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 © 1997 by Stephen J. Chapman, 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.
The 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.

```fortran
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:

```fortran
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.

```fortran
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
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

<table>
<thead>
<tr>
<th>Generic name, keyword(s), and calling sequence</th>
<th>Specific name</th>
<th>Function type</th>
<th>Section</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ABS(A)</strong></td>
<td>ABS(r)</td>
<td>Argument type</td>
<td>B.3</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>CABS(c)</td>
<td>Default real</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DABS(d)</td>
<td>Default real</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IABS(i)</td>
<td>Double Prec.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default integer</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ACHAR(I)</strong></td>
<td>Character(1)</td>
<td>B.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ACOS(X)</strong></td>
<td>ACOS(r)</td>
<td>Argument type</td>
<td>B.3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DACOS(d)</td>
<td>Default Real</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Double Prec.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ADJUSTL(STRING)</strong></td>
<td>Character</td>
<td>B.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ADJUSTR(STRING)</strong></td>
<td>Character</td>
<td>B.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>AIMAG(Z)</strong></td>
<td>AIMAG(c)</td>
<td>Real</td>
<td>B.3</td>
<td></td>
</tr>
<tr>
<td><strong>AINT(A, KIND)</strong></td>
<td>AINT(r)</td>
<td>Argument type</td>
<td>B.3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Default Real</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Double Prec.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DNINT(d)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ALL(MASK, DIM)</strong></td>
<td>Logical</td>
<td>B.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ALLOCATED(ARRAY)</strong></td>
<td>Logical</td>
<td>B.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ANINT(A, KIND)</strong></td>
<td>ANINT(r)</td>
<td>Argument type</td>
<td>B.3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Real</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Double Prec.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DNINT(d)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ANY(MASK, DIM)</strong></td>
<td>Logical</td>
<td>B.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ASIN(X)</strong></td>
<td>ASIN(r)</td>
<td>Argument type</td>
<td>B.3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ASIN(r)</td>
<td>Real</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DASIN(d)</td>
<td>Double Prec.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ASSOCIATED(POINTER, TARGET)</strong></td>
<td>Logical</td>
<td>B.9</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ATAN(X)</strong></td>
<td>ATAN(r)</td>
<td>Argument type</td>
<td>B.3</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Real</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Double Prec.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DATAN(d)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ATAN2(Y,X)</strong></td>
<td>ATAN2(r2,r1)</td>
<td>Argument type</td>
<td>B.3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DATAN2(d2,d1)</td>
<td>Real</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Double Prec.</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>BIT_SIZE(I)</strong></td>
<td>Integer</td>
<td>B.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>BTEST(I, POS)</strong></td>
<td>Logical</td>
<td>B.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CEILING(A, KIND)</strong></td>
<td>Integer</td>
<td>B.3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td><strong>CHAR(I, KIND)</strong></td>
<td>Character(1)</td>
<td>B.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fortran 90/95 Intrinsic Procedure Descriptions, a supplement to Introduction to Fortran 90/95, by Stephen J. Chapman</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------------------------------------------------------------------------------------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CMPLX(X, Y, KIND)</strong></td>
<td>Complex</td>
<td>B.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>CONJG(X)</strong></td>
<td>CONJG(c)</td>
<td>Complex</td>
<td>B.3</td>
<td></td>
</tr>
<tr>
<td><strong>COS(X)</strong></td>
<td>CCOS(c)</td>
<td>Complex</td>
<td>B.3</td>
<td></td>
</tr>
<tr>
<td><strong>COSH(X)</strong></td>
<td>COSH(r)</td>
<td>Real</td>
<td>B.3</td>
<td></td>
</tr>
<tr>
<td><strong>COUNT(MASK, DIM)</strong></td>
<td></td>
<td>Integer</td>
<td>B.8</td>
<td></td>
</tr>
<tr>
<td><strong>CPU_TIME(TIME)</strong></td>
<td></td>
<td>Subroutine</td>
<td>B.5</td>
<td></td>
</tr>
<tr>
<td><strong>CSHIFT(ARRAY, SHIFT, DIM)</strong></td>
<td></td>
<td>Array type</td>
<td>B.8</td>
<td></td>
</tr>
<tr>
<td><strong>DATE_AND_TIME(DATE, TIME, ZONE, VALUES)</strong></td>
<td></td>
<td>Subroutine</td>
<td>B.5</td>
<td></td>
</tr>
<tr>
<td><strong>DBLE(A)</strong></td>
<td>DDIM(d1,d2)</td>
<td>Double Prec.</td>
<td>B.3</td>
<td></td>
</tr>
<tr>
<td><strong>DIGITS(X)</strong></td>
<td>DDIM(r1,r2)</td>
<td>Real</td>
<td>B.3</td>
<td></td>
</tr>
<tr>
<td><strong>DIM(X,Y)</strong></td>
<td>IDIM(i1,i2)</td>
<td>Integer</td>
<td>B.3</td>
<td></td>
</tr>
<tr>
<td><strong>DOT_PRODUCT(VECTOR_A, VECTOR_B)</strong></td>
<td></td>
<td>Argument type</td>
<td>B.3</td>
<td></td>
</tr>
<tr>
<td><strong>DPROD(X,Y)</strong></td>
<td>DPROD(x1,x2)</td>
<td>Double Prec.</td>
<td>B.3</td>
<td></td>
</tr>
<tr>
<td><strong>EOSHIFT(ARRAY, SHIFT, BOUNDARY, DIM)</strong></td>
<td></td>
<td>Array type</td>
<td>B.8</td>
<td></td>
</tr>
<tr>
<td><strong>EPSILON(X)</strong></td>
<td></td>
<td>Real</td>
<td>B.3</td>
<td></td>
</tr>
<tr>
<td><strong>EXP(X)</strong></td>
<td>CEPS(c)</td>
<td>Complex</td>
<td>B.3</td>
<td></td>
</tr>
<tr>
<td><strong>EXPONENT(X)</strong></td>
<td></td>
<td>Integer</td>
<td>B.4</td>
<td></td>
</tr>
<tr>
<td><strong>FLOOR(A,KIND)</strong></td>
<td></td>
<td>Integer</td>
<td>B.3</td>
<td></td>
</tr>
<tr>
<td><strong>FRACTION(X)</strong></td>
<td></td>
<td>Real</td>
<td>B.4</td>
<td></td>
</tr>
<tr>
<td><strong>HUGE(X)</strong></td>
<td></td>
<td>Argument type</td>
<td>B.4</td>
<td></td>
</tr>
<tr>
<td><strong>IACHAR(C)</strong></td>
<td></td>
<td>Integer</td>
<td>B.7</td>
<td></td>
</tr>
<tr>
<td><strong>IAND(I,J)</strong></td>
<td></td>
<td>Integer</td>
<td>B.6</td>
<td></td>
</tr>
<tr>
<td><strong>IBCLR(I,POS)</strong></td>
<td></td>
<td>Integer</td>
<td>B.6</td>
<td></td>
</tr>
<tr>
<td><strong>IBITSI,POS,LEN)</strong></td>
<td></td>
<td>Integer</td>
<td>B.6</td>
<td></td>
</tr>
<tr>
<td><strong>IBSET(I,POS)</strong></td>
<td></td>
<td>Integer</td>
<td>B.6</td>
<td></td>
</tr>
<tr>
<td><strong>ICHAR(C)</strong></td>
<td></td>
<td>Integer</td>
<td>B.7</td>
<td></td>
</tr>
<tr>
<td><strong>IEOR(I,J)</strong></td>
<td></td>
<td>Integer</td>
<td>B.6</td>
<td></td>
</tr>
<tr>
<td><strong>INDEX(STRING, SUBSTRING, BACK)</strong></td>
<td>INDEX(str1,str2)</td>
<td>Integer</td>
<td>B.7</td>
<td></td>
</tr>
<tr>
<td><strong>INT(A, KIND)</strong></td>
<td>IDINT(i)</td>
<td>Integer</td>
<td>B.3</td>
<td></td>
</tr>
<tr>
<td><strong>IOR(I,J)</strong></td>
<td></td>
<td>Integer</td>
<td>B.6</td>
<td></td>
</tr>
<tr>
<td><strong>ISHIFT(I,SHIFT)</strong></td>
<td></td>
<td>Argument type</td>
<td>B.6</td>
<td></td>
</tr>
<tr>
<td><strong>ISHFTC(I,SHIFT,SIZE)</strong></td>
<td></td>
<td>Argument type</td>
<td>B.6</td>
<td></td>
</tr>
<tr>
<td><strong>KIND(X)</strong></td>
<td></td>
<td>Integer</td>
<td>B.4</td>
<td></td>
</tr>
<tr>
<td><strong>LBOUND(ARRAY, DIM)</strong></td>
<td></td>
<td>Integer</td>
<td>B.8</td>
<td></td>
</tr>
<tr>
<td>Procedure</td>
<td>Description</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>-------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LEN(STRING)</td>
<td>Return the length of a string</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LEN(STR)</td>
<td>Integer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LGE(STRING A, STRING B)</td>
<td>Logical</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LGT(STRING A, STRING B)</td>
<td>Logical</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LLE(STRING A, STRING B)</td>
<td>Logical</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LLT(STRING A, STRING B)</td>
<td>Logical</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOG(X)</td>
<td>Argument type</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALOG(r)</td>
<td>Real</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALOG10(r)</td>
<td>Real</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DLOG(d)</td>
<td>Double Prec.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DLOG10(d)</td>
<td>Double Prec.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOGICAL(L, KIND)</td>
<td>Logical</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MATMUL(MATRIX A, MATRIX B)</td>
<td>Argument type</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAX(A1, A2, A3, ...)</td>
<td>Argument type</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AMAX0(i1, i2, ...)</td>
<td>Real</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AMAX1(r1, r2, ...)</td>
<td>Real</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DMAX1(d1, d2, ...)</td>
<td>Double Prec.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAX0(i1, i2, ...)</td>
<td>Integer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAX1(r1, r2, ...)</td>
<td>Integer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAXEXPONENT(X)</td>
<td>Integer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAXLOC(ARRAY, DIM, MASK)</td>
<td>Integer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAXVAL(ARRAY, DIM, MASK)</td>
<td>Argument type</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MERGE(TSOURCE, FSOURCE, MASK)</td>
<td>Argument type</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MIN(A1, A2, A3, ...)</td>
<td>Argument type</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AMIN0(i1, i2, ...)</td>
<td>Real</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AMIN1(r1, r2, ...)</td>
<td>Real</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DMIN1(d1, d2, ...)</td>
<td>Double Prec.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MIN0(i1, i2, ...)</td>
<td>Integer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MIN1(r1, r2, ...)</td>
<td>Integer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MINEXPONENT(X)</td>
<td>Integer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MINLOC(ARRAY, DIM, MASK)</td>
<td>Integer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MINVAL(ARRAY, DIM, MASK)</td>
<td>Argument type</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOD(A, P)</td>
<td>Argument type</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AMOD(r1, r2)</td>
<td>Real</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MOD(i, j)</td>
<td>Integer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DMOD(d1, d2)</td>
<td>Double Prec.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MODULO(A, P)</td>
<td>Argument type</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MVBITS(FROM, FROMPOS, LEN, TO, TOPOS)</td>
<td>Subroutine</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NEAREST(X, S)</td>
<td>Real</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NINT(A, KIND)</td>
<td>Argument type</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IDNINT(i)</td>
<td>Integer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NINT(x)</td>
<td>Integer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOT(I)</td>
<td>Argument type</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NULL(MOLD)</td>
<td>Pointer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PACK(ARRAY, MASK, VECTOR)</td>
<td>Argument type</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRECISION(X)</td>
<td>Integer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRESENT(A)</td>
<td>Logical</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Argument type</td>
<td>B.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------------------------</td>
<td>---------------</td>
<td>-----</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRODUCT(ARRAY, DIM, MASK)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RADIUS(X)</td>
<td>Integer</td>
<td>B.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RANDOM_NUMBER(HARVEST)</td>
<td>Subroutine</td>
<td>B.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RANDOM_SEED(SIZE, PUT, GET)</td>
<td>Subroutine</td>
<td>B.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RANGE(X)</td>
<td>Integer</td>
<td>B.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REAL(A, KIND)</td>
<td>Real</td>
<td>B.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REAL(A, KIND)</td>
<td>Real</td>
<td>B.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REAL(A, KIND)</td>
<td>Real</td>
<td>B.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>REMPRT(STRING, NCOPIES)</td>
<td>Character</td>
<td>B.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RESHAPE(SOURCE,SHAPE,PAD, ORDER)</td>
<td>Argument type</td>
<td>B.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RRSPACING(X)</td>
<td>Argument type</td>
<td>B.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCALE(X, I)</td>
<td>Argument type</td>
<td>B.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCAN(STRING, SET, BACK)</td>
<td>Integer</td>
<td>B.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SELECTED_INT_KIND(R)</td>
<td>Integer</td>
<td>B.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SELECTED_REAL_KIND(P,R)</td>
<td>Integer</td>
<td>B.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SET_EXPONENT(X, 1)</td>
<td>Argument type</td>
<td>B.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHAPE(SOURCE)</td>
<td>Integer</td>
<td>B.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIGN(A,B)</td>
<td>Argument type</td>
<td>B.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIN(X)</td>
<td>Argument type</td>
<td>B.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SINH(X)</td>
<td>Argument type</td>
<td>B.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIZE(ARRAY, DIM)</td>
<td>Argument type</td>
<td>B.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPACING(X)</td>
<td>Argument type</td>
<td>B.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPREAD(SOURCE, DIM, NCOPIES)</td>
<td>Argument type</td>
<td>B.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQRT(X)</td>
<td>Argument type</td>
<td>B.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SYSTEM_CLOCK(COUNT, COUNT_RATE, COUNT_MAX)</td>
<td>Subroutine</td>
<td>B.5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TAN(X)</td>
<td>Argument type</td>
<td>B.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TANH(X)</td>
<td>Argument type</td>
<td>B.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TINY(X)</td>
<td>Real</td>
<td>B.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRANSFER(SOURCE, MOLD, SIZE)</td>
<td>Argument type</td>
<td>B.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRANSPOSE(MATRIX)</td>
<td>Argument type</td>
<td>B.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRIM(STRING)</td>
<td>Character</td>
<td>B.7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UBOUND(ARRAY, DIM)</td>
<td>B.8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
   \[ \text{Integer}(m) \]
   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 \( |X| \leq 1.0 \), and \( 0 \leq \text{ACOS}(X) \leq \pi \).
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 ANINT(-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| ≤ 1.0, and -π/2 ≤ ASIN(X) ≤ π/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 -π/2 ≤ ATAN(X) ≤ π/2.

ATAN2(Y, X)
- Elemental function of the same type and kind as X
- Returns the inverse tangent of Y/X in the range -π < ATAN2(Y, X) ≤ π.
- 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 ≥ 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 \( \leq A \). 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, \( x \neq 0 \).

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

\( \text{MIN}(A_1, A_2, A_3, \ldots) \)
• Elemental function of same kind as its arguments
• Returns the minimum value of \( A_1, A_2, \) etc.
• Arguments may be Real or Integer; all must be of the same type

\( \text{MOD}(A_1, P) \)
• Elemental function of same kind as its arguments
• Returns the value \( \text{MOD}(A, P) = A - P \cdot \text{INT}(A/P) \) 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:

<table>
<thead>
<tr>
<th>Function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{MOD}(5, 3) )</td>
<td>2</td>
</tr>
<tr>
<td>( \text{MOD}(-5, 3) )</td>
<td>-2</td>
</tr>
<tr>
<td>( \text{MOD}(5, -3) )</td>
<td>2</td>
</tr>
<tr>
<td>( \text{MOD}(-5, -3) )</td>
<td>-2</td>
</tr>
</tbody>
</table>

\( \text{MODULO}(A_1, P) \)
• 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 \( \text{MOD} \) function for two positive or two negative arguments; results disagree for arguments of mixed signs.
• Examples:

<table>
<thead>
<tr>
<th>Function</th>
<th>Result</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{MODULO}(5, 3) )</td>
<td>2</td>
<td>5 is 2 up from 3</td>
</tr>
<tr>
<td>( \text{MODULO}(-5, 3) )</td>
<td>1</td>
<td>-5 is 1 up from -6</td>
</tr>
<tr>
<td>( \text{MODULO}(5, -3) )</td>
<td>-1</td>
<td>5 is 1 down from 6</td>
</tr>
<tr>
<td>( \text{MODULO}(-5, -3) )</td>
<td>-2</td>
<td>-5 is 2 down from -3</td>
</tr>
</tbody>
</table>

\( \text{NEAREST}(X, S) \)
• 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 \( S \neq 0 \)

\( \text{NINT}(A, \text{KIND}) \)
• 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 \(0 \leq \text{HARVEST} < 1\). 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:
<table>
<thead>
<tr>
<th>Keyword</th>
<th>Type</th>
<th>Intent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HARVEST</td>
<td>Real</td>
<td>OUT</td>
<td>Holds random numbers.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>May be scalar or array.</td>
</tr>
</tbody>
</table>

**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:
<table>
<thead>
<tr>
<th>Keyword</th>
<th>Type</th>
<th>Intent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIZE</td>
<td>Integer</td>
<td>OUT</td>
<td>Number of integers used to hold the seed ((n))</td>
</tr>
<tr>
<td>PUT</td>
<td>Integer((m))</td>
<td>IN</td>
<td>Set the seed to the value in PUT. Note that (m \geq n).</td>
</tr>
<tr>
<td>GET</td>
<td>Integer((m))</td>
<td>OUT</td>
<td>Get the current value of the seed. Note that (m \geq n).</td>
</tr>
</tbody>
</table>
- \(SIZE\) 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

$$i = s \sum_{k=0}^{q-1} w_k \times r^k$$

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

$$i = \pm \sum_{k=0}^{30} w_k \times 2^k$$

where each $w_k$ is either 0 or 1.

The Fortran model for a real number $x$ is

$$x = \left\{ \begin{array}{ll} 0 & \text{or } 0 \times b^e \times \sum_{k=1}^{p} f_k \times b^{-k} \\ s \times b^e \times \sum_{k=1}^{p} f_k \times b^{-k} & \end{array} \right. \quad \text{(B-3)}$$

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 $e_{\text{max}}$ and some integer minimum $e_{\text{min}}$. The values of $b$, $p$, $e_{\text{min}}$, and $e_{\text{max}}$ 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, $e_{\text{max}} = 2^7 = 127$, and $e_{\text{min}} = -126$. For a typical 32-bit single precision real number on a base 2 computer, the model of the number becomes

$$x = \left\{ \begin{array}{ll} 0 & \text{or } 0 \times \left( \frac{1}{2} + \sum_{k=2}^{24} f_k \times 2^{-k} \right) \\ \pm 2^e \times \left( \frac{1}{2} + \sum_{k=2}^{24} f_k \times 2^{-k} \right) & -126 \leq e \leq 127 \end{array} \right. \quad \text{(B-4)}$$

\(^2\) 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 \( e_{\text{max}} \) 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 \( \text{RANGE}(X) \) instead.

**MINEXPONENT(X)**
• Integer inquiry function.
• Returns the minimum exponent of the same type and kind as \( X \). (The returned value is \( e_{\text{min}} \) 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 \( |x \times b^{-e} | \times b^p \), 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 $\text{ABS}(n) < 10^{**R}$. 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 $b^{e_{\text{min}} - 1}$, where $b$ and $e_{\text{min}}$ 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:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Type</th>
<th>Intent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME</td>
<td>Real</td>
<td>OUT</td>
<td>Processor time.</td>
</tr>
</tbody>
</table>

- 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:

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

- 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:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Type</th>
<th>Intent</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNT</td>
<td>Integer</td>
<td>OUT</td>
<td>Number of counts of the system clock.</td>
</tr>
<tr>
<td>COUNT_RATE</td>
<td>Integer</td>
<td>OUT</td>
<td>Number of clock counts per second.</td>
</tr>
<tr>
<td>COUNT_MAX</td>
<td>Integer</td>
<td>OUT</td>
<td>The maximum value for COUNT.</td>
</tr>
</tbody>
</table>

- 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

\[
j = \sum_{k=0}^{\zeta-1} w_k \times 2^k \tag{B-5}\]

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, \( \zeta \) is the number of bits in the integer, and the bits are numbered 0, 1, ..., 19.
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 IEO. 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.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BTEST(I,POS)</td>
<td>Logical elemental function that returns true if bit POS of I is 1, and false otherwise.</td>
</tr>
<tr>
<td>IAND(I,J)</td>
<td>Elemental function that returns the bit by bit logical AND of I and J.</td>
</tr>
<tr>
<td>IBCLR(I,POS)</td>
<td>Elemental function that returns I with bit POS set to 0.</td>
</tr>
<tr>
<td>IBITS(I,POS,LEN)</td>
<td>Elemental function that returns a right-adjusted sequence of bits extracted from I of length LEN starting at bit POS. All other bits are zero.</td>
</tr>
<tr>
<td>IBSET(I,POS)</td>
<td>Elemental function that returns I with bit POS set to 1.</td>
</tr>
<tr>
<td>IEO(I,J)</td>
<td>Elemental function that returns the bit by bit exclusive OR of I and J.</td>
</tr>
<tr>
<td>IOR(I,J)</td>
<td>Elemental function that returns the bit by bit inclusive OR of I and J.</td>
</tr>
</tbody>
</table>
• 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:

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

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 ≤ I ≤ 127, the result is the character in position I of the ASCII collating sequence. If I ≥ 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 ≤ I ≤ n-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 $0 \leq \text{ICHAR}(C) \leq n-1$, 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 $\geq$ 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 $\geq$ 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 \( 1 \leq DIM \leq r \).
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 \( 2 \times 3 \) real array \( \text{array1} \) \((r = 2)\) and two masking arrays \( \text{mask1} \) and \( \text{mask2} \) defined as follows:

\[
\begin{align*}
\text{array1} &= \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \\
\text{mask1} &= \begin{bmatrix} \text{.TRUE.} & \text{.TRUE.} & \text{.TRUE.} \\ \text{.TRUE.} & \text{.TRUE.} & \text{.TRUE.} \end{bmatrix}
\end{align*}
\]
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

\[
\text{result} = \text{MAXVAL}(\text{ARRAY}, \text{DIM}, \text{MASK})
\]

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

\[
\text{result} = \text{MAXVAL}(\text{array1}, \text{MASK}=\text{mask1})
\]

will produce a value of 6, while the function

\[
\text{result} = \text{MAXVAL}(\text{array1}, \text{MASK}=\text{mask2})
\]

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

\[
\text{result} = \text{MAXVAL}(\text{array1}, \text{DIM}=1, \text{MASK}=\text{mask1})
\]

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

\[
\text{result} = \text{MAXVAL}(\text{array1}, \text{DIM}=2, \text{MASK}=\text{mask1})
\]

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 \( 1 \leq \text{DIM} \leq r \). 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 $1 \leq DIM \leq r$. 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 $(d(1), d(2), \ldots, d(DIM-1), d(DIM+1), \ldots, d(r))$ where the shape of MASK is $(d(1), d(2), \ldots, d(r))$. 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 $1 \leq DIM \leq r$. 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 $(d(1), d(2), \ldots, d(DIM-1), d(DIM+1), \ldots, d(r))$ where the shape of MASK is $(d(1), d(2), \ldots, d(r))$. 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 $(d(1), d(2), \ldots, d(DIM-1), d(DIM+1), \ldots, d(r))$ where the shape of ARRAY is $(d(1), d(2), \ldots, d(r))$.
- DIM is an optional integer in the range $1 \leq DIM \leq r$. 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 \(1 \leq DIM \leq r\). 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 \(1 \leq DIM \leq r\). 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 \(1 \leq DIM \leq r\). 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

\[
\begin{bmatrix}
  1 & 3 & -9 \\
  2 & 2 & 6
\end{bmatrix}
\text{ and } \begin{bmatrix}
  \text{TRUE} & \text{FALSE} & \text{FALSE} \\
  \text{TRUE} & \text{TRUE} & \text{FALSE}
\end{bmatrix}
\]
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 $1 \leq DIM \leq r$. 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 $(d(1),d(2),\ldots,d(DIM-1),d(DIM+1),\ldots,d(r))$ where the shape of `ARRAY` is $(d(1),d(2),\ldots,d(r))$.
- For example, if 
  
  $\begin{bmatrix}
  1 & 3 & -9 \\
  2 & 2 & 6
  \end{bmatrix}$
  
  and
  
  $\begin{bmatrix}
  \text{TRUE} & \text{FALSE} & \text{FALSE} \\
  \text{TRUE} & \text{TRUE} & \text{FALSE}
  \end{bmatrix}$

  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 $1 \leq \text{DIM} \leq r$. 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 $(d(1), d(2), \ldots, d(\text{DIM}-1), d(\text{DIM}+1), \ldots, d(r))$ where the shape of ARRAY is $(d(1), d(2), \ldots, d(r))$. 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
  \[
  \text{ARR} = \begin{bmatrix}
    1 & 3 & -9 \\
    2 & 2 & 6 
  \end{bmatrix}
  \text{ and } \text{MASK} = \begin{bmatrix}
    \text{TRUE} & \text{FALSE} & \text{FALSE} \\
    \text{TRUE} & \text{TRUE} & \text{FALSE}
  \end{bmatrix}
  \]
  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 $1 \leq \text{DIM} \leq r$. 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 $(d(1), d(2), \ldots, d(\text{DIM}-1), d(\text{DIM}+1), \ldots, d(r))$ where the shape of ARRAY is $(d(1), d(2), \ldots, d(r))$.

- For example, if
  \[
  \text{ARR} = \begin{bmatrix}
    1 & 3 & -9 \\
    2 & 2 & 6 
  \end{bmatrix}
  \text{ and } \text{MASK} = \begin{bmatrix}
    \text{TRUE} & \text{FALSE} & \text{FALSE} \\
    \text{TRUE} & \text{TRUE} & \text{FALSE}
  \end{bmatrix}
  \]
  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 $(/3,1/)$.  

**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
  \[
  \text{ARRAY} = \begin{bmatrix} 1 & -3 \\ 4 & -2 \end{bmatrix} \quad \text{and} \quad \text{MASK} = \begin{bmatrix} \text{False} & \text{True} \\ \text{True} & \text{True} \end{bmatrix}
  \]
  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 \(\leq\) DIM \(\leq\) 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 
\( (d(1), d(2), \ldots, d(DIM-1), d(DIM+1), \ldots, d(r)) \) where the shape of ARRAY is 
\( (d(1), d(2), \ldots, d(r)) \).

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 \( (1, 2, \ldots, n) \) where \( n \) is the size of SHAPE.
- For example, if 
\[ \text{SOURCE} = [1 \ 2 \ 3 \ 4 \ 5 \ 6] , \quad \text{SHAPE} = [2 \ 5] , \quad \text{PAD} = [0 \ 0] , \]
then
\[
\text{RESHAPE}(\text{SOURCE},\text{SHAPE},\text{PAD}) = \begin{bmatrix} 1 & 3 & 5 & 0 & 0 \\ 2 & 4 & 6 & 0 & 0 \end{bmatrix}
\]
and
\[
\text{RESHAPE}(\text{SOURCE},\text{SHAPE},\text{PAD},(/2,1/)) = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 6 & 0 & 0 & 0 & 0 \end{bmatrix}
\]

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 $1 \leq DIM \leq r$. 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 $1 \leq DIM \leq r+1$.
- **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 $(s_1, s_2, ..., s_{DIM-1}, s_{DIM+1}, ..., s_{r+1})$ has the value **SOURCE**($s_1, s_2, ..., s_{DIM-1}, s_{DIM+1}, ..., s_{r+1})$.
- For example, if **SOURCE** = $[1 \ 3 \ 5]$, then the result of function
  \[
  \text{SPREAD}([1 \ 3 \ 5], \text{DIM}=1, \text{NCOPIES}=3) \]
  is the array
  \[
  \begin{bmatrix}
  1 & 3 & 5 \\
  1 & 3 & 5 \\
  \end{bmatrix}
  \]

**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 $1 \leq DIM \leq r$. 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 ≤ DIM ≤ 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 $\text{UBOUND(ARRAY, }i\text{)}$ for $i=1,2,\ldots,r$.

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 $V=[1 \ 2 \ 3]$, $M=\begin{bmatrix} \text{TRUE} & \text{FALSE} & \text{FALSE} \\ \text{FALSE} & \text{FALSE} & \text{TRUE} \end{bmatrix}$, and $F=\begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}$. Then the function $\text{UNPACK}(V,\text{MASK}=M,\text{FIELD}=0)$ would have the value $\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 2 & 0 & 3 \end{bmatrix}$, and the function $\text{UNPACK}(V,\text{MASK}=M,\text{FIELD}=F)$ would have the value $\begin{bmatrix} 1 & 1 & 1 \\ 2 & 0 & 3 \end{bmatrix}$.

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.