kind as `X`
- Returns the inverse tangent of `Y/X` in the range $-\pi < \texttt{ATAN2(Y,X)} \le \pi$.
- `X,Y` are Real of any kind, and must be of same kind
- Both `X` and `Y` cannot be simultaneously 0.

`CEILING(A,KIND)`

- Integer elemental function
- Returns the smallest integer $\ge$ `A`. For example, `CEILING(3.7)` is 4, and `CEILING(-3.7)` is -3.
- Argument `A` is Real of any kind; optional argument *`KIND`* is Integer
- Argument *`KIND`* is only available in Fortran 95

`CMPLX(X,Y,KIND)`

- Complex elemental function
- Returns a complex value as follows:
  1. If `X` is complex, then `Y` must not exist, and the value of `X` is returned.
  2. If `X` is not complex, and `Y` doesn't exist, then the returned value is `(X,0)`.

3. If X is not complex and Y exists, then the returned value is (X,Y).

- X is Complex, Real, or Integer, Y is Real or Integer, and *KIND* is an Integer

CONJG(Z)

- Complex elemental function of the same kind as Z
- Returns the complex conjugate of Z
- Z is Complex

COS(X)

- Elemental function of the same type and kind as X
- Returns the cosine of X
- X is Real or Complex

COSH(X)

- Elemental function of the same type and kind as X
- Returns the hyperbolic cosine of X
- X is Real

DIM(X,Y)

- Elemental function of the same type and kind as X
- Returns X-Y if > 0; otherwise returns 0.
- X and Y are Integer or Real; both must be of the same type and kind

DBLE(A)

- Double precision real elemental function
- Converts value of A to double precision real
- A is numeric. If A is complex, then only the real part of A is converted.

DOT_PRODUCT(VECTOR_A,VECTOR_B)

- Transformational function of the same type as VECTOR_A
- Returns the dot product of numeric or logical vectors.
- Arguments are Numeric or Logical vectors. Both vectors must be of the same type, kind, and length.

DPROD(X,Y)

- Double precision real elemental function
- Returns the double precision product of X and Y
- Arguments X and Y are default real

EXP(X)

- Elemental function of the same type and kind as X
- Returns e^x
- X is Real or Complex

`FLOOR(A,KIND)`

- Integer elemental function
- Returns the largest integer ≤ `A`. For example, `FLOOR(3.7)` is 3, and `FLOOR(-3.7)` is -4.
- Argument `A` is Real of any kind; optional argument *`KIND`* is Integer
- Argument *`KIND`* is only available in Fortran 95

`INT(A,KIND)`

- Integer elemental function
- This function truncates `A` and converts it into an integer. If `A` is complex, only the real part is converted. If `A` is integer, this function changes the kind only.
- `A` is numeric; optional argument *`KIND`* is Integer.

`LOG(X)`

- Elemental function of the same type and kind as `X`
- Returns $\log_e(x)$
- `X` is Real or Complex. If Real, `X` > 0. If Complex, `X` ≠ 0.

`LOG10(X)`

- Elemental function of the same type and kind as `X`
- Returns $\log_{10}(x)$
- `X` is Real and positive.

`LOGICAL(L,KIND)`

- Logical elemental function
- Converts the logical value `L` to the specified kind.
- `L` is Logical, and *`KIND`* is integer.

`MATMUL(MATRIX_A,MATRIX_B)`

- Transformational function of the same type and kind as `MATRIX_A`
- Returns the *matrix product* of numeric or logical matrices. The resulting matrix will have the same number of rows as `MATRIX_A` and the same number of columns as `MATRIX_B`.
- Arguments are Numeric or Logical matrices. Both matrices must be of the same type and kind, and of compatible sizes. The following constraints apply:
  1. In general, both matrices are of rank 2.
  2. `MATRIX_A` may be rank-1. If so, `MATRIX_B` must be rank-2 with only one column.
  3. In all cases, the number of columns in `MATRIX_A` must be the same as the number of rows in `MATRIX_B`.

`MAX(A1,A2,A3,...)`

- Elemental function of same kind as its arguments
- Returns the maximum value of `A1`, `A2`, etc.

- Arguments may be Real or Integer; all must be of the same type

`MIN(A1,A2,`*`A3`*`,...)`

- Elemental function of same kind as its arguments
- Returns the minimum value of `A1`, `A2`, etc.
- Arguments may be Real or Integer; all must be of the same type

`MOD(A1,P)`

- Elemental function of same kind as its arguments
- Returns the value `MOD(A,P) = A - P*INT(A/P)` if $P \neq 0$. Results are processor dependent if $P = 0$.
- Arguments may be Real or Integer; they must be of the same type
- Examples:

Function	Result
`MOD(5,3)`	2
`MOD(-5,3)`	-2
`MOD(5,-3)`	2
`MOD(-5,-3)`	-2

`MODULO(A1,P)`

- Elemental function of same kind as its arguments
- Returns the modulo of `A` with respect to `P` if $P \neq 0$. Results are processor dependent if $P = 0$.
- Arguments may be Real or Integer; they must be of the same type
- If $P > 0$, then the function determines the positive difference between `A` and then next lowest multiple of `P`. If $P < 0$, then the function determines the negative difference between `A` and then next highest multiple of `P`.
- Results agree with the `MOD` function for two positive or two negative arguments; results disagree for arguments of mixed signs.
- Examples:

Function	Result	Explanation
`MODULO(5,3)`	2	5 is 2 up from 3
`MODULO(-5,3)`	1	-5 is 1 up from -6
`MODULO(5,-3)`	-1	5 is 1 down from 6
`MODULO(-5,-3)`	-2	-5 is 2 down from -3

`NEAREST(X,S)`

- Real elemental function
- Returns the nearest machine-representable number different from `X` in the direction of `S`. The returned value will be of the same kind as `X`.
- `X` and `S` are Real, and $S \neq 0$

`NINT(A,`*`KIND`*`)`

- Integer elemental function

- Returns the nearest integer to the real value A.
- A is Real

RANDOM_NUMBER(HARVEST)

- Intrinsic subroutine
- Returns pseudo-random number(s) from a uniform distribution in the range $0 \le$ HARVEST < 1. HARVEST may be either a scalar or an array. If it is an array, then a separate random number will be returned in each element of the array.
- Arguments:

Keyword	**Type**	**Intent**	**Description**
HARVEST	Real	OUT	Holds random numbers. May be scalar or array.

RANDOM_SEED(*SIZE*,*PUT*,*GET*)

- Intrinsic subroutine
- Performs three functions: (1) restarts the pseudo-random number generator used by subroutine RANDOM_NUMBER, (2) gets information about the generator, and (3) puts a new seed into the generator.
- Arguments:

Keyword	**Type**	**Intent**	**Description**
SIZE	Integer	OUT	Number of integers used to hold the seed (n)
PUT	Integer(m)	IN	Set the seed to the value in *PUT*. Note that $m \ge n$.
GET	Integer(m)	OUT	Get the current value of the seed. Note that $m \ge n$.

- *SIZE* is an Integer, and *PUT* and *GET* are Integer arrays. All arguments are optional, and at most one can be specified in any given call.
- Functions:
  1. If no argument is specified, the call to RANDOM_SEED restarts the pseudo-random number generator.
  2. If *SIZE* is specified, then the subroutine returns the number of integers used by the generator to hold the seed.
  3. If *GET* is specified, then the current random generator seed is returned to the user. The integer array associated with keyword *GET* must be at least as long as *SIZE*.
  4. If *PUT* is specified, then the value in the integer array associated with keyword *PUT* is set into the generator as a new seed. The integer array associated with keyword *PUT* must be at least as long as *SIZE*.

REAL(A,*KIND*)

- Real elemental function
- This function converts A into a real value. If A is complex, it converts the real part of A only. If A is real, this function changes the kind only.
- A is numeric; *KIND* is Integer.

`SIGN(A,B)`

- Elemental function of same kind as its arguments
- Returns the value of `A` with the sign of `B`.
- Arguments may be Real or Integer; they must be of the same type

`SIN(X)`

- Elemental function of the same type and kind as `X`
- Returns the sine of `X`
- `X` is Real or Complex

`SINH(X)`

- Elemental function of the same type and kind as `X`
- Returns the hyperbolic sine of `X`
- `X` is Real

`SQRT(X)`

- Elemental function of the same type and kind as `X`
- Returns the square root of `X`
- `X` is Real or Complex
- If `X` is real, `X` must be ≥ 0. If `X` is complex, then the real part of `X` must be ≥ 0. If `X` is purely imaginary, then the imaginary part of `X` must be ≥ 0.

`TAN(X)`

- Elemental function of the same type and kind as `X`
- Returns the tangent of `X`
- `X` is Real

`TANH(X)`

- Elemental function of the same type and kind as `X`
- Returns the hyperbolic tangent of `X`
- `X` is Real

B.4. Kind and Numeric Processor Intrinsic Functions

Many of the functions in this section are based on the Fortran models for Integer and Real data. These models must be understood in order to make sense of the values returned by the functions.

Fortran uses **numeric models** to insulate a programmer from the physical details of how bits are laid out in a particular computer. For example, some computers use two's complement representations for numbers while other computers use sign-magnitude representations for numbers. Approximately the same range of numbers can be represented in either case, but the bit patterns are different. The numeric models tell the programmer what range and precision can be represented by a given type and kind of numbers without requiring a knowledge of the physical bit layout on a particular machine.

The Fortran model for an integer *i* is

$$i = s \times \sum_{k=0}^{q-1} w_k \times r^k \qquad \text{(B-1)}$$

where *r* is an integer exceeding one, *q* is a positive integer, each w_k is a nonnegative integer less than *r*, and *s* is +1 or -1. The values of *r* and *q* determine the set of model integers for a processor. They are chosen to make the model fit as well as possible to the machine on which the program is executed. Note that this model is independent of the actual bit pattern used to store integers on a particular processor.

The value *r* in this model is the **radix** or base of the numbering system used to represent integers on a particular computer. Essentially all modern computers use a base 2 numbering system, so *r* is 2. If *r* is 2, then the value *q* is one less than the number of bits used to represent an integer (one bit is used for the sign of the number). For a typical 32-bit integer on a base 2 computer, the model of an integer becomes

$$i = \pm \sum_{k=0}^{30} w_k \times 2^k \qquad \text{(B-2)}$$

where each w_k is either 0 or 1.

The Fortran model for a real number *x* is

$$x = \begin{cases} 0 & \text{or} \\ s \times b^e \times \sum_{k=1}^{p} f_k \times b^{-k} \end{cases} \qquad \text{(B-3)}$$

where *b* and *p* are integers exceeding one, each f_k is a non-negative integer less than *b* (and f_1 must not be zero), *s* is +1 or -1, and *e* is an integer that lies between some integer maximum $e_{\max}$ and some integer minimum $e_{\min}$. The values of *b*, *p*, $e_{\min}$, and $e_{\max}$ determine the set of model floating point numbers. They are chosen to make the model fit as well as possible to the machine on which the program is executed. This model is independent of the actual bit pattern used to store floating point numbers on a particular processor.

The value *b* in this model is the **radix** or base of the numbering system used to represent real numbers on a particular computer. Essentially all modern computers use a base 2 numbering system, so *b* is 2, and each f_k must be either 0 or 1 (f_1 must be 1).

The bits that make up a real or floating-point number are divided into two separate fields, one for the mantissa (the fractional part of the number) and one for the exponent. For a base 2 system, *p* is the number of bits in the mantissa, and the value of *e* is stored in a field that is one less than the number of bits in the exponent[2]. Since the IEEE single precision standard devotes 24 bits to the mantissa and 8 bits to the exponent, *p* is 24, $e_{\max} = 2^7 = 127$, and $e_{\min} = -126$. For a typical 32-bit single precision real number on a base 2 computer, the model of the number becomes

$$x = \begin{cases} 0 & \text{or} \\ \pm 2^e \times \left(\frac{1}{2} + \sum_{k=2}^{24} f_k \times 2^{-k} \right) \end{cases}, \; -126 \le e \le 127 \qquad \text{(B-4)}$$

[2] It is one less than the number of bits in the exponent because one bit is reserved for the sign of the exponent.

The inquiry functions DIGITS, EPSILON, HUGE, MAXEXPONENT, MINEXPONENT, PRECISION, RANGE, RADIX, and TINY all return values related to the model parameters for the type and kind associated with the calling arguments. Of these functions, only PRECISION and RANGE matter to most programmers.

BIT_SIZE(I)

- Integer inquiry function
- Returns the number of bits in integer I.
- I must be Integer.

DIGITS(X)

- Integer inquiry function
- Returns the number of significant digits in X. (This function returns *q* from the integer model in Equation B-1, or *p* from the real model in Equation B-3).
- X must be Integer or Real.
- **Caution:** This function returns the number of significant digits in the base of the numbering system used on the computer. For most modern computers, this is base 2, so this function returns the number of significant bits. If you want the number of significant decimal digits, use PRECISION(X) instead.

EPSILON(X)

- Integer inquiry function of the same type as X.
- Returns a positive number that is almost negligible compared to 1.0 of the same type and kind as X. (The returned value is b^{1-p}, where *b* and *p* are defined in Equation B-3.)
- X must be Real.
- Essentially, EPSILON(X) is the number which, when added to 1.0, produces the next number representable by the given KIND of real number on a particular processor.

EXPONENT(X)

- Integer inquiry function of the same type as X.
  Returns the exponent of X in the base of the computer numbering system. (This is *e* from the real number model as defined in Equation B-3.)
- X must be Real.

FRACTION(X)

- Real elemental function of same kind as X.
- Returns the mantissa or the fractional part of the model representation of X. (This function returns the summation term from Equation B-3.)
- X must be Real.

HUGE(X)

- Integer inquiry function of the same type as X.
- Returns the largest number of the same type and kind as X

- X must be Integer or Real.

KIND(X)

- Integer inquiry function.
- Returns the kind value of X
- X may be any intrinsic type.

MAXEXPONENT(X)

- Integer inquiry function.
- Returns the maximum exponent of the same type and kind as X. (The returned value is $e_{\max}$ from the model in Equation B-3.)
- X must be Real.
- **Caution:** This function returns the maximum exponent in the base of the numbering system used on the computer. For most modern computers, this is base 2, so this function returns the maximum exponent as a base 2 number. If you want the maximum exponent as a decimal value, use RANGE(X) instead.

MINEXPONENT(X)

- Integer inquiry function.
- Returns the minimum exponent of the same type and kind as X. (The returned value is $e_{\min}$ from the model in Equation B-3.)
- X must be Real.

PRECISION(X)

- Integer inquiry function.
- Returns the number of significant *decimal digits* in values of the same type and kind as X.
- X must be Real or Complex.

RADIX(X)

- Integer inquiry function.
- Returns the base of the mathematical model for the type and kind of X. Since most modern computers work on a base 2 system, this number will almost always be 2. (This is *r* in Equation B-1, or *b* Equation B-3.)
- X must be Integer or Real.

RANGE(X)

- Integer inquiry function.
- Returns the *decimal* exponent range for values of the same type and kind as X.
- X must be Integer, Real, or Complex.

RRSPACING(X)

- Elemental function of the same type and kind as X.
- Returns the reciprocal of the relative spacing of the numbers near X. (The result has the value $\left|x \times b^{-e}\right| \times b^{p}$, where *b*, *e*, and *p* are defined as in Equation B-3.)

- X must be Real.

SCALE(X,I)

- Elemental function of the same type and kind as X.
- Returns the value $x \times b^I$, where b is the base of the model used to represent X. The base b can be found with the RADIX(X) function; it is almost always 2.
- X must be Real, and I must be Integer.

SELECTED_INT_KIND(R)

- Integer transformational function.
- Returns the kind number for the smallest integer kind which can represent all integers n whose values satisfy the condition ABS(n) < 10**R. If more than one kind satisfies this constraint, then the kind returned will be the one with the smallest decimal range. If no kind satisfies the requirement, the value -1 is returned.
- R must be Integer.

SELECTED_REAL_KIND(P,R)

- Integer transformational function.
- Returns the kind number for the smallest real kind which has a decimal precision of at least P digits and an exponent range of at least R powers of 10. If more than one kind satisfies this constraint, then the kind returned will be the one with the smallest decimal precision.
- If no real kind satisfies the requirement, a -1 is returned if the requested precision was not available, a -2 is returned if the requested range was not available, and a -3 is returned if neither was available.
- P and R must be Integers.

SET_EXPONENT(X,I)

- Elemental function of the same type as X.
- Returns the number whose fractional part is the fractional part of the number X, and whose exponent part is I. If X = 0, then the result is 0.
- X is Real, and I is Integer.

SPACING(X)

- Elemental function of the same type and kind as X.
- Returns the absolute spacing of the numbers near X in the model used to represent real numbers. If the absolute spacing is out of range, then this function returns the same value as TINY(X). (This function returns the value b^{e-p}, where b, e, and p are as defined in Equation B-3, as long as that value is in range.)
- X must be Real.
- The result of this function is useful for establishing convergence criteria in a processor-independent manner. For example, we might conclude that a root-solving algorithm has converged when the answer gets within 10 times the minimum representable spacing.

`TINY(X)`

- Elemental function of the same type and kind as `X`.
- Returns the smallest positive number of the same type and kind as `X`. (The returned value is $b^{e_{\min}-1}$, where b and $e_{\min}$ are as defined in Equation B-3.)
- `X` must be Real.

B.5. Date and Time Intrinsic Subroutines

`CPU_TIME(TIME)`

- Intrinsic subroutine
- Returns processor time expended on current program in seconds.
- Arguments:

Keyword	Type	Intent	Description
`TIME`	Real	`OUT`	Processor time.

- The purpose of this subroutine is to time sections of code by comparing the processor time before and after the code is executed.
- The definition of the time returned by this subroutine is processor dependent. On most processors, it is the CPU time spent executing the current program.
- On computers with multiple CPUs, `TIME` may be implemented as an array containing the times associated with each processor.
- Fortran 95 only.

`DATE_AND_TIME(`*`DATE`*`,`*`TIME`*`,`*`ZONE`*`,`*`VALUE`*`)`

- Intrinsic subroutine
- Returns date and time.
- All arguments are optional, but at least one must be included:

Keyword	Type	Intent	Description
`DATE`	Character(8)	`OUT`	Returns a string in the form CCYYMMDD, where CC is century, YY is year, MM is month, and DD is day.
`TIME`	Character(10)	`OUT`	Returns a string in the form HHMMSS.SSS, where HH is hour, MM is minute, SS is second, and SSS is millisecond.
`ZONE`	Character(5)	`OUT`	Returns a string in the form ±HHMM, where HHMM is the time difference between local time and Coordinated Universal Time (UCT, or GMT).
`VALUES`	Integer(8)	`OUT`	See table below for values.

- If a value is not available for *`DATE`*, *`TIME`*, or *`ZONE`*, then the string blanked.
- The information returned in array *`VALUES`* is:

VALUES(1)	Century and year (for example, 1996)
VALUES(2)	Month (1-12)
VALUES(3)	Day (1-31)
VALUES(4)	Time zone difference from UTC in minutes.
VALUES(5)	Hour (0-23)
VALUES(6)	Minutes (0-59)
VALUES(7)	Seconds (0-60)
VALUES(8)	Milliseconds (0-999)

- If no information is available for one of the elements of array *VALUES*, that element is set to the most negative representable integer (-HUGE(0)).
- Note that the seconds field ranges from 0 to 60. The extra second is included to allow for leap-seconds.

SYSTEM_CLOCK(COUNT,COUNT_RATE,COUNT_MAX)

- Intrinsic subroutine.
- Returns raw counts from the processor's real-time clock. The value in COUNT is increased by one for each clock count until COUNT_MAX is reached. When COUNT_MAX is reached, the value in COUNT is reset to 0 on the next clock count. Variable COUNT_RATE specifies the number of real-time clock counts per second, so it tells how to interpret the count information.
- Arguments:

Keyword	Type	Intent	Description
COUNT	Integer	OUT	Number of counts of the system clock. The starting count is arbitrary.
COUNT_RATE	Integer	OUT	Number of clock counts per second.
COUNT_MAX	Integer	OUT	The maximum value for COUNT.

- If there is no clock, COUNT and COUNT_RATE are set to -HUGE(0) and COUNT_MAX is set to 0.

B.6. Bit Intrinsic Procedures

The layout of bits within an integer varies from processor to processor. For example, some processors place the most significant bit of a value at the bottom of the memory representing that value, while other processors place the least significant bit of a value at the top of the memory representing that value. To insulate programmers from these machine dependencies, Fortran defines a bit to be a binary digit *w* located at position *k* of a non-negative integer based on a model non-negative integer defined by

$$j = \sum_{k=0}^{z-1} w_k \times 2^k \qquad \text{(B-5)}$$

where w_k can be either 0 or 1. Thus bit 0 is the coefficient of 2^0, bit 1 is the coefficient of 2^1, etc. In this model, *z* is the number of bits in the integer, and the bits are numbered 0, 1, ...,

z-1, regardless of the physical layout of the integer. The least significant bit is considered to be at the right of the model and the most significant bit is considered to be at the left of the model, regardless of the actual physical implementation. Thus, shifting a bit left increases its value, and shifting a bit right decreases its value.

Fortran 90/95 includes 10 elemental functions and 1 elemental subroutine that manipulate bits according to this model. Logical operations on bits are performed by the elemental functions IOR, IAND, NOT, and IEOR. Shift operations are performed by the elemental functions ISHFT and ISHFTC. Bit sub-fields may be referenced by the elemental function IBITS and the elemental subroutine MVBITS. Finally, single-bit processing is performed by the elemental functions BTEST, IBSET, and IBCLR.

BTEST(I,POS)

- Logical elemental function
- Returns true if bit POS of I is 1, and false otherwise.
- I and POS must be Integers, with 0 ≤ POS < BIT_SIZE(I)

IAND(I,J)

- Elemental function of the same type and kind as I
- Returns the bit by bit logical AND of I and J.
- I and J must be Integers of the same kind

IBCLR(I,POS)

- Elemental function of the same type and kind as I
- Returns I with bit POS set to 0.
- I and POS must be Integers, with 0 ≤ POS < BIT_SIZE(I)

IBITS(I,POS,LEN)

- Elemental function of the same type and kind as I
- Returns a right-adjusted sequence of bits extracted from I of length LEN starting at bit POS. All other bits are zero.
- I , POS, and LEN must be Integers, with POS + LEN < BIT_SIZE(I)

IBSET(I,POS)

- Elemental function of the same type and kind as I
- Returns I with bit POS set to 1.
- I and POS must be Integers, with 0 ≤ POS < BIT_SIZE(I)

IEOR(I,J)

- Elemental function of the same type and kind as I
- Returns the bit by bit exclusive OR of I and J.
- I and J must be Integers of the same kind

IOR(I,J)

- Elemental function of the same type and kind as I
- Returns the bit by bit inclusive OR of I and J.

- I and J must be Integers of the same kind

`ISHFT(I,SHIFT)`

- Elemental function of the same type and kind as I
- Returns I logically shifted to the left (if SHIFT is positive) or right (if SHIFT is negative). The empty bits are filled with zeros.
- I must be an Integer
- SHIFT must be an Integer, with ABS(SHIFT) <= BIT_SIZE(I)
- A shift to the left implies moving the bit in position *i* to position *i*+1, and a shift to the right implies moving the bit in position *i* to position *i*-1.

`ISHFTC(I,SHIFT,SIZE)`

- Elemental function of the same type and kind as I
- Returns the value obtained by shifting the *SIZE* rightmost bits of I circularly by SHIFT bits. If SHIFT is positive, the bits are shifted left, and if SHIFT is negative, the bits are shifted right. If the optional argument *SIZE* is missing, all BIT_SIZE(I) bits of I are shifted.
- I must be an Integer
- SHIFT must be an Integer, with ABS(SHIFT) <= *SIZE*
- *SIZE* must be a positive integer, with 0 < *SIZE* <= BIT_SIZE(I)

`MVBITS(FROM,FROMPOS,LEN,TO,TOPOS)`

- Elemental subroutine
- Copies a sequence of bits from integer FROM to integer TO. The subroutine copies a sequence of LEN bits starting at FROMPOS in integer FROM, and stores them starting at TOPOS in integer TO. All other bits in integer TO are undisturbed.
- Note that FROM and TO can be the same integer.
- Arguments:

Keyword	Type	Intent	Description
FROM	Integer	IN	The object from which the bits are to be moved.
FROMPOS	Integer	IN	Starting bit to move; must be ≥ 0
LEN	Integer	IN	Number of bits to move; FROMPOS+LEN must be ≤ BIT_SIZE(FROM)
TO	Integer, same kind as FROM	INOUT	Destination object.
TOPOS	Integer	IN	Starting bit in destination; 0 ≤ TOPOS+LEN ≤ BIT_SIZE(TO)

`NOT(I)`

- Elemental function of the same type and kind as I
- Returns the logical complement of the bits in I.
- I must be Integer

B.7. Character Intrinsic Functions

These functions produce, manipulate, or provide information about character strings.

`ACHAR(I)`

- Character(1) elemental function
- Returns the character in position `I` of the ASCII collating sequence.
- If $0 \le I \le 127$, the result is the character in position `I` of the ASCII collating sequence. If $I \ge 128$, the results are processor-dependent.
- `I` must be Integer
- `IACHAR` is the inverse function of `ACHAR`

`ADJUSTL(STRING)`

- Character elemental function
- Returns a character value of the same length as `STRING`, with the non-blank contents left justified. That is, the leading blanks of `STRING` are removed and the same number of trailing blanks are added at the end.
- `STRING` must be Character

`ADJUSTR(STRING)`

- Character elemental function
- Returns a character value of the same length as `STRING`, with the non-blank contents right justified. That is, the trailing blanks of `STRING` are removed and the same number of leading blanks are added at the beginning.
- `STRING` must be Character

`CHAR(I,`*`KIND`*`)`

- Character(1) elemental function
- Returns the character in position `I` of the processor collating sequence associated with the specified kind.
- `I` must be an Integer in the range $0 \le I \le n\text{-}1$, where n is the number of characters in the processor-dependent collating sequence.
- *`KIND`* must be an Integer whose value is a legal kind of character for the particular computer; if it is absent, the default kind of character is assumed.
- `ICHAR` is the inverse function of `CHAR`

`IACHAR(C)`

- Integer elemental function
- Returns the position of a character in the ASCII collating sequence. A processor-dependent value is returned if `C` is not in the collating sequence.
- `C` must be Character(1)
- `ACHAR` is the inverse function of `IACHAR`

`ICHAR(C)`

- Integer elemental function

- Returns the position of a character in the processor collating sequence associated with the kind of the character.
- C must be Character(1)
- The result is in the range $0 \le$ ICHAR(C) $\le n$-1, where n is the number of characters in the processor-dependent collating sequence.
- CHAR is the inverse function of ICHAR

INDEX(STRING,SUBSTRING,*BACK*)

- Integer elemental function
- Returns the starting position of a substring within a string.
- STRING and SUBSTRING must be Character values of the same kind, and *BACK* must be Logical.
- If the substring is longer than the string, the result is 0. If the length of the substring is 0, then the result is 1. Otherwise, if *BACK* is missing or false, the function returns the starting position of the *first* occurrence of the substring within the string, searching from left to right through the string. If *BACK* is true, the function returns the starting position of the *last* occurrence of the substring within the string.

LEN(STRING)

- Integer inquiry function
- Returns the length of STRING in characters.
- STRING must be Character

LEN_TRIM(STRING)

- Integer inquiry function
- Returns the length of STRING in characters, less any trailing blanks. If STRING is completely blank, then the result is 0.
- STRING must be Character

LGE(STRING_A,STRING_B)

- Logical elemental function
- Returns true if STRING_A $\ge$ STRING_B in the ASCII collating sequence.
- STRING_A and STRING_B must be of type default Character
- The comparison process is similar to that used by the >= relational operator, except that the comparison always uses the ASCII collating sequence.

LGT(STRING_A,STRING_B)

- Logical elemental function
- Returns true if STRING_A > STRING_B in the ASCII collating sequence.
- STRING_A and STRING_B must be of type default Character
- The comparison process is similar to that used by the > relational operator, except that the comparison always uses the ASCII collating sequence.

`LLE(STRING_A,STRING_B)`

- Logical elemental function
- Returns true if `STRING_A` ≤ `STRING_B` in the ASCII collating sequence.
- `STRING_A` and `STRING_B` must be of type default Character
- The comparison process is similar to that used by the `<=` relational operator, except that the comparison always uses the ASCII collating sequence.

`LLT(STRING_A,STRING_B)`

- Logical elemental function
- Returns true if `STRING_A` < `STRING_B` in the ASCII collating sequence.
- `STRING_A` and `STRING_B` must be of type default Character
- The comparison process is similar to that used by the `<` relational operator, except that the comparison always uses the ASCII collating sequence.

`REPEAT(STRING,NCOPIES)`

- Character transformational function.
- Returns a character string formed by concatenating `NCOPIES` copies of `STRING` one after another. If `STRING` is zero length or if `NCOPIES` is 0, the function returns a zero length string.
- `STRING` must be of type Character; `NCOPIES` must be a non-negative Integer.

`SCAN(STRING,SET,`*`BACK`*`)`

- Integer elemental function.
- Scans `STRING` for the first occurrence of any one of the characters in `SET`, and returns the position of that occurrence. If no character of `STRING` is in set, or if either `STRING` or `SET` is zero length, the function returns a zero.
- `STRING` and `SET` must be of type Character and the same kind, and *`BACK`* must be of type Logical.
- If *`BACK`* is missing or false, the function returns the position of the *first* occurrence (searching left to right) of any of the characters contained in `SET`. If *`BACK`* is true, the function returns the position of the *last* occurrence (searching right to left) of any of the characters contained in `SET`.

`TRIM(STRING)`

- Character transformational function.
- Returns `STRING` with trailing blanks removed. If `STRING` is completely blank, then a zero length string is returned.
- `STRING` must be of type Character.

`VERIFY(STRING,SET,`*`BACK`*`)`

- Integer elemental function.
- Scans `STRING` for the first occurrence of any one of the characters *not* in `SET`, and returns the position of that occurrence. If all characters of `STRING` are in `SET`, or if either `STRING` or `SET` is zero length, the function returns a zero.

- STRING and SET must be of type Character and the same kind, and *BACK* must be of type Logical.
- If *BACK* is missing or false, the function returns the position of the *first* occurrence (searching left to right) of any of the characters not contained in SET. If *BACK* is true, the function returns the position of the *last* occurrence (searching right to left) of any of the characters not in SET.

B.8. Array and Pointer Intrinsic Functions

This section describes the 24 standard array and pointer intrinsic functions. Because certain arguments appear in many of these functions, they will be described in detail before we examine the functions themselves.

1. The rank of an array is defined as the number of dimensions in the array. It is abbreviated as *r* throughout this section.
2. A scalar is defined to be an array of rank 0.
3. The optional argument *MASK* is used by some functions to select the elements of another argument to operate on. When present, *MASK* must be a logical array of the same size and shape as the target array; if an element of *MASK* is true, then the corresponding element of the target array will be operated on.
4. The optional argument *DIM* is used by some functions to determine the dimension of an array along which to operate. When supplied, *DIM* must be a number in the range $1 \le \mathrm{DIM} \le r$.
5. In the functions ALL, ANY, LBOUND, MAXVAL, MINVAL, PRODUCT, SUM, and UBOUND, the optional argument *DIM* affects the type of argument returned by the function. If the argument is absent, then the function returns a scalar result. If the argument is present, then the function returns a vector result. Because the presence or absence of *DIM* affects the type of value returned by the function, the compiler must be able to determine whether or not the argument is present when the program is compiled. Therefore, *the actual argument corresponding to DIM must not be a optional dummy argument in the calling program unit*. If it were, the compiler would be unable to determine whether or not DIM is present at compilation time. This restriction does not apply to functions CSHIFT, EOSHIFT, SIZE, and SPREAD, since the argument *DIM* does not affect the type of value returned from these functions.

To illustrate the use of *MASK* and *DIM*, let's apply the function MAXVAL to a 2×3 real array array1 ($r = 2$) and two masking arrays mask1 and mask2 defined as follows:

$$\mathtt{array1} = \begin{bmatrix} 1. & 2. & 3. \\ 4. & 5. & 6. \end{bmatrix}$$

$$\mathtt{mask1} = \begin{bmatrix} .\mathrm{TRUE}. & .\mathrm{TRUE}. & .\mathrm{TRUE}. \\ .\mathrm{TRUE}. & .\mathrm{TRUE}. & .\mathrm{TRUE}. \end{bmatrix}$$

$$\texttt{mask2} = \begin{bmatrix} .\text{TRUE.} & .\text{TRUE.} & .\text{FALSE.} \\ .\text{TRUE.} & .\text{TRUE.} & .\text{FALSE.} \end{bmatrix}$$

The function MAXVAL returns the maximum value(s) along the dimension *DIM* of an array corresponding to the true elements of *MASK*. It has the calling sequence

```
result = MAXVAL(ARRAY,DIM,MASK)
```

If *DIM* is not present, the function returns a scalar equal to the largest value in the array for which *MASK* is true. Therefore, the function

```
result = MAXVAL(array1,MASK=mask1)
```

will produce a value of 6, while the function

```
result = MAXVAL(array1,MASK=mask2)
```

will produce a value of 5. If *DIM* is present, then the function will return an array of rank *r*-1 containing the maximum values along dimension *DIM* for which *MASK* is true. That is, the function will hold the subscript in the specified dimension constant while searching along all other dimensions to find the masked maximum value in that sub-array, and then repeat the process for every other possible value of the specified dimension. Since there are three elements in each row of the array, the function

```
result = MAXVAL(array1,DIM=1,MASK=mask1)
```

will search along the *columns* of the array at each row position, and will produce the vector [4. 5. 6.], where 4. was the maximum value in column 1, 5. was the maximum value in column 2, and 6. was the maximum value in column 3. Similarly, there are two elements in each column of the array, so the function

```
result = MAXVAL(array1,DIM=2,MASK=mask1)
```

will search along the *rows* of the array at each column position, and will produce the vector [3. 6.], where 3. was the maximum value in row 1, and 6. was the maximum value in row 2.

ALL(MASK,*DIM*)

- Logical transformational function
- Returns true if all MASK values are true along dimension *DIM*, or if MASK has zero size. Otherwise, it returns false.
- MASK is a Logical array.
  DIM is an integer in the range $1 \le DIM \le r$. The corresponding actual argument must not be an optional argument in the calling procedure.
- The result is a scalar if *DIM* is absent. It is an array of rank *r*-1 and shape (d(1),d(2),…,d(DIM-1),d(DIM+1),…,d(r)) where the shape of MASK is (d(1),d(2),…,d(r)). In other words, the shape of the returned vector is the same as the shape of the original mask with dimension *DIM* deleted.

ANY(MASK,*DIM*)

- Logical transformational function
- Returns true if any MASK value is true along dimension *DIM*. Otherwise, it returns false. If MASK has zero size, it returns false.
- MASK is a Logical array.
  DIM is an integer in the range $1 \le DIM \le r$. The corresponding actual argument must not be an optional argument in the calling procedure.
- The result is a scalar if *DIM* is absent. It is an array of rank *r*-1 and shape (d(1),d(2),…,d(DIM-1),d(DIM+1),…,d(r)) where the shape of MASK is (d(1),d(2),…,d(r)). In other words, the shape of the returned vector is the same as the shape of the original mask with dimension *DIM* deleted.

COUNT(MASK,*DIM*)

- Logical transformational function
- Returns the number of true elements of MASK along dimension *DIM*, and returns 0 if MASK has zero size.
- MASK is a Logical array.
  DIM is an integer in the range $1 \le DIM \le r$. The corresponding actual argument must not be an optional argument in the calling procedure.
- The result is a scalar if *DIM* is absent. It is an array of rank *r*-1 and shape (d(1),d(2),…,d(DIM-1),d(DIM+1),…,d(r)) where the shape of MASK is (d(1),d(2),…,d(r)). In other words, the shape of the returned vector is the same as the shape of the original mask with dimension *DIM* deleted.

CSHIFT(ARRAY,SHIFT,*DIM*)

- Transformational function of the same type as ARRAY.
- Performs a circular shift on an array expression of rank-1, or performs circular shifts on all the complete rank-1 sections along a given dimension of an array expression of rank-2 or greater. Elements shifted out at one end of a section are shifted in at the other end. Different sections may be shifted by different amounts and in different directions.
- ARRAY may be an array of any type and rank, but not a scalar.
  SHIFT is a scalar if ARRAY is rank 1. Otherwise, it is an array of rank *r*-1 and of shape (d(1),d(2),…,d(DIM-1),d(DIM+1),…,d(r)) where the shape of ARRAY is (d(1),d(2),…,d(r)).
  DIM is an optional integer in the range $1 \le DIM \le r$. If *DIM* is missing, the function behaves as though *DIM* were present and equal to 1.

EOSHIFT(ARRAY,SHIFT,*DIM*)

- Transformational function of the same type as ARRAY.
- Performs an end-off shift on an array expression of rank-1, or performs end-off shifts on all the complete rank-one sections along a given dimension of an array expression of rank-2 or greater. Elements are shifted off at one end of a section and copies of a boundary value are shifted in at the other end. Different sections

may have different boundary values and may be shifted by different amounts and in different directions.

- ARRAY may be an array of any type and rank, but not a scalar.
  SHIFT is a scalar if ARRAY is rank 1. Otherwise, it is an array of rank *r*-1 and of shape (d(1),d(2),…,d(DIM-1),d(DIM+1),…,d(r)) where the shape of ARRAY is (d(1),d(2),…,d(r)).
  DIM is an optional integer in the range $1 \le DIM \le r$. If *DIM* is missing, the function behaves as though *DIM* were present and equal to 1.

LBOUND(ARRAY,*DIM*)

- Integer inquiry function.
- Returns all of the lower bounds or a specified lower bound of ARRAY.
- ARRAY is an array of any type. It must not be an unassociated pointer or an unallocated allocatable array.
  DIM is an integer in the range $1 \le DIM \le r$. The corresponding actual argument must not be an optional argument in the calling procedure.
- If *DIM* is present, the result is a scalar. If the actual argument corresponding to ARRAY is an array section or an array expression, or if dimension *DIM* has zero size, then the function will return 1. Otherwise, it will return the lower bound of that dimension of ARRAY. If *DIM* is not present, then the function will return an array whose *i*th element is LBOUND(ARRAY,*i*) for *i*=1,2,…,*r*.

MAXLOC(ARRAY,*DIM*,*MASK*)

- Integer transformational function, returning a rank-1 array of size *r*.
- Returns the location of the maximum value of the elements in ARRAY along dimension *DIM* (if present) corresponding to the true elements of *MASK* (if present). If more than one element has the same maximum value, the location of the first one found is returned.
- ARRAY is an array of type Integer or Real.
  DIM is an integer in the range $1 \le DIM \le r$. The corresponding actual argument must not be an optional argument in the calling procedure.
  MASK is a logical scalar or a logical array conformable with ARRAY.
- If *DIM* is not present and *MASK* is not present, the result is a rank-1 array containing the subscripts of the first element found in ARRAY having the maximum value. If *DIM* is not present and *MASK* is present, the search is restricted to those elements for which *MASK* is true. If *DIM* is present, the result is an array of rank *r*-1 and of shape (d(1),d(2),…,d(DIM-1), d(DIM+1),…,d(r)) where the shape of ARRAY is (d(1),d(2),…,d(r)). This array contains the subscripts of the largest values found along dimension *DIM*.
- The optional argument *DIM* is only present in Fortran 95.
- For example, if

$$\text{ARRAY} = \begin{bmatrix} 1 & 3 & -9 \\ 2 & 2 & 6 \end{bmatrix} \text{ and } \text{MASK} = \begin{bmatrix} \text{TRUE} & \text{FALSE} & \text{FALSE} \\ \text{TRUE} & \text{TRUE} & \text{FALSE} \end{bmatrix}$$

then the result of the function MAXLOC(ARRAY) is (/2,3/). The result of MAXLOC(ARRAY,MASK) is (/2,1/). The result of MAXLOC(ARRAY,DIM=1) is (/2,1,2/), and the result of MAXLOC(ARRAY,DIM=2) is (/2,3/).

MAXVAL(ARRAY,*DIM*,*MASK*)

- Transformational function of the same type as ARRAY.
- Returns the maximum value of the elements in ARRAY along dimension *DIM* (if present) corresponding to the true elements of *MASK* (if present). If ARRAY has zero size, or if all the elements of *MASK* are false, then the result is the largest possible negative number of the same type and kind as ARRAY.
- ARRAY is an array of type Integer or Real.
  DIM is an integer in the range $1 \le DIM \le r$. The corresponding actual argument must not be an optional argument in the calling procedure.
  MASK is a logical scalar or a logical array conformable with ARRAY.
- If *DIM* is not present, the result is a scalar containing the maximum value found in the elements of ARRAY corresponding to true elements of *MASK*. If *MASK* is absent, the search is over all of the elements in ARRAY. If *DIM* is present, the result is an array of rank *r*-1 and of shape (d(1),d(2),…,d(DIM-1), d(DIM+1),…,d(r)) where the shape of ARRAY is (d(1),d(2),…,d(r)).
- For example, if
  $$\text{ARRAY} = \begin{bmatrix} 1 & 3 & -9 \\ 2 & 2 & 6 \end{bmatrix} \text{ and } \text{MASK} = \begin{bmatrix} \text{TRUE} & \text{FALSE} & \text{FALSE} \\ \text{TRUE} & \text{TRUE} & \text{FALSE} \end{bmatrix}$$
  then the result of the function MAXVAL(ARRAY) is 6. The result of MAXVAL(ARRAY,MASK) is 2. The result of MAXVAL(ARRAY,DIM=1) is (/2,3,6/), and the result of MAXLOC(ARRAY,DIM=2) is (/3,6/).

MERGE(TSOURCE,FSOURCE,MASK)

- Elemental function of the same type as TSOURCE.
- Selects one of two alternative values according to MASK. If a given element of MASK is true, then the corresponding element of the result comes from array TSOURCE. If a given element of MASK is false, then the corresponding element of the result comes from array FSOURCE. MASK may also be a scalar, in which case either all of TSOURCE or all of FSOURCE is selected.
- TSOURCE is any type of array; FSOURCE is the same type and kind as TSOURCE.
  MASK is a logical scalar, or a logical array conformable with TSOURCE.

MINLOC(ARRAY,*DIM*,*MASK*)

- Integer transformational function, returning a rank-1 array of size *r*.
- Returns the *location* of the minimum value of the elements in ARRAY along dimension *DIM* (if present) corresponding to the true elements of *MASK* (if present). If more than one element has the same minimum value, the location of the first one found is returned.
- ARRAY is an array of type Integer or Real.

DIM is an integer in the range $1 \le DIM \le r$. The corresponding actual argument must not be an optional argument in the calling procedure.
MASK is a logical scalar, or a logical array conformable with ARRAY.

- If *DIM* is not present and *MASK* is not present, the result is a rank-1 array containing the subscripts of the first element found in ARRAY having the minimum value. If *DIM* is not present and *MASK* is present, the search is restricted to those elements for which *MASK* is true. If *DIM* is present, the result is an array of rank *r*-1 and of shape (d(1),d(2),…,d(DIM-1), d(DIM+1),…,d(r)) where the shape of ARRAY is (d(1),d(2),…,d(r)). This array contains the subscripts of the smallest values found along dimension *DIM*.
- The optional argument *DIM* is only present in Fortran 95.
- For example, if

$$\text{ARRAY} = \begin{bmatrix} 1 & 3 & -9 \\ 2 & 2 & 6 \end{bmatrix} \text{ and } \text{MASK} = \begin{bmatrix} \text{TRUE} & \text{FALSE} & \text{FALSE} \\ \text{TRUE} & \text{TRUE} & \text{FALSE} \end{bmatrix}$$

then the result of the function MINLOC(ARRAY) is (/1,3/). The result of MINLOC(ARRAY,MASK) is (/1,1/). The result of MINLOC(ARRAY,DIM=1) is (/1,2,1/), and the result of MINLOC(ARRAY,DIM=2) is (/3,1/).

MINVAL(ARRAY,*DIM*,*MASK*)

- Transformational function of the same type as ARRAY.
- Returns the minimum value of the elements in ARRAY along dimension *DIM* (if present) corresponding to the true elements of *MASK* (if present). If ARRAY has zero size, or if all the elements of *MASK* are false, then the result is the largest possible positive number of the same type and kind as ARRAY.
- ARRAY is an array of type Integer or Real.
  DIM is an integer in the range $1 \le DIM \le r$. The corresponding actual argument must not be an optional argument in the calling procedure.
  MASK is a logical scalar, or a logical array conformable with ARRAY.
- If *DIM* is not present, the result is a scalar containing the minimum value found in the elements of ARRAY corresponding to true elements of *MASK*. If *MASK* is absent, the search is over all of the elements in ARRAY. If *DIM* is present, the result is an array of rank *r*-1 and of shape (d(1),d(2),…,d(DIM-1), d(DIM+1),…,d(r)) where the shape of ARRAY is (d(1),d(2),…,d(r)).
- For example, if

$$\text{ARRAY} = \begin{bmatrix} 1 & 3 & -9 \\ 2 & 2 & 6 \end{bmatrix} \text{ and } \text{MASK} = \begin{bmatrix} \text{TRUE} & \text{FALSE} & \text{FALSE} \\ \text{TRUE} & \text{TRUE} & \text{FALSE} \end{bmatrix}$$

then the result of the function MINVAL(ARRAY) is -9. The result of MINVAL(ARRAY,MASK) is 1. The result of MINVAL(ARRAY,DIM=1) is (/1,2,-9/), and the result of MINLOC(ARRAY,DIM=2) is (/-9,2/).

NULL(*MOLD*)

- Transformational function.
- Returns a disassociated pointer of the same type as *MOLD*, if present. If *MOLD* is not present, the pointer type is determined by context. (For example, if NULL() is being used to initialize an Integer pointer, the returned value will be a disassociated Integer pointer.)
- *MOLD* is a pointer of any type. Its pointer association status may be undefined, disassociated, or associated.
- This function is useful for initializing the status of a pointer at the time it is declared. It is only available in Fortran 95.

PACK(ARRAY,MASK,*VECTOR*)

- Transformational function of the same type as ARRAY.
- Packs an array into an array of rank-1 under the control of a mask.
- ARRAY is an array of any type.
  MASK is a logical scalar, or a logical array conformable with ARRAY.
  VECTOR is a rank-1 array of the same type as ARRAY. It must have at least as many elements as there are true values in the mask. If MASK is a true scalar with the value true, then it must have at least as many elements as there are in ARRAY.
- This function packs the elements of ARRAY into an array of rank-1 under the control of MASK. An element of ARRAY will be packed into the output vector if the corresponding element of MASK is true. If MASK is a true scalar value, then the entire input array will be packed into the output array. The packing is done in column order.
- If argument *VECTOR* is present, then the length of the function output will be the length of *VECTOR*. This length must be greater than or equal to the number of elements to be packed.
- For example, if

$$\text{ARRAY} = \begin{bmatrix} 1 & -3 \\ 4 & -2 \end{bmatrix} \text{ and } \qquad \text{MASK} = \begin{bmatrix} \text{False} & \text{True} \\ \text{True} & \text{True} \end{bmatrix}$$

  then the result of the function PACK(ARRAY,MASK) will be [4 -3 -2].

PRODUCT(ARRAY,*DIM*,*MASK*)

- Transformational function of the same type as ARRAY.
- Returns the product of the elements in ARRAY along dimension *DIM* (if present) corresponding to the true elements of *MASK* (if present). If ARRAY has zero size, or if all the elements of *MASK* are false, then the result has the value one.
- ARRAY is an array of type Integer, Real, or Complex.
  DIM is an integer in the range $1 \le DIM \le r$. The corresponding actual argument must not be an optional argument in the calling procedure.
  MASK is a logical scalar or a logical array conformable with ARRAY.
- If *DIM* is not present or if ARRAY has rank-1, the result is a scalar containing the product of all the elements of ARRAY corresponding to true elements of *MASK*. If

MASK is also absent, the result is the product of all of the elements in ARRAY. If *DIM* is present, the result is an array of rank *r*-1 and of shape (d(1),d(2),…,d(DIM-1),d(DIM+1),…,d(r)) where the shape of ARRAY is (d(1),d(2),…,d(r)).

RESHAPE(SOURCE,SHAPE,*PAD*,*ORDER*)

- Transformational function of the same type as SOURCE.
- Constructs an array of a specified shape from the elements of another array.
- SOURCE is an array of any type.
  SHAPE is a one to seven element Integer array containing the desired extent of each dimension of the output array.
  PAD is a rank-1 array of the same type as SOURCE. It contains elements to be used as a pad on the end if the output array if there are not enough elements in SOURCE.
  ORDER is an Integer array of the same shape as SHAPE. It specifies the order in which dimensions are to be filled with elements from SOURCE.
- The result of this function is an array of shape SHAPE constructed from the elements of SOURCE. If SOURCE does not contain enough elements, the elements of *PAD* are used repeatedly to fill out the remainder of the output array. *ORDER* specifies the order in which the dimensions of the output array will be filled; by default they fill in the order (1, 2, ..., *n*) where *n* is the size of SHAPE.
- For example, if $\text{SOURCE} = [1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6]$, $\text{SHAPE} = [2 \quad 5]$, and $\text{PAD} = [0 \quad 0]$, then

$$\text{RESHAPE(SOURCE,SHAPE,PAD)} = \begin{bmatrix} 1 & 3 & 5 & 0 & 0 \\ 2 & 4 & 6 & 0 & 0 \end{bmatrix}$$

and

$$\text{RESHAPE(SOURCE,SHAPE,PAD,(/2,1/))} = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 0 & 0 & 0 & 0 \end{bmatrix}$$

SHAPE(SOURCE)

- Integer inquiry function.
- Returns the shape of SOURCE as a rank-1 array whose size is *r* and whose elements are the extents of the corresponding dimensions of SOURCE. If SOURCE is a scalar, a rank-1 array of size zero is returned.
- SOURCE is an array or scalar of any type. It must not be an unassociated pointer or an unallocated allocatable array.

SIZE(ARRAY,*DIM*)

- Integer inquiry function.
- Returns either the extent of ARRAY along a particular dimension if *DIM* is present; otherwise, it returns the total number of elements in the array.
- ARRAY is an array of any type. It must not be an unassociated pointer or an unallocated allocatable array.

DIM is an integer in the range $1 \le DIM \le r$. If ARRAY is an assumed size array, *DIM* must be present, and must have a value less than *r*.

SPREAD(SOURCE,DIM,NCOPIES)

- Transformational function of the same type as SOURCE.
- Constructs an array of rank *r*+1 by copying SOURCE along a specified dimension (as in forming a book from copies of a single page).
- SOURCE is an array or scalar of any type. The rank of SOURCE must be less than 7.
  DIM is an Integer specifying the dimension over which to copy SOURCE. It must satisfy the condition $1 \le DIM \le r+1$.
  NCOPIES is the number of copies of SOURCE to make along dimension DIM. If NCOPIES is less than or equal to zero, a zero-sized array is produced.
- If SOURCE is a scalar, each element in the result has a value equal to SOURCE. If source is an array, the element in the result with subscripts $(s_1, s_2,...,s_{n+1})$ has the value SOURCE$(s_1, s_2,...,s_{DIM-1}, s_{DIM+1},...,s_{n+1})$.
- For example, if $\text{SOURCE} = \begin{bmatrix} 1 & 3 & 5 \end{bmatrix}$, then the result of function SPREAD(SOURCE,DIM=1,NCOPIES=3) is the array $\begin{bmatrix} 1 & 3 & 5 \\ 1 & 3 & 5 \\ 1 & 3 & 5 \end{bmatrix}$.

SUM(ARRAY,*DIM*,*MASK*)

- Transformational function of the same type as ARRAY.
- Returns the sum of the elements in ARRAY along dimension *DIM* (if present) corresponding to the true elements of *MASK* (if present). If ARRAY has zero size, or if all the elements of *MASK* are false, then the result has the value zero.
- ARRAY is an array of type Integer, Real, or Complex.
  DIM is an integer in the range $1 \le DIM \le r$. The corresponding actual argument must not be an optional argument in the calling procedure.
  MASK is a logical scalar or a logical array conformable with ARRAY.
- If *DIM* is not present or if ARRAY has rank-1, the result is a scalar containing the sum of all the elements of ARRAY corresponding to true elements of *MASK*. If *MASK* is also absent, the result is the sum of all of the elements in ARRAY. If *DIM* is present, the result is an array of rank *r*-1 and of shape (d(1),d(2),…,d(DIM-1),d(DIM+1),…,d(r)) where the shape of ARRAY is (d(1),d(2),…,d(r)).

TRANSFER(SOURCE,MOLD,*SIZE*)

- Transformational function of the same type as MOLD.
- Returns either a scalar or a rank-one array with a physical representation identical to that of SOURCE, but interpreted with the type and kind of MOLD. Effectively, this function takes the bit patterns in SOURCE and interprets them as though they were of the type and kind of MOLD.
- SOURCE is an array or scalar of any type.

MOLD is an array or scalar of any type.
SIZE is an scalar integer value. The corresponding actual argument must not be an optional argument in the calling procedure.

- If MOLD is a scalar and *SIZE* is absent, the result is a scalar. If MOLD is an array and *SIZE* is absent, the result has the smallest possible size which makes use of all of the bits in SOURCE. If *SIZE* is present, the result is a rank-1 array of length *SIZE*. If the number of bits in the result and in SOURCE are not the same, then bits will be truncated or extra bits will be added in an undefined, processor-dependent manner.
- Example 1: TRANSFER(4.0,0) has the integer value 1082130432 on a PC using IEEE Standard floating point numbers, because the bit representations of a floating point 4.0 and an integer 1082130432 are identical. The transfer function has caused the bit associated with the floating point 4.0 to be reinterpreted as an integer.
- Example 2: In the function TRANSFER((/1.1,2.2,3.3/),(/(0.,0.)/)), the SOURCE is three real values long. The MOLD is a rank-1 array containing a complex number, which is two real values long. Therefore, the output will be a Complex rank-1 array. In order to use all of the bits in SOURCE, the result of the function is a Complex rank-1 array with two elements. The first element in the output array is (1.1,2.2), and the second element has a real part of 3.3 together with an unknown imaginary part.
- Example 3: In the function TRANSFER((/1.1,2.2,3.3/),(/(0.,0.)/),1), the SOURCE is three real values long. The MOLD is a rank-1 array containing a complex number, which is two real values long. Therefore, the output will be a Complex rank-1 array. Since the *SIZE* is specified to be 1, only one complex value is produced. The result of the function is a Complex rank-1 array with one element: (1.1,2.2).

TRANSPOSE(MATRIX)

- Transformational function of the same type as MATRIX.
- Transposes a matrix of rank-2. Element *(i,j)* of the output has the value of MATRIX(*j*,*i*).
- MATRIX is a rank-2 matrix of any type.

UBOUND(ARRAY,*DIM*)

- Integer inquiry function.
- Returns all of the upper bounds or a specified upper bound of ARRAY.
- ARRAY is an array of any type. It must not be an unassociated pointer or an unallocated allocatable array.
  DIM is an integer in the range $1 \le DIM \le r$. The corresponding actual argument must not be an optional argument in the calling procedure.
- If *DIM* is present, the result is a scalar. If the actual argument corresponding to ARRAY is an array section or an array expression, or if dimension *DIM* has zero size, then the function will return 1. Otherwise, it will return the upper bound of

that dimension of ARRAY. If *DIM* is not present, then the function will return an array whose *i*th element is UBOUND(ARRAY,*i*) for *i*=1,2,…,*r*.

UPACK(VECTOR,MASK,FIELD)

- Transformational function of the same type as VECTOR.
- Unpacks a rank-one array into an array under the control of a mask. The result is an array of the same type and type parameters as VECTOR and the same shape as MASK.
- VECTOR is a rank-one array of any type. It must be at least as large as the number of true elements in MASK.
  MASK is a logical array.
  FIELD is of the same type as VECTOR and comformable with MASK.
- This function produces an array with the shape of MASK. The first element of the VECTOR is placed in the location corresponding to the first true value in MASK, the second element of VECTOR is placed in the location corresponding to the second true value in MASK, etc. If a location in MASK is false, then the corresponding element from FIELD is placed in the output array. If FIELD is a scalar, the same value is placed in the output array for all false locations.
- This function is the inverse of the PACK function.
- For example, suppose that $V=\begin{bmatrix}1 & 2 & 3\end{bmatrix}$, $M=\begin{bmatrix}\texttt{TRUE} & \texttt{FALSE} & \texttt{FALSE}\\ \texttt{FALSE} & \texttt{FALSE} & \texttt{FALSE}\\ \texttt{TRUE} & \texttt{FALSE} & \texttt{TRUE}\end{bmatrix}$, and $F=\begin{bmatrix}0 & 0 & 0\\ 1 & 1 & 1\\ 0 & 0 & 0\end{bmatrix}$. Then the function UNPACK(V,MASK=M,FIELD=0) would have the value $\begin{bmatrix}1 & 0 & 0\\ 0 & 0 & 0\\ 2 & 0 & 3\end{bmatrix}$, and the function UNPACK(V,MASK=M,FIELD=F) would have the value $\begin{bmatrix}1 & 0 & 0\\ 1 & 1 & 1\\ 2 & 0 & 3\end{bmatrix}$.

B.9. Miscellaneous Inquiry Functions

ALLOCATED(ARRAY)

- Logical inquiry function.
- Returns true if ARRAY is currently allocated, and false if ARRAY is not currently allocated. The result is undefined if the allocation status of ARRAY is undefined.
- ARRAY is any type of allocatable array.

`ASSOCIATED(POINTER,`*`TARGET`*`)`

- Logical inquiry function.
- There are three possible cases for this function:

  1. If *TARGET* is not present, this function returns true if POINTER is associated, and false otherwise.

  2. If *TARGET* is present and is a target, the result is true if TARGET does not have size zero and POINTER is currently associated with TARGET. Otherwise, the result is false.

  3. If *TARGET* is present and is a pointer, the result is true if both POINTER and TARGET are currently associated with the same nonzero-sized target. Otherwise, the result is false.
- POINTER is any type of pointer whose pointer association status is not undefined. *TARGET* is any type of pointer or target. If it is a pointer, its pointer association status must not be undefined.

`PRESENT(A)`

- Logical inquiry function.
- Returns true if optional argument A is present, and false otherwise.
- A is any optional argument.