

# Unit bigdecimalmath

---

[Units](#)

[Class Hierarchy](#)

[Classes, Interfaces, Objects and Records](#)

[Types](#)

[Variables](#)

[Constants](#)

[Functions and Procedures](#)

[Identifiers](#)

<a href="#">Description</a>	<a href="#">Uses</a>	<a href="#">Classes, Interfaces, Objects and Records</a>	<a href="#">Functions and Procedures</a>	<a href="#">Types</a>	<a href="#">Constants</a>	<a href="#">Variables</a>
-----------------------------	----------------------	--	--	-----------------------	---------------------------	---------------------------

## Description

An unit for arbitrary precision arithmetic on bcd floats

See [BigDecimal](#)

## Overview

### Classes, Interfaces, Objects and Records

Name	Description
record <a href="#">BigDecimal</a>	Big Decimal type

### Functions and Procedures

```
function TryStrToBigDecimal(const s: string; res: PBigDecimal; errCode:
PBigDecimalErrorCode = nil): boolean;
function StrToBigDecimal(const s: string): BigDecimal; inline;
function BigDecimalToStr(const v: BigDecimal; format: TBigDecimalFormat = bdfExact):
string;
function BigDecimalToLongint(const a: BigDecimal): Longint;
function BigDecimalToInt64(const a: BigDecimal): Int64;
function BigDecimalToExtended(const a: BigDecimal): Extended;
operator :=(const a: Integer): BigDecimal;
operator :=(const a: Int64): BigDecimal;
operator :=(const a: QWord): BigDecimal;
```

```

operator :=(const a: Extended): BigDecimal; deprecated 'Direct casting of float to
bigdecimal might lead to rounding errors. Consider using StrToBigDecimal.';
operator -(const a: BigDecimal): BigDecimal;
operator +(const a: BigDecimal; const b: BigDecimal): BigDecimal;
operator -(const a: BigDecimal; const b: BigDecimal): BigDecimal;
operator *(const a: BigDecimal; const b: BigDecimal): BigDecimal;
operator /(const a: BigDecimal; const b: BigDecimal): BigDecimal;
operator div(const a: BigDecimal; const b: BigDecimal): BigDecimal;
operator mod(const a: BigDecimal; const b: BigDecimal): BigDecimal;
operator **(const a: BigDecimal; const b: int64): BigDecimal;
procedure divideModNoAlias(out quotient, remainder: BigDecimal; const a, b: BigDecimal;
targetPrecision: integer = 18; flags: TBigDecimalDivisionFlags =
[bddfKeepDividentPrecision, bddfKeepDivisorPrecision, bddfAddHiddenDigit]);
function divide(const a, b: BigDecimal; maximalAdditionalFractionDigits: integer = 18;
flags: TBigDecimalDivisionFlags = [bddfKeepDividentPrecision, bddfKeepDivisorPrecision,
bddfAddHiddenDigit]): BigDecimal;
procedure shift10(var v: BigDecimal; shift: integer);
function shifted10(const v: BigDecimal; shift: integer): BigDecimal;
function compareBigDecimals(const a, b: BigDecimal): integer;
operator <(const a: BigDecimal; const b: BigDecimal): boolean;
operator <=(const a: BigDecimal; const b: BigDecimal): boolean;
operator =(const a: BigDecimal; const b: BigDecimal): boolean;
operator >=(const a: BigDecimal; const b: BigDecimal): boolean;
operator >(const a: BigDecimal; const b: BigDecimal): boolean;
procedure normalize(var x: BigDecimal);
function precision(const v: BigDecimal): integer;
function round(const v: BigDecimal; toDigit: integer = 0; roundingMode:
TBigDecimalRoundingMode = bfrmRound): BigDecimal; overload;
function roundInRange(mi, exact, ma: BigDecimal): BigDecimal;
function getDigit(const v: BigDecimal; digit: integer): BigDecimalBin;
procedure setZero(out r: BigDecimal);
procedure setOne(out r: BigDecimal);
function isZero(const v: BigDecimal): boolean; overload;
function isIntegral(const v: BigDecimal): boolean;
function isLongint(const v: BigDecimal): boolean;
function isInt64(const v: BigDecimal): boolean;
function odd(const v: BigDecimal): boolean; overload;
function even(const v: BigDecimal): boolean; overload;
function abs(const v: BigDecimal): BigDecimal; overload;
function power(const v: BigDecimal; const exp: Int64): BigDecimal; overload;
function sqrt(const v: BigDecimal; precision: integer = 9): BigDecimal; overload;
function gcd(const a,b: BigDecimal): BigDecimal; overload;
function lcm(const a,b: BigDecimal): BigDecimal; overload;
function fastpower2to(const exp: Int64): BigDecimal;
function fastpower5to(const exp: Int64): BigDecimal;

```

## Types

```

BigDecimalBin = shortint;
BigDecimalBinSquared = longint;
BigDecimalBin = smallint ;
BigDecimalBinSquared = longint;
BigDecimalBin = smallint;
BigDecimalBinSquared = longint;

```

```

BigDecimalBin = smallint;
BigDecimalBinSquared = longint;
BigDecimalBin = integer;
BigDecimalBinSquared = int64;
BigDecimalBin = integer;
BigDecimalBinSquared = int64;
BigDecimalBin = integer;
BigDecimalBinSquared = int64;
BigDecimalBin = integer;
BigDecimalBinSquared = int64;
BigDecimalBin = integer;
BigDecimalBinSquared = int64;
PBigDecimal = ^BigDecimal;
TBigDecimalErrorCode = (...);
PBigDecimalErrorCode = ^TBigDecimalErrorCode;
TBigDecimalFormat = (...);
TBigDecimalFloatFormat = (...);
TBigDecimalDivisionFlags = set of (bddfKeepDividentPrecision, bddfKeepDivisorPrecision,
bddfAddHiddenDigit, bddfFillIntegerPart, bddfNoFractionalPart);
TBigDecimalRoundingMode = (...);

```

## Constants

```

DIGITS_PER_ELEMENT = 1;
ELEMENT_OVERFLOW = 10;
DIGITS_PER_ELEMENT = 2;
ELEMENT_OVERFLOW = 100;
DIGITS_PER_ELEMENT = 3;
ELEMENT_OVERFLOW = 1000;
DIGITS_PER_ELEMENT = 4;
ELEMENT_OVERFLOW = 10000;
DIGITS_PER_ELEMENT = 5;
ELEMENT_OVERFLOW = 100000;
DIGITS_PER_ELEMENT = 6;
ELEMENT_OVERFLOW = 1000000;
DIGITS_PER_ELEMENT = 7;
ELEMENT_OVERFLOW = 10000000;
DIGITS_PER_ELEMENT = 8;
ELEMENT_OVERFLOW = 100000000;
DIGITS_PER_ELEMENT = 9;
ELEMENT_OVERFLOW = 1000000000;

```

## Description

### Functions and Procedures

```

function TryStrToBigDecimal(const s: string; res: PBigDecimal; errCode:
PBigDecimalErrorCode = nil): boolean;

```

Converts a decimal string to a [bigdecimal](#).

Supports standard decimal notation, like -123.456 or 1E-2 (-(?[0-9]+)(.[0-9]+)?([eE][+-]?[0-9]+))

```

function StrToBigDecimal(const s: string): BigDecimal; inline;

```

Converts a decimal string to a [bigdecimal](#).

Supports standard decimal notation, like -123.456 or 1E-2 (-?[0-9]+(.[0-9]+)?([eE][+-]?[0-9]+)) Raises an exception on invalid input.

```
function BigDecimalToStr(const v: BigDecimal; format: TBigDecimalFormat = bdfExact): string;
```

Converts a [bigdecimal](#) to a decimal string

The result will be fixed width format [0-9]+([0-9]+)?, [even](#) if the input had an exponent

```
function BigDecimalToLongint(const a: BigDecimal): Longint;
```

Converts a [bigdecimal](#) to a native int (can overflow)

```
function BigDecimalToInt64(const a: BigDecimal): Int64;
```

Converts a [bigdecimal](#) to a native int (can overflow)

```
function BigDecimalToExtended(const a: BigDecimal): Extended;
```

Converts a [bigdecimal](#) to an extended (may introduce rounding errors)

```
operator :=(const a: Integer): BigDecimal;
```

Converts a native integer to a [BigDecimal](#)

```
operator :=(const a: Int64): BigDecimal;
```

Converts a native integer to a [BigDecimal](#)

```
operator :=(const a: QWord): BigDecimal;
```

Converts a native integer to a [BigDecimal](#)

```
operator :=(const a: Extended): BigDecimal; deprecated 'Direct casting of float to bigdecimal might lead to rounding errors. Consider using StrToBigDecimal.';
```

Warning: this symbol is deprecated: Direct casting of float to bigdecimal might lead to rounding errors. Consider using StrToBigDecimal.

Converts an extended to a [BigDecimal](#)

Marked as deprecated, because it may lead to rounding errors. FloatToBigDecimal is exact, but probably some magnitudes slower. For constant values [StrToBigDecimal](#) should be used instead.

```
operator -(const a: BigDecimal): BigDecimal;
```

Standard operator unary -

```
operator +(const a: BigDecimal; const b: BigDecimal): BigDecimal;
```

Standard operator binary +

```
operator -(const a: BigDecimal; const b: BigDecimal): BigDecimal;
```

Standard operator binary -

```
operator *(const a: BigDecimal; const b: BigDecimal): BigDecimal;
```

Standard operator binary \*

```
operator /(const a: BigDecimal; const b: BigDecimal): BigDecimal;
```

Standard operator binary /

If the result can not be represented as finite decimal number (e.g. 1/3) it will be calculated with 18 digit precision after the decimal point, with an additional hidden digit for rounding (so 1/3 is 0.3333333333333333, and 0.3333333333333333\*3 is 0.9999999999999999, but (1/3) \* 3 is 1).

```
operator div(const a: BigDecimal; const b: BigDecimal): BigDecimal;
```

Standard operator binary div

The result is an integer, so 1.23E3 / 7 will be 175

```
operator mod(const a: BigDecimal; const b: BigDecimal): BigDecimal;
```

Standard operator binary mod

Calculates the remainder of an integer division  $a - (a \text{ div } b) * b$

```
operator **(const a: BigDecimal; const b: int64): BigDecimal;
```

Standard operator binary \*\*

```
procedure divideModNoAlias(out quotient, remainder: BigDecimal; const a, b: BigDecimal;
```

```
targetPrecision: integer = 18; flags: TBigDecimalDivisionFlags =
[bddfKeepDividentPrecision, bddfKeepDivisorPrecision, bddfAddHiddenDigit]);
```

Universal division/modulo function. Calculates the quotient and remainder of a / b.

**bddfKeepDividentPrecision**: calculates as least as many non-zero digit of the quotient as the dividend (1st arg) has

**bddfKeepDivisorPrecision**: calculates as least as many non-zero digit of the quotient as the divisor (2nd arg) has

**bddfAddHiddenDigit**: Calculates an additional digit for rounding, which will not be displayed by [BigDecimalToStr](#)

**bddfFillIntegerPart**: Calculate at least all digits of the integer part of the quotient, independent of the precision of the input

**bddfNoFractionalPart**: Do not calculate the fractional part of the quotient (remember that a [bigdecimal](#) is a scaled integer. So **bddfFillIntegerPart** ensures that the result has not less digits than an integer division (necessary in case of an exponent > 0) and **bddfKillFractions** that the result has not more digits than an integer division (in case of an exponent < 0) )

not all flag combinations were tested

#### Parameters

**maximalAdditionalFractionDigits**

How many digits should be added to the quotient, if the result cannot be represented with the current precision

**flags**

Division options:

```
function divide(const a, b: BigDecimal; maximalAdditionalFractionDigits: integer = 18;
flags: TBigDecimalDivisionFlags = [bddfKeepDividentPrecision, bddfKeepDivisorPrecision,
bddfAddHiddenDigit]): BigDecimal;
```

Wrapper around [divideModNoAlias](#), ignoring the calculated remainder

```
procedure shift10(var v: BigDecimal; shift: integer);
```

Calculates a decimal shift:  $v := v * 10^{\text{shift}}$

```
function shifted10(const v: BigDecimal; shift: integer): BigDecimal;
```

Calculates a decimal shift:  $\text{result} := v * 10^{\text{shift}}$

```
function compareBigDecimals(const a, b: BigDecimal): integer;
```

Compares the big decimals. Returns -1, 0 or 1 corresponding to  $a <$ ,  $=$  or  $> b$

```
operator <(const a: BigDecimal; const b: BigDecimal): boolean;
```

```
operator <=(const a: BigDecimal; const b: BigDecimal): boolean;
```

```
operator =(const a: BigDecimal; const b: BigDecimal): boolean;
```

```
operator >=(const a: BigDecimal; const b: BigDecimal): boolean;
```

```
operator >(const a: BigDecimal; const b: BigDecimal): boolean;
```

```
procedure normalize(var x: BigDecimal);
```

Removes leading (pre .) and trailing (post .) zeros

```
function precision(const v: BigDecimal): integer;
```

How many non-zero digits the number contains

```
function round(const v: BigDecimal; toDigit: integer = 0; roundingMode:
TBigDecimalRoundingMode = bfrmRound): BigDecimal; overload;
```

## Universal rounding function

Rounds *v* to the precision of a certain digit, subject to a certain rounding mode.

Positive *toDigit* will *round* to an integer with *toDigit* trailing zeros, negative *toDigit* will *round* to a decimal with *-toDigit* numbers after the decimal point

```
function roundInRange(mi, exact, ma: BigDecimal): BigDecimal;
```

Given *mi* < *exact* < *ma*, truncate *exact* to a *bigdecimal* result, such that

*mi* < *result* < *ma*

*result* has the minimal number of non-zero digits

| *result* - *exact* | is minimized

```
function getDigit(const v: BigDecimal; digit: integer): BigDecimalBin;
```

Returns the *digit*-th digit of *v*.

Last integer digit is digit 0, digits at negative indices are behind the decimal point.

```
procedure setZero(out r: BigDecimal);
```

Sets the *bigdecimal* to 0

```
procedure setOne(out r: BigDecimal);
```

Sets the *bigdecimal* to 1

```
function isZero(const v: BigDecimal): boolean; overload;
```

Returns true iff the *bigdecimal* is zero

```
function isIntegral(const v: BigDecimal): boolean;
```

Returns true iff *v* has no fractional digits

```
function isLongint(const v: BigDecimal): boolean;
```

Returns true iff *v* has no fractional digits and can be stored within an longint (32 bit integer)

```
function isInt64(const v: BigDecimal): boolean;
```

Returns true iff *v* has no fractional digits and can be stored within an int64

```
function odd(const v: BigDecimal): boolean; overload;
```

Checks if *v* is odd. A number with fractional digits is never odd (only weird)

```
function even(const v: BigDecimal): boolean; overload;
```

Checks if *v* is even. A number with fractional digits is never even (and neither odd, which is odd)

```
function abs(const v: BigDecimal): BigDecimal; overload;
```

Returns the absolute value of *v*

```
function power(const v: BigDecimal; const exp: Int64): BigDecimal; overload;
```

Calculates *v* \*\* *exp*, with *exp* being an integer

```
function sqrt(const v: BigDecimal; precision: integer = 9): BigDecimal; overload;
```

Calculates the square root of *v*, to *precision* digits after the decimal point

Not much tested

```
function gcd(const a,b: BigDecimal): BigDecimal; overload;
```

Calculates the greatest common denominator (only makes sense for positive integer input)

```
function lcm(const a,b: BigDecimal): BigDecimal; overload;
```

Calculates the least common multiple

```
function fastpower2to(const exp: Int64): BigDecimal;
```

Calculates 2 \*\* *exp* exactly, with *exp* being an integer (faster than *power* for negative *exp*)

```
function fastpower5to(const exp: Int64): BigDecimal;
```

Calculates 5 \*\* *exp* exactly, with *exp* being an integer (faster than *power* for negative *exp*)

## Types

```
BigDecimalBin = shortint;
```

**BigDecimalBinSquared** = longint;

**BigDecimalBin** = smallint ;

**BigDecimalBinSquared** = longint;

**BigDecimalBin** = smallint;

**BigDecimalBinSquared** = longint;

**BigDecimalBin** = smallint;

**BigDecimalBinSquared** = longint;

**BigDecimalBin** = integer;

**BigDecimalBinSquared** = int64;

**BigDecimalBin** = integer;

**BigDecimalBinSquared** = int64;

**BigDecimalBin** = integer;

**BigDecimalBinSquared** = int64;

**BigDecimalBin** = integer;

**BigDecimalBinSquared** = int64;

**BigDecimalBin** = integer;

**BigDecimalBinSquared** = int64;

**PBigDecimal** = ^[BigDecimal](#);

**TBigDecimalErrorCode** = (...);

#### Values

- bdceNoError:
- bdceParsingInvalidFormat:
- bdceParsingTooBig:

**PBigDecimalErrorCode** = ^[TBigDecimalErrorCode](#);

**TBigDecimalFormat** = (...);

#### Values

- bdfExact:
- bdfExponent:

**TBigDecimalFloatFormat** = (...);

#### Values

- bddfExact:
- bddfShortest:

**TBigDecimalDivisionFlags** = set of (bddfKeepDividentPrecision, bddfKeepDivisorPrecision, bddfAddHiddenDigit, bddfFillIntegerPart, bddfNoFractionalPart);

**TBigDecimalRoundingMode** = (...);

#### Values

- bfrmTrunc:
- bfrmCeil:
- bfrmFloor:
- bfrmRound:
- bfrmRoundHalfUp:
- bfrmRoundHalfToEven:

## Constants

**DIGITS\_PER\_ELEMENT** = 1;

**ELEMENT\_OVERFLOW** = 10;

**DIGITS\_PER\_ELEMENT** = 2;

**ELEMENT\_OVERFLOW** = 100;

**DIGITS\_PER\_ELEMENT** = 3;

**ELEMENT\_OVERFLOW** = 1000;

**DIGITS\_PER\_ELEMENT** = 4;

**ELEMENT\_OVERFLOW** = 10000;

**DIGITS\_PER\_ELEMENT** = 5;



```
ELEMENT_OVERFLOW = 100000;
```

```
DIGITS_PER_ELEMENT = 6;
```

```
ELEMENT_OVERFLOW = 1000000;
```

```
DIGITS_PER_ELEMENT = 7;
```

```
ELEMENT_OVERFLOW = 10000000;
```

```
DIGITS_PER_ELEMENT = 8;
```

```
ELEMENT_OVERFLOW = 100000000;
```

```
DIGITS_PER_ELEMENT = 9;
```

```
ELEMENT_OVERFLOW = 1000000000;
```

---

Generated by [PasDoc 0.14.0](#).