Subject: Physical or engineering units
From: Van Snyder

1 Number

TBD

2 Title

Physical or engineering units

3 Submitted By

J3

4 Status

For consideration.

5 Basic Functionality

Provide for numeric variables and named constants to have physical or engineering units such as length, mass, area, temperature.... "Compute" and check units for expressions and assignment at compile time. Verify that units for corresponding actual and dummy arguments are identical at compile time if an explicit interface is available. (A processor could provide for checking units for corresponding actual and dummy arguments at run time, which would be useful when procedures without explicit interface are used, but this should not be required by the standard.) Input and output units. Convert numeric values during formatted, namelist or list-directed input and output if the item has units, a conversion expression is defined, and unit input/output is requested.

It may be possible to apply units to derived types, which would be useful for such user-defined types as intervals or extended-precision numbers. This proposal does not include a provision for doing so. It does not appear to be sensible to apply units to objects of character or logical type.

6 Rationale

Incorrect use of physical units is the second-most-common error in scientific or engineering software, coming immediately after mismatching the types of actual and dummy arguments. Explicit interfaces largely solve the second problem, but do nothing directly for the first. (One can use derived types to provide a physical units system, at the expense of redefining assignment and all of the necessary operations and intrinsic functions.) A particularly expensive ($\approx \$3 \times 10^8$) and embarrassing example of an units mistake was the the cause of the loss of the Mars Climate Orbiter. The loss resulted because the NASA contract required small forces, e.g. from attitude-control maneuvers, to be reported in Newton-Seconds, but Lockheed nonetheless reported them in Pound-Seconds. (This was quite inscrutable, as Lockheed had had contracts with JPL for over thirty years, and they've *always* specified SI units.)

7 Estimated Impact

In terms of changes to the standard, this is a substantial project, requiring changes in Sections 4, 5, 6, 8, 10, and 13. Determining the units of an expression, checking units of subexpressions, and doing conversions automatically, is not tremendously difficult — I've already done it in an expression evaluator for an input routine.

8 Detailed Specification

Define a new UNIT attribute or type parameter (call it what you will) that can be specified for any numeric variable or named constant. Literal constants are unitless except in one case explained below.

Four varieties of units are defined: *Atomic* units are not defined in terms of any other units. *Composite* units are defined in terms of atomic units or other composite units. *Value-converting* units convert values according to specified linear conversions. *Abstract units* are provided to specify the relation between units for dummy arguments and function results. They can be either atomic or composite, but not value-converting. Non-abstract units participate in generic resolution; abstract units do not.

Multiplication and division operations on units are defined. Exponentiation by an unitless integer constant is defined to be equivalent to repeated multiplication or division.

Each unit declaration creates a *unit coercion function* having the same name as the unit, that takes an argument with any unit, and coerces it to have the unit specified by the name of the function. If the unit of the argument and the result are related by a linear conversion expression, value conversion is also implied. There is an intrinsic UNITLESS coercion function.

Quantities can be added, subtracted, assigned, or compared by relational operations only if they have equivalent units. Atomic and value-converting units are equivalent by name. Composite units are equivalent by structure. If a dummy argument has a unit that is not abstract, the associated actual argument shall have an equivalent unit.

Abstract units allow enforcing a particular relation between the units, without requiring particular units. For example, the SQRT intrinsic function result has abstract units A, and its argument has abstract units A*A. If a dummy argument has an abstract unit, the associated actual argument can have any unit, but if several dummy arguments have abstract units, the units of their corresponding actual arguments shall be related in the same way as the units of the dummy arguments. If a function result has an abstract unit, and that unit is related to units of dummy arguments, the unit of the result of invoking the function is related to the actual arguments in the same way.

There is an intrinsic RADIAN unit, and a parallel set of generic intrinsic trigonometric functions that take RADIAN arguments and produce unitless results. All of the remaining intrinsic procedures have arguments with abstract units and results that are unitless (e.g. SELECTED_INT_KIND) or have the same units as their argument (e.g. TINY). Because function results do not participate in generic resolution, it is not possible to have a parallel set of generic intrinsic inverse trigonometric functions that return RADIAN results. It may be useful to provide an intrinsic module that has some public units and procedures, e.g. units TICK and SECOND and a SYSTEM_CLOCK module procedure that has arguments with units TICK, TICK/SECOND and SECOND.

When quantities are added or subtracted, the units of the result are the same as the units of the operands. When quantities are multiplied or divided, the units of the result are the units that result from applying the operation to the operands' units. Multiplication or division by an unitless operand produces a result having the same units as the other operand. Exponentiation by an unitless integer constant is defined to be equivalent to multiplication or division. In an exponentiation operation, the exponent shall be unitless. For intrinsic assignment, the units shall be the same.

There are two forms of the UNIT statement. One without (*unit-name*) declares and defines units. Units are declared to be atomic by appearing without a definition. Units are declared to be composite or value-converting by having a defining expression that uses literal constants, unitless named constants, multiplication, division, exponentiation by an unitless integer, previously-declared unit names, and, in the case of value-converting units, addition or subtraction. Units are declared to be abstract by having the ABSTRACT attribute in their declarations. A nonatomic abstract unit shall be defined in terms of abstract units.

Value-converting units are defined using linear expressions equivalent in form to $U = [\,a\,]\,U'\,[\,\pm\,b\,]$, where U is thereby declared to be an unit name, U' is required to be a previously declared nonabstract unit name, and $a \neq 0$ and b are unitless numeric initialization expressions. A value-converting unit is defined even if a is absent or one and b is absent or zero. In these expressions, a is considered to have units U/U' and b is considered to have units U, but we can't say that since U isn't defined yet. This is the

only case where literal constants are not unitless. This explicitly defines a conversion function from U to U' and implicitly defines its inverse (always possible because a is required to be nonzero). Although U' need not be atomic, if it is atomic it remains atomic even though an inverse conversion is defined. Value-converting functions are generic, since there may be several implicitly-defined inverse conversions. Neither U nor U' shall be abstract.

Value conversions are transitive. Since each unit can only be declared once, and cannot be referenced before being declared, an explicitly circular dependence of conversions is impossible. There will usually be an atomic unit involved in one of the value conversions, but a composite unit is possible. There are examples of transitive conversions, multiple implicit conversions, and conversions that depend on nonatomic units below.

Value conversion is implied during input or explicit units conversion using the units conversion function implied by the unit declaration or its inverse, provided the argument or input value is related to the result or input list item by a sequence of explicit or implicit conversions. Value conversion does not occur during intrinsic assignment (this could be an extension), argument association or output. When conversion is applied, all constants within a and b are converted to the kind of the argument expression or input list item. This specification could interact with the proposal to include function result types in the criteria for generic resolution.

> The Mars Climate Orbiter crashed because quantities with the wrong units were written by one program, and assumed to have the correct units by another program. Automatic units checking and value conversion would have let Lockheed use whatever units they wanted to use, so long as the JPL software had the unit name, and the appropriate conversion, available. If JPL software insisted on units, and the Lockheed data were unitless, or used units that JPL software did not specify, the error would have been detected.

In all other unit-defining expressions, the only constants allowed are 1 and 1.0, or named constants having those values.

Variables or named constants are declared to have units by specifying UNIT(*unit-name*) in their declarations, **unit-name* after their names in declaration statements, or by an UNIT(*unit-name*) statement.

UNIT(*unit-name*) is allowed in the *prefix* of a *function-stmt*.

There is an optional U[w] suffix to numeric format descriptors, that causes units to be output by write statements, or input and checked by read statements. The text of the unit that is output, or checked during input, is the same as the unit name, except that case of letters is not significant. There is a specification in OPEN, READ and WRITE statements that controls whether units for numeric quantities that have units are to be output (checked) by namelist or list-directed output (input). There is a specification for the INQUIRE statement to inquire whether this mode is set by an OPEN statement for a connection. A single specification, rather than separate ones for namelist and list-directed transfers, is adequate.

Some thought and debate will be necessary to decide what to do about input and output of arrays that have units. Should the value in the input for every element be required to specify its units, or is it enough that at least one does? If one is enough should it be specified to be first (or last)? Should units be provided for every element of output, or is one enough? There is not a problem for array constructors, since they can be wrapped with an units coercion or conversion function.

8.1 Examples

8.1.1 Atomic and composite units

```
UNIT :: INCH, SECOND
UNIT :: CM, INCH_TO_CM = CM / INCH
REAL, PARAMETER, UNIT(INCH_TO_CM) :: CONVERT = INCH_TO_CM(2.54)
UNIT :: SQINCH = INCH * INCH ! or INCH ** 2
UNIT :: IPS = INCH / SECOND, Hz = 1 / SECOND ! or SECOND ** (-1)
REAL, UNIT(SQINCH) :: A
REAL, UNIT(Hz) :: F
```

```
REAL, UNIT(INCH) :: L, L2, C*CM
REAL, UNIT(SECOND) :: T
REAL :: V
UNIT(IPS) :: V

V = A + L                   ! INVALID -- SQINCH cannot be added to INCH,
                            ! and neither one can be assigned to IPS
V = IPS(A + SQINCH(L))      ! VALID -- I'm screwing this up intentionally
V = (A / L + L2) / T        ! VALID -- IPS is compatible with INCH / SECOND
A = L * L2                  ! VALID -- SQINCH is compatible with INCH * INCH
F = V / L                   ! VALID -- units of LHS and RHS are both 1/SECOND
C = CONVERT * L             ! VALID -- CM / INCH * INCH = CM
L = SQRT(A) * 5.0e-3        ! VALID -- exercise for reader
```

8.1.2 Abstract units

```
INTERFACE
  REAL UNIT(UR) FUNCTION CBRT ( A )
    UNIT, ABSTRACT :: UR, UA = UR**3
    REAL, UNIT(UA) :: A
  END FUNCTION CBRT
END INTERFACE
```

8.1.3 Value-converting units

```
UNIT :: MHz = 1.0e6 * Hz, GHz = 1000 * MHz, KHz = 0.001 * MHz
UNIT :: F, C = ( F - 32 ) * 5.0 / 9.0 ! Also defines (atomic) F = 1.8 * C + 32.0
UNIT :: DEGREES = 45.0 * RADIAN / ATAN(1.0)
REAL, UNIT(Radian) :: Angle = Degrees(45) ! = Radian(0.785398163)
REAL, UNIT(MHz) :: Frq = MHz(3310.0)
REAL, UNIT(C) :: Temp = F(212.0)          ! = C(100.0)
...
CHARACTER(32) :: LINE
WRITE(LINE,*,UNITS="yes") GHz(Frq), F(Temp), Degrees(Angle)
READ(LINE,*,UNITS="yes") Frq, Temp, Angle
```

Execution of the `WRITE` statement causes LINE to have the value `"3.31 GHz 212.0 F 45.0 Degrees"` (approximately). Execution of the `READ` statement causes the variables `Frq`, `Temp` and `Angle` to get their original values (approximately).

Notice that `MHz(3.31)`, `C(212)` and `Radian(45)` have values 3.31, 212 and 45 respectively, with units `MHz`, `C` and `Radian` respectively, not 3310.0, 100.0 and 0.785398163 respectively, since generic resolution selects the simple unit-coercion functions, not the conversion ones, for unitless arguments.

Input of the form `"3.3021148036e-10 Second 212 F 45 Degrees"` is not permitted, since the declaration of Hz does not define a value-converting unit.

If we have

```
REAL(kind(0.0d0)), UNIT(Radian) :: AngleD
```

then in Degrees(AngleD) the 1.0 that is the argument of ATAN in the definition of the Degrees conversion function is converted to 1.0d0.

9 History