

33. Real data types

This clause describes support in the *e* language of the **real** data type. Objects of type **real** are double precision floating point numbers, similar to the precision of a C type **double**.

33.1 Real data type usage

In any context expecting a numeric object, a **real** object is acceptable, except in the following cases:

- Both operands of the shift operators (\ll , \gg)
- Bitwise operators ($|$, $\&$, \wedge)
- Bitwise routines
- Modulo (%)
- **odd()**
- **even()**

33.2 Real literals

Real literals are numbers that have a decimal point or an exponential part or both. If a decimal point exists, there must be digits on both sides of the decimal point. Underscores can be added for readability and are ignored.

Table 46—Examples of real literals

Real Constant	Value
5.3876e4	53,876
4e-11	0.00000000004
1e+5	100,000
7.321E-3	0.007321
3.2E+4	32,000
0.5e-6	0.0000005
0.45	0.45
6.e10	60,000,000,000

33.3 Real constants

The **real** constants in Table 47 and Table 48 are defined in both *e* code and in C code that includes a suitable header file:

NOTE— All mathematical constants are prefixed by P1647_M_.

NOTE— All physical constants are prefixed by P1647_P_.

Table 47—Mathematical constants

Constant	Value
IEEE_1647_M_E	e
IEEE_1647_M_LOG2E	Logarithm base 2 of e
IEEE_1647_M_LOG10E	Logarithm base 10 of e
IEEE_1647_M_LN2	Natural logarithm of 2
IEEE_1647_M_LN10	Natural logarithm of 10
IEEE_1647_M_PI	PI
IEEE_1647_M_TWO_PI	2*PI
IEEE_1647_M_PI_2	PI/2
IEEE_1647_M_PI_4	PI/4
IEEE_1647_M_1_PI	1/PI
IEEE_1647_M_2_PI	2/PI
IEEE_1647_M_2_SQRTPI	2/sqrt(PI)
IEEE_1647_M_SQRT2	sqrt(2)
IEEE_1647_M_SQRT1_2	sqrt(1/2)

Table 48—Physical constants

Constant	Value
IEEE_1647_P_Q	Charge of electron in coulombs
IEEE_1647_P_C	Speed of light in vacuum in meters/sec
IEEE_1647_P_K	Boltzmann's constant in joules/kelvin
IEEE_1647_P_H	Planck's constant in joules*sec
IEEE_1647_P_EPS0	Permittivity of vacuum in farads/meter
IEEE_1647_P_U0	Permeability of vacuum in henrys/meter
IEEE_1647_P_CELSIUS0	Zero Celsius in kelvin

33.4 Conversion between real and integer data types

Automatic casting is performed between the **real** type and the other numeric types.

Converting a **real** type object to an integer type object uses the following process:

- a) The object is first converted to type **int** (bits:*) with the value of the largest integer whose absolute value is less than or equal to the absolute value of the **real** object.

- b) The object is then converted to the expected integer type.

Additional rules apply to converting **real** objects to integer objects:

- If the object's floating-point value is infinity (inf), negative infinity (-inf), or Not-a-Number (NaN), an error will be emitted when trying to convert to an integer value.
- When converting an integer object to the **real** type, the object is converted to the value closest to the integer value that can be represented in the double precision format.

When converting from an integer data type to a real, the integer value is simply converted to its identical value represented as a real.

33.5 Conversions using the **as_a()** operator

Converting a non-numeric scalar type object to a **real** type object using the **as_a()** operator uses the following process:

- a) The scalar type object is first converted to an integer value.
- b) The object is then converted to a **real** value according to process and rules listed in [33.4](#).

Additional rules apply to converting non-numeric scalar objects to **real** objects using the **as_a()** operator:

- When converting a string value to **real** using the **as_a()** operator, the string is parsed as if it was a **real** literal, and the value of the **real** literal is returned.
- If the string does not conform to the definition of a **real** literal, an error is emitted.

33.6 Real data type precision, data conversion, and sign extension

The rules for deciding precision, performing data conversion, and sign extension are as follows:

- a) Determine the context of the expression. The context may be comprised of up to three types.
- b) If all types involved in an expression and its context are integer values of 32 bits in width or less:
 - 1) The operation is performed in 32 bits.
 - 2) If any of the types are unsigned, the operation is performed with unsigned integers.
NOTE—Decimal constants are treated as signed integers, whether they are negative or not. All other constants are treated as unsigned integers unless preceded by a hyphen.
 - 3) Each operand is automatically cast, if necessary, to the required type.
NOTE—Casting of small negative numbers (signed integers) to unsigned integers produces large positive numbers.
- c) If all types are integer types, and any of the types is greater than 32 bits:
 - 1) The operation is performed in infinite precision (int(bits:*)).
 - 2) Each operand is zero-extended if it is unsigned, or sign-extended if it is signed, to infinite precision.
- d) If any of the types is a **real** type, then the operation is done in double precision, and all objects should first be converted according to the rules described above.

33.7 Packing values for real types

Real values take up 64 bits when packed. These bits are actual bit representation of the double value. The effect of the various packing options on real type objects is similar to their effect on an integer (bits:64) value.

33.8 Printing real values

By default, **real** values are printed similar to the %g format of printf in C. When using formatted printing, you can use the %e, %f, and %g formats similar to C. Printing an integer value with these formats will cause an automatic conversion to the **real** type. Printing **real** values with integer formatting will cause an automatic conversion to int (bits:*)).

33.9 Arithmetic routines supporting real type

The following arithmetic routines support of **real** type objects:

Table 49—Arithmetic routines supporting real types

Routine	Description
floor(real): real	Returns the largest integer that is less than or equal to the parameter.
ceil(real): real	Returns the smallest integer that is greater than or equal to the parameter.
round(real): real	Returns the closest integer to the parameter. In the case of a tie then it returns the integer with the higher absolute value.
log(real): real	Returns the natural logarithm of the parameter.
log10(real): real	Returns the base-10 logarithm of parameter.
pow(real, real): real	Returns the value of the first parameter raised to the power of second one.
sqrt(real): real	Returns the square root of the parameter.
exp(real): real	Returns the value of e raised to the power of the parameter.
sin(real): real	Returns the sine of the parameter given in radians.
cos(real): real	Returns the cosine of the parameter given in radians.
tan(real): real	Returns the tangent of the parameter given in radians.
asin(real): real	Returns the arc sine of the parameter.
acos(real): real	Returns the arc cosine of the parameter.
atan(real): real	Returns the arc tangent of the parameter.
sinh(real): real	Returns the hyperbolic sine of the parameter.
cosh(real): real	Returns the hyperbolic cosine of the parameter.
tanh(real): real	Returns the hyperbolic tangent of the parameter.
asinh(real): real	Returns the inverse hyperbolic sine of the parameter.
acosh(real): real	Returns the inverse hyperbolic cosine of the parameter.
atanh(real): real	Returns the inverse hyperbolic tangent of the parameter.
atan2(real, real): real	Returns the arc tangent of the two parameters.
hypot(real, real): real	Returns the distance of the point defined by the two parameters from the origin.
is_nan(real): bool	Returns TRUE if the parameter's value is Not-a-Number (NaN).
is_finite(real): bool	Returns TRUE if the parameter's value is a finite real value that is, it is not infinity, negative infinity, or NaN).

NOTE—For integer routines like **ilog()**, **ilog10()**, **ilog2()**, **ipow()**, and **isqrt()**, whose return type is based on the expected type, if the expected type is **real**, then the return type is **int** (bits:*) .

33.10 Random routines

Specman supports the following routines to generate random **real** numbers:

Table 50—Random routines

Routine	Description
<code>rdist_uniform(from: real, to:real): real</code>	Returns a random real number using uniform distribution in the range from to to .

33.11 C interface macros that support real type objects

The following C interface macros support **real** type objects:

Table 51—C interface macros supporting real type objects

Macro	Description
<code>P1647_REAL_NEW(c_double_value)</code>	Returns a Specman allocated real object with its value specified by the parameter <code>c_double_value</code> of type C double.
<code>P1647_REAL_GET ()</code>	Receives an object of type <code>SN_TYPE(real)</code> and returns a C double value.
<code>P1647_TYPE(real)</code>	Declares an <i>e</i> real type in C.

33.12 Real type limitations

- The key of a keyed list cannot be of type **real**.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65