

**Binary Coded Decimal**

**The case for a bcd type**

**REFERENCE MANUAL**

*Version: 1.1.0*

*Status: Release*

*Author: ir. W.E. Huisman*

Copyright © 2019, ir. W.E. Huisman

All rights reserved. Published under the MIT software license

Contents

[INTRODUCTION 3](#_Toc27498960)

[The ODBC case 3](#_Toc27498961)

[PREDECESSORS OF BCD 4](#_Toc27498962)

[COBOL and EBCDIC 4](#_Toc27498963)

[4GL programming languages 4](#_Toc27498964)

[The Borland C++ compiler 4](#_Toc27498965)

[The integer-coded-decimal 4](#_Toc27498966)

[The arbitrary-floating-point 4](#_Toc27498967)

[The best of both worlds 4](#_Toc27498968)

[THE BCD DATATYPE 5](#_Toc27498969)

[Introduction 5](#_Toc27498970)

[Construction and initialization 5](#_Toc27498971)

[Constants 5](#_Toc27498972)

[Assignments 5](#_Toc27498973)

[Increments and decrements 6](#_Toc27498974)

[Operators 6](#_Toc27498975)

[Comparisons 6](#_Toc27498976)

[Mathematical functions 6](#_Toc27498977)

[Trigionometric functions 7](#_Toc27498978)

[Conversions 7](#_Toc27498979)

[String display 8](#_Toc27498980)

[File reading and writing 8](#_Toc27498981)

[Information and other methods 9](#_Toc27498982)

[Error handling 9](#_Toc27498983)

[Enhancement and refinements 9](#_Toc27498984)

[BCD AND THE ODBC STANDARD 11](#_Toc27498985)

[The main advantage 11](#_Toc27498986)

[PERFORMANCE MEASURING 12](#_Toc27498987)

[APPENDIX: THE BCD INTERFACE 14](#_Toc27498988)

# INTRODUCTION

Computers calculate numbers in binary. We forget about this many times as the illusion of a mathematical machine is quite compelling, It’s far easier to forget about binary rounding errors and pretend that calculations or precise.

Alas! This doesn’t add up for dull purposes like accounting and bookkeeping. Here a roundoff error of 1 cent can sent an accountant screaming for an explanation and a multi-million dollar investigation. And it is for accounting reasons that databases have special datatypes like NUMERIC and DECIMAL that are precise number formats. This in contrast with approximate data types like FLOAT and REAL.

In the C++ language the built-in datatype “double” (IEEE 754) datatype is in radix 2, so it is also an approximate datatype. Although the Intel x386 processor has something of a ‘bcd’ type of operator to correct the effect of binary calculations, no C++ compiler today exists with a built-in exact numeric datatype. (Borland C++ 2.0 being the last one that had that!)

The solution has been to store these numbers in the so-called “BINARY-CODED-DECIMAL” format. BCD for short. In such an implementation no rounding errors can occur, as extra bits are used to represent a decimal number instead of a binary number.

This lack of a binary-coded-decimal datatype makes it cumbersome to do accounting and book­keeping calculations in C++. Also the storing and retrieving of NUMERIC and DECIMAL numbers to and from databases require lengthy calculations to convert the numbers, requiring precious CPU cycles.

## The ODBC case

The ODBC (Open-DataBase Connectivity) standard has a data structure (SQL\_NUMERIC\_STRUCT) to transport the data for NUMERIC and DECIMAL numbers. Data for these datatypes is often binded and used in ODBC applications as a string. The binary transport of that data in a radix 256 numeric struct is often too great a challenge for most programmers on a daily business schedule.

The chapter about “BCD and the ODBC Standard” explains how this bcd class solves that problem.

# PREDECESSORS OF BCD

There have existed (and still exist!) a number of predecessors to this bcd class.

## COBOL and EBCDIC

The COBOL (COmmon Business Language) is a long existing language and business standard, widely used for accounting and bookkeeping purposes. It’s existence in the ’60, ’70 and ’80 of the last century was mostly in combination with the IBM’s mainframe platforms on EBCDIC character sets, especially suited for accounting in binary-coded-decimal.

## 4GL programming languages

Fourth generation languages bonded to a specific database platform (e.g. INFORMIX-4GL) had a DECIMAL datatype as a built-in feature. Binary coded decimal calculations where the default on this platform. Which made it typically suited to build accounting software and store the results in a database.

## The Borland C++ compiler

Upto the point where Borland sold there C++ compiler to Embarcadero, it had a built in ‘bcd’ datatype that could be directly used, just as you would use an ‘int’ or a ‘double’. To my best knowledge this data­type was dropped in the later versions of this compiler.

Microsoft has never bundled a ‘bcd’ datatype with their implementation of the C++ language. And the language itself, not even through the process of the ISO standard, has ever featured such a datatype. So Borland was unique in this respect.

## The integer-coded-decimal

In order to do exact numerical calculations my first try was to implement a numeric class that stores the numbers in integer format. 4 integers before the decimal point and 4 integers after the decimal point. This implementation implementation – dubbed the integer-coded-decimal – did allow for 16 decimal places before and after the decimal point. Enough to do the bookkeeping of a large multi-natioanl company or a small country 😊

Although the classical operators like adding and subtracting are easy to implement in this format, much was left to be desired. As soon as we want to implement more mathematical operators, this impelementation becomes bothersome, and as we will see later: slow.

## The arbitrary-floating-point

After some searching in the mathematical realm, I found the arbitrary-floating-point (AFP) class of Henrik Vestermark. This library takes the approach of storing the mathematical mantissa and the fractional part seperatly. The fractional part is stored in a character array, and interpreted in each mathematical operation. As such it is a implementation of a precise binary-coded-decimal.

The source code of this library is included in the BCD project. But see also the website of “Numerical Methods at Work” at <http://hvks.com> as this project has evolved since I took the idea from it.

The down side of this library is (apart from performance problems) that the format is not easily transcoded to database formats like NUMERIC and DECIMAL.

## The best of both worlds

Both methods (integer-coded-decimal and arbitrary-floating-point) have led the the resulting design of the binary-coded-decimal class here presented. It stores the exponent and the mantissa seperatly. The mantissa however is stored as a set of integers. In the current configuration 5 integers for 8 decimal positions each is used. Thus allowing for a mantissa of 40 positions. More than enough to even handle the most demanding database implementation (Oracle with 38 positions).

Integers can hold at least 9 decimal places as a positive 32-bit “int” or “long” can hold up to the number of 2.147.483.647. This makes it possible to hold 8 decimal places and still have one digit left to hold any carry or borrow number when iterating over and array of these integers.

# THE BCD DATATYPE

## Introduction

The BCD (Binary-Coded-Decimal) datatype was built with database numeric and decimal datatypes in mind. A binary-coded-decimal number is an EXACT number with no rounding errors due to the binary nature of a computer CPU (Central Processing Unit). This makes the bcd datatype especially suited for financial and bookkeeping purposes.

BCD calculations have been present in computer science for quite some time, and in various forms. This BCD class was especially designed to co-exist with ODBC database adapters. For more information see appendix B.

## Construction and initialization

You can construct a bcd from just about every base C++ datatype, while initializing the bcd at the same time. This goes for chars, integers, longs, 64bit-integers, floats and doubles.

|  |
| --- |
| // Made from an integer and a floating point number  bcd num1(2);  bcd num2(4.0);  bcd num3 = num1 + num2; // will become 6 |

But also for strings and from other bcd number.

|  |
| --- |
| // Made from a string and a different bcd number  bcd num4(“7.25”);  bcd num5(num3);  bcd num6 = num4 + num5; // will become 13.5 |

Here is a complete list of all constructor types:

* The default constructor (initializes the number to zero (‘0.0’))
* Constructed from a char number (-127 to +127)
* Constructed from an unsigned char number (0 upto 255)
* Constructed from a short (-32767 upto 32767)
* Constructed from an unsigned short (0 upto 65535)
* Constructed from an integer (-2147483647 upto 2147483647)
* Constructed from an unsigned integer (0 upto 4294967295)
* Constructed from a 64bits integer (-9223372036854775807 upto 9223372036854775807)
* Constructed from an unsigned 64bits integer (0 upto 18446744073709551615)
* Constructed from another bcd
* Constructed from a string of type CString (MFC)
* Constructed from a string of type “const char \*”
* Constructed from a SQL\_NUMERIC\_STRUCT (as appearing in the ODBC standard)

## Constants

There are three defined constants in the bcd datatype. Theses constants are:

* PI The well known circle/radius ratio
* LN2 The natural logarithm of 2
* LN10 The logarithm of 10

They appear as “const bcd” numbers and can be used as such. Here are a few simple examples to show there use.

|  |
| --- |
| // Numbers made up with the help of constants  bcd ratio = 2 \* PI();  bcd quart = bcd::PI() / bcd(2); |

## Assignments

Other bcd’s, integers, doubles and strings can be assigned to a bcd number. That is: you can use the standard ‘=’ assignment operator or the operators that are combined with the standard mathematical operations ‘+=’, ‘-=’, ‘\*=’, ‘/=’ and ‘%=’. Assignment operators made of a combination with bitwise operators like ‘|=’ or ‘&=’ have no logical counterpart in the bcd class, as a bitwise operation has no logical meaning for a binary coded decimal number.

|  |
| --- |
| // Calculation with assignments  bcd a = SomeFunc();  a += 2;  a \*= b; // a = b + (2 \* SomeFunc) |

## Increments and decrements

Both prefix and postfix increments and decrements can be used with a bcd just as with any integer number.

|  |
| --- |
| // Calculation with prefix and postfix in- and decrements  bcd a = ++b;  bcd c = a--; |

## Operators

The standard mathematical operators ‘+’ (addition), ‘-‘ (subtraction), ‘\*’ (multiplication), ‘/’ (division) and ‘%’ (modulo) are implemented for the bcd class.

As this class is designed to do bookkeeping, it is the ‘bread-and-butter’ of this class. More than eighty percent of all calculations are done in these operators in real-world applications.

Here is a typical example:

|  |
| --- |
| // Finding an average price from a std::vector of objects  // Where GetPrice() and GetVAT() both return a bcd value  bcd total;  for(auto& obj : objectlist)  {  total += obj.GetPrice() + obj.GetVAT();  }  bcd average = total / bcd(objectlist.count());  if(average > bcd(400.0))  {  ReportAverageToHigh(average);  } |

## Comparisons

All typical comparison operators like equal (==), not-equal (!=), smaller (<), smaller-than-or-equal-to (<=), greater (>) and greater-than-or-equal-to (>=) are implemented for the bcd class.

For an example, see the previous paragraph where we report an average that is too high.

## Mathematical functions

The C library contains a number of mathematical functions that are solely implemented in the ‘double’ basic datatype. An example of these is e.g. “pow” for the taking of a power. These functions are implemented as statical functions with bcd’s as parameters. Static mathematical functions include:

Apart from the static functions, they are also implemented as methods of the bcd class. So “pow” has a symeterical method ‘Power’.

|  |  |  |
| --- | --- | --- |
| Math function | Static function | Method of bcd |
| Integer floor of a number | bcd floor(bcd); | bcd Floor() const; |
| Ceiling of a number | bcd ceil(bcd); | bcd Ceiling() const; |
| Absolute value of a number | bcd fabs(bcd); | bcd AbsoluteValue() const; |
| Square root of a number | bcd sqrt(bcd); | bcd SquareRoot() const; |
| 10logarithm of a number | bcd log10(bcd); | bcd Log10() const |
| Natural logarithm of a number | bcd log(bcd); | bcd Log() const; |
| Exponent ‘e’ to this number | bcd exp(bcd); | bcd Exp() const; |
| Power of a number | bcd pow(bcd,bcd); | bcd Power(const bcd& p\_power) const |
| Getting the fractional part | bcd frexp(bcd,int\*) | bcd GetMantisse() const; |
| Integer times 2 to the power | bcd ldexp(bcd,int); | Use: ‘integer \* bcd(2).Power(bcd)’ |
| Fraction part and integer part | bcd modf(bcd,int \*); | bcd Fraction() const; |
| Modulo | bcd fmod(bcd,bcd); | The ‘%’ operator  bcd Mod(bcd) const; |

Here are two examples that do exactly the same:

|  |
| --- |
| // Calculate the side of a square  bcd surface = GetSquareSurfaceArea();  bcd side = surface.SquareRoot(); |

|  |
| --- |
| // Calculate the side of a square  bcd surface = GetSquareSurfaceArea();  bcd side = sqrt(surface); |

NOTE: Overloading the mathematical functions make is meant to make it easier to port existing code with doubles to be converted into bcd’s.

## Trigionometric functions

The standard C trigionometric functions are overloaded for the bcd class as is the case with the standard mathematical functions. Just as with the standard trigionometric functions the number is an angle measured in radians. The following functions exist:

|  |  |  |
| --- | --- | --- |
| Trigionometric function | Static function | Method of bcd |
| Sine of an angle | bcd sin(bcd); | bcd Sine() const; |
| Cosine of an angle | bcd cos(bcd); | bcd Cosine() const; |
| Tangent of an angle | bcd tan(bcd); | bcd Tangent() const; |
| Getting an angle from a sine ratio | bcd asin(bcd); | bcd ArcSine() const; |
| Getting an angle from a cosine ratio | bcd acos(bcd); | bcd ArcCosine() const; |
| Getting an angle from a tangent ratio | bcd atan(bcd); | bcd ArcTangent() const; |
| Tangent from 2 points | bcd atan2(bcd,bcd); | bcd ArcTangent2Points(bcd) const; |

Here are two examples that do exactly the same:

|  |
| --- |
| // Calculate the height of a given wave  bcd waveHeight = GetSignal().Sine(); |

|  |
| --- |
| // Calculate the height of a given wave  bcd waveHeight = sin(GetSignal()); |

## Conversions

It is possible to convert a bcd to ‘something-else’. This other ‘something’ is a base datatype from the C++ language. Most methods are named something like “AsXXXX”, where XXXX denotes the type we want. The following methods exist:

|  |  |
| --- | --- |
| Conversion method | The result |
| AsDouble() const; | Returns the number as a ‘double’, or throws an error if the bcd number is ‘out-of-bounds’ for a double |
| AsShort() const; | Returns a ‘short’ integer or throws an error if the bcd number is ‘out-of-bounds’ for a short |
| AsUShort() const; | Returns an ‘unsigned-short’ integer or throws an error if the bcd number is ‘out-of-bounds’ for an unsigned short |
| AsLong() const; | Returns a ‘long’ integer or throws an error if the bcd number is ‘out-of-bounds’ for a long |
| AsULong() const; | Returns a ‘unsigned-long’ integer or throws an error if the bcd number is ‘out-of-bounds’ for an unsigned long |
| AsInt64() const; | Returns a 64 bit integer or throws an error if the bcd number is ‘out-of-bounds’ for a 64 bit integer |
| AsUInt64() const; | Returns an unsigned 64 bit integer or throws an error if the bcd number is ‘out-of-bounds’ for a 64 bit integer |
| AsString(int, bool) const; | Returns a string. Parameters control the formatting of the string (bookkeeping format and +/- printing) |
| AsDisplayString(int) const; | Returns a string. Uses the current locale to print a ‘displayable’ string with correct delimiters. The parameter controls the number of decimal places |
| AsNumeric(SQL\_NUMERIC\_STRUCT\*) const; | Returns the bcd number in a ODBC numeric structure. For use with ODBC drivers and compatible programs. |

Here is an example of a calculation returing an engineering number string in 10 exponential format.

|  |
| --- |
| // Return a calculation in a IEEE number string  CString GetCalculation()  {  bcd number1 = SomeFunction();  bcd number2 = AnotherFunction();  bcd number3 = number1.Power(number2);  // Something like “5.6773E-03”  return number3.AsString(Engineering,false);  } |

## String display

Numbers can be displayed as strings. How they are displayed depends on the application we are using the number in. This can be quite different for a scientific or engineering application in contrast to a bookkeeping application. The differences are loosly defined as:

1. In bookkeeping applications, we tend to display numbers with one decimal marker and as much thousand parts markers as needed. We display decimal places upto a defined amount and round of the rest of the decimal places;
2. In engineering applications, we tend to print the exact number just with one decimal marker. If the number gets to great (or to small) we shift to exponential display in powers of ten.
3. In both cases we always print a negative number with a negative sign (-), but we can choose to print the positive sign (+) as well;
4. The decimal and thousand markers are defined by the current system local of the machine and thus the language the desktop is currently using.

Here are a few examples of both ways of displaying a number:

|  |  |  |
| --- | --- | --- |
| The number | Bookkeeping | Engineering |
| 12.3456 | 12.34 | 12.3456 |
| 123456.789123 | + 123,456.79 | 123456.789123 |
| 123456789.12345 | 123,456,789.12 | 1.2345678012345E+08 |
| -0.00012345 | - 0.00 | -1.2345E-04 |
| -1234567890123456 | - 1.123.567.890.123.456 (Dutch, German, French) | -1.234567890123456E+15 |

## File reading and writing

Applications might need to write information to a binary file. So there are two methods for integration with binary files. The first (WriteToFile(FILE\*)) writes the bcd number to a file. The second (ReadFromFile(FILE\*)) reads the bcd number back from that file. All primary factual information of the bcd number is stored.

Any disturbance in the force (oh sorry: the file) will lead to an error, meaning the whole number gets stored or read back, or an error occurs. See the implementation for more details about the storing format of the bcd.

Storage and retrieval of the bcd number in the file is also network independent and litte-big-endian independent, meaning you can store and retrieve the number in a portable way.

## Information and other methods

A number of methods exist that have not yet been discussed. The give information of some property of the bcd number or do a basic operation. Here is the remainder list:

|  |  |
| --- | --- |
| Method | Documentation |
| bool IsNull() const; | Only returns ‘true’ if the bcd number is exactly 0.0 (zero) |
| int GetSign() const; | Retrieves a -1, 0 or 1 as the sign of the number |
| int GetLength() const; | Retrieves to total of decimal places before and after the decimal point in a total |
| int GetPrecision() const | Retrieves the decimal places BEHIND the decimal point |
| Int GetMaxSize() const; | Constant denotes the maximum of decimal places. 40 in the current implementation! |
| bool GetFitsInLong() const; | True if the bcd number will fit into a long without throwing an error exception |
| bool GetFitsInInt64() const; | True if the bcd number will fit into a 64 bits integer without throwing an error exception |
| bool GetHasDecimals() const; | True if there are decimal places behind the decimal marker |
| int GetExponent() const; | Returns the current exponent of the number |
| bcd GetsMantissa() const; | Only retrieves the mantissa part of the bcd of a number |
| void Zero() | Set the bcd number back to 0.0 (zero) |
| void Round(int precision); | Round of the bcd number on this number of decimal places |
| void Truncate(int precision); | Trunacte the bcd number on this number of decimal places |
| void Negate(); | Negate the sign of a number |

## Error handling

Error handling is done by throwing a StdException. This exception is integrated with the MS-Windows C++ Safe Exception Handling in such a way that critical errors like null-pointer references and division-by-zero errors do \*\*NOT\*\* get an different exception handling – stopping the application e.g. – but are integrated in the exception throwing.

Here is a list of all errors in the bcd class. There descriptions are meant to be self-explanatory:

|  |
| --- |
| StdException |
| BCD: Cannot get a square root from a negative number |
| BCD: Cannot calculate a natural logarithm of a number <= 0 |
| BCD: Cannot get a 10-logarithm of a number <= 0 |
| BCD: Cannot calculate a tangent from a angle of 1/2 pi or 3/2 pi |
| BCD: Cannot calculate an arcsine from a number > 1 or < -1 |
| BCD: Overflow in conversion to short number. |
| BCD: Underflow in conversion to short number. |
| BCD: Cannot convert a negative number to an unsigned short number. |
| BCD: Overflow in conversion to unsigned short number. |
| BCD: Overflow in conversion to integer number. |
| BCD: Underflow in conversion to integer number. |
| BCD: Cannot convert a negative number to an unsigned long. |
| BCD: Overflow in conversion to unsigned long integer. |
| BCD: Overflow in conversion to 64 bits integer number. |
| BCD: Cannot convert a negative number to an unsigned 64 bits integer |
| BCD: Overflow in conversion to 64 bits unsigned integer number. |
| BCD: Overflow in converting bcd to SQL NUMERIC/DECIMAL |
| BCD: Conversion from string. Bad format in decimal number |
| BCD: Division by zero. |

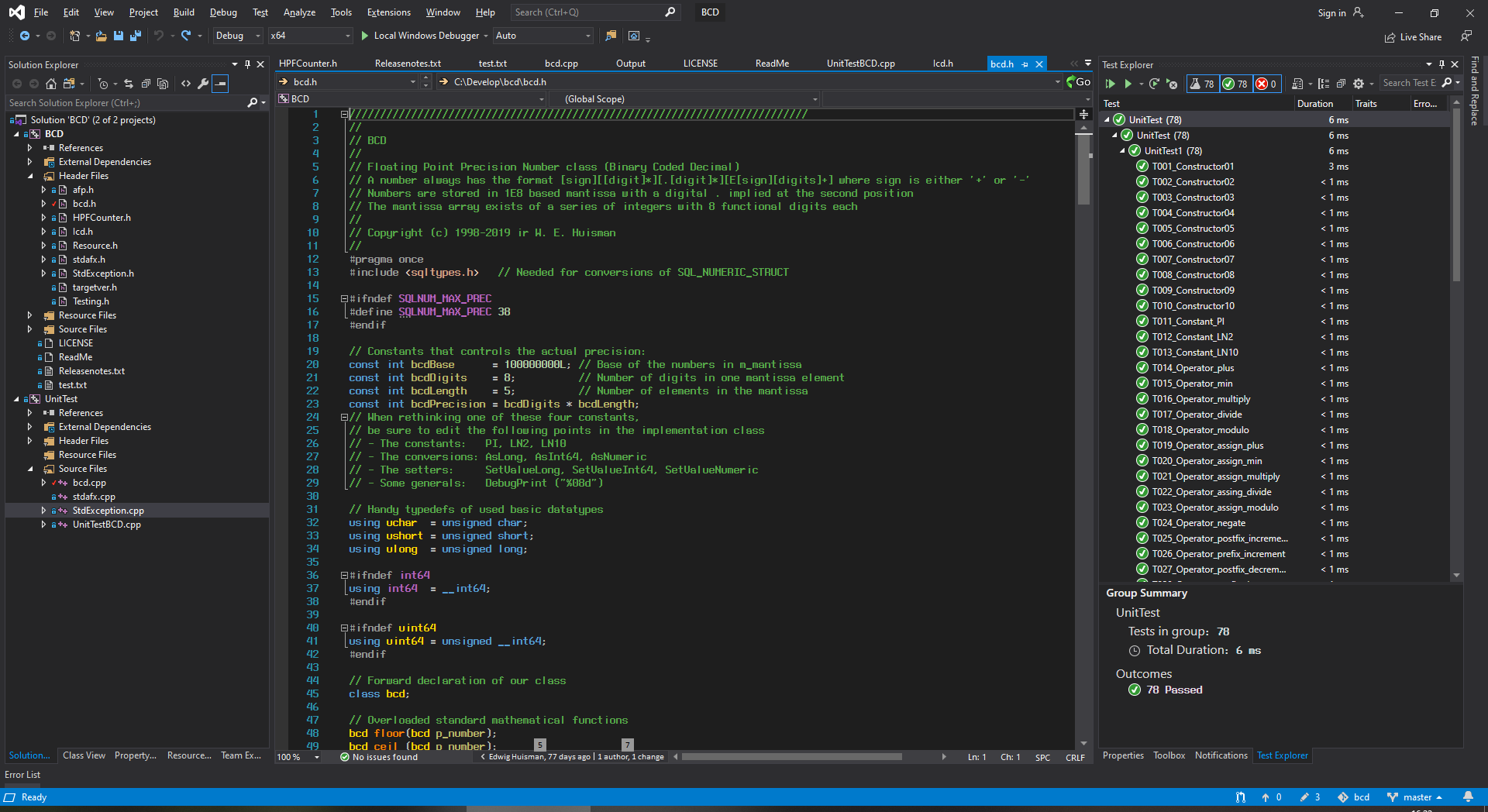
## Enhancement and refinements

The bcd class can easily be enhanced. You can simply expand the number of digits in the mantissa by using a greater number on integers in the mantissa array. See the constants “bcdDigits” and “bcdLength” at the beginning of the class interface definition.

Extra methods and / or data, operators, stream interfaces like std::iostream can easily be added to this class.

The bcd project comes with a unit test module dll. Goal of the unit test is of course to test the correct workings of the other functionality while you expand the class.

Just open the test explorer in Visual Studio (from the menu “Test” / “Run all tests”) and check that all unit test are ‘in-the-green’.



# BCD AND THE ODBC STANDARD

Now with the mathematical calculations firmly in place, we can turn to the usage of the bcd number in combination with the ODBC drivers. Binary data flows to and from the ODBC driver of a database in the form of the SQL\_NUMERIC\_STRUCT. This struct supports both the NUMERIC and DECIMAL datatypes of a modern ISO:9075 compliant SQL database.

Binding directly to a DECIMAL or NUMERIC column in a database query will result in the retrieval of a SQL\_NUMERIC\_STRUCT in memory. Changing and using that struct in an update or insert statement will use the contents of that struct and transport it to our database record.

But what happens when we get that data. Peeking at the source code of some open-source database implementations like the MySql, MariaDB, Firebird or PostgreSQL reveals that most odbc drivers just convert a string to a SQL\_NUMERIC\_STRUCT. At the moment that this dataclass was written those conversions where quite complicated, long and error prone. In the last years the situation has improved, but…

A lot of confusion still exists in the usage of NUMERIC and DECIMAL. Looking up answers on the stackoverflow platform, even experienced programmers opt-in to let the database convert the data to a string and plucking that string data out of the query.

The BCD class has been designed to easily convert to and from the SQL\_NUMERIC\_STRUCT. With the following two methods:

* bcd::SetValueNumeric(SQL\_NUMERIC\_STRUCT\*);
* bcd::AsNumeric(SQL\_NUMERIC\_STRUCT\*);

Data is directly converted to and from the ODBC bind area and to the database. These conversions are a simple iteration over the mantissa and copy of the mantissa and sign bit.

## The main advantage

The key factor here is that we can directly use our NUMERIC and DECIMAL numbers without having them to convert first to a string and back to a format where we can begin calculations in them. Round-about the other direction: we can directly calculate and store the result in the database without having to convert everything to strings and order the database to convert it back to exactly the same data again!

# PERFORMANCE MEASURING

In order to be able to measure the performance of my implementions, I designed a test program that does any number of calculations a configurable number of times ‘n’. When setting ‘n’ to for instance a 1000 times, the length of the calculations will be great enough to be able to measure it with a high performance counter like “[QueryPerformanceCounter](https://docs.microsoft.com/en-us/windows/win32/api/profileapi/nf-profileapi-queryperformancecounter)” of the MS-Windows kernel.

The test program compares the result of each operation with the result of the MS-Windows desktop calculator “calc.exe” and shows the performance results of four implementations:

1. C++ built-in “double”
2. Arbitrary-floating-point
3. Integer-coded-decimal
4. Binary-coded-decimal

A typical output of the test program looks like:

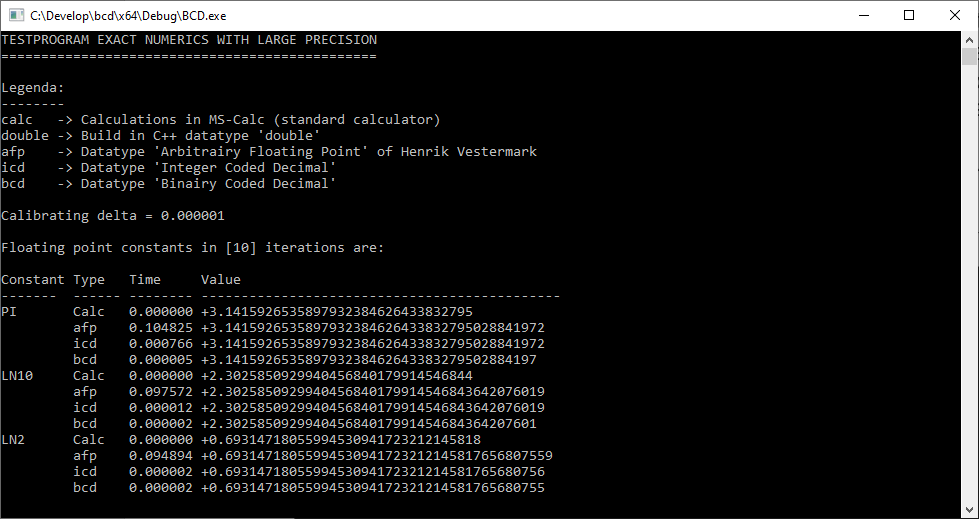
|  |
| --- |
| Testing the function [log10] for a total of [1000] iterations:  Input: 98765432109876543210.123456789012345678901234567890  Type Time Value  ------ ---------- ------------------------------------------------------  calc 0.000000 +19.994604968162151965673558368195  double 0.000005 +19.994604968162150  afp 0.982142 +19.99460496816215196567355836819543212297  icd 0.191501 +19.9946049681621519656735583681954349795885  bcd 0.050899 +19.9946049681621519656735583681954321229 |

In this example we see the results for the “log10” function (logarithm in base 10). As we can see, the result is correct upto at least 32 decimal places for each implementation (apart from ‘double’ ☹)

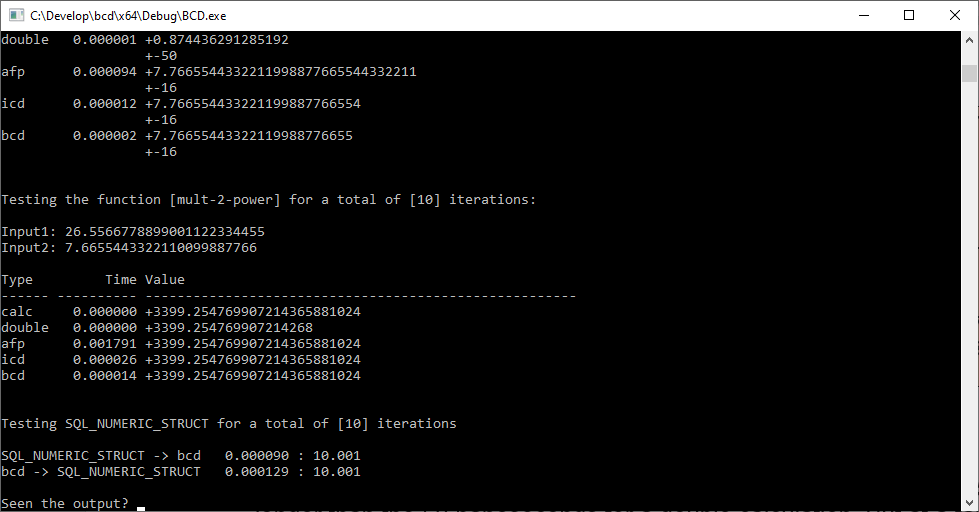
A thousand iterations take 0,05 seconds in the bcd implementation: 50 microseconds each. Quite a bit longer than the 50 nanoseconds for a double calculation. But at a far greater precision!

The BCD solution in the example program runs this test by default.

This is a screenshot of the beginning of the testrun:



And here is a sample of the output at the end of the testrun:



# APPENDIX: THE BCD INTERFACE

Here is the complete interface definition in the C++ language:

//////////////////////////////////////////////////////////////////////////

//

// BCD

//

// Floating Point Precision Number class (Binary Coded Decimal)

// A number always has the format [sign][[digit]\*][.[digit]\*][E[sign][digits]+]

// where sign is either '+' or '-'

// Numbers are stored in 1E8 based mantissa with a digital . implied at the

// second position

// The mantissa array exists of a series of integers with 8 functional digits each

//

// Copyright (c) 1998-2019 ir W. E. Huisman

//

#pragma once

#include <sqltypes.h> // Needed for conversions of SQL\_NUMERIC\_STRUCT

#ifndef SQLNUM\_MAX\_PREC

#define SQLNUM\_MAX\_PREC 38

#endif

// Constants that controls the actual precision:

const int bcdBase = 100000000L; // Base of the numbers in m\_mantissa

const int bcdDigits = 8; // Number of digits in one mantissa element

const int bcdLength = 5; // Number of elements in the mantissa

const int bcdPrecision = bcdDigits \* bcdLength;

// When rethinking one of these four constants,

// be sure to edit the following points in the implementation class

// - The constants: PI, LN2, LN10

// - The conversions: AsLong, AsInt64, AsNumeric

// - The setters: SetValueLong, SetValueInt64, SetValueNumeric

// - Some generals: DebugPrint ("%08d")

// Handy typedefs of used basic datatypes

using uchar = unsigned char;

using ushort = unsigned short;

using ulong = unsigned long;

#ifndef int64

using int64 = \_\_int64;

#endif

#ifndef uint64

using uint64 = unsigned \_\_int64;

#endif

// Forward declaration of our class

class bcd;

// Overloaded standard mathematical functions

bcd floor(bcd p\_number);

bcd ceil (bcd p\_number);

bcd fabs (bcd p\_number);

bcd sqrt (bcd p\_number);

bcd log10(bcd p\_number);

bcd log (bcd p\_number);

bcd exp (bcd p\_number);

bcd pow (bcd p\_number,bcd p\_power);

bcd frexp(bcd p\_number,int\* p\_exponent);

bcd ldexp(bcd p\_number,int p\_power);

bcd modf (bcd p\_number,bcd\* p\_intpart);

bcd fmod (bcd p\_number,bcd p\_divisor);

// Overloaded standard trigonometric functions on a bcd number

bcd sin (bcd p\_number);

bcd cos (bcd p\_number);

bcd tan (bcd p\_number);

bcd asin (bcd p\_number);

bcd acos (bcd p\_number);

bcd atan (bcd p\_number);

bcd atan2(bcd p\_y,bcd p\_x);

//////////////////////////////////////////////////////////////////////////

//

// The Binary Coded Decimal class

//

//////////////////////////////////////////////////////////////////////////

class bcd

{

public:

typedef enum \_sign { Positive, Negative } Sign;

typedef enum \_format { Engineering, Bookkeeping } Format;

typedef enum \_operator{ Addition, Subtraction } Operator;

// CONSTRUCTORS/DESTRUCTORS

// Default constructor.

bcd();

// BCD from a char value

bcd(const char p\_value);

// BCD from an unsigned char value

bcd(const unsigned char p\_value);

// BCD from a short value

bcd(const short p\_value);

// BCD from an unsigned short value

bcd(const unsigned short p\_value);

// BCD from an integer

bcd(const int p\_value);

// BCD from an unsigned integer

bcd(const unsigned int p\_value);

// BCD from a long

bcd(const long p\_value, const long p\_restValue = 0);

// BCD from an unsigned long

bcd(const unsigned long p\_value, const unsigned long p\_restValue = 0);

// BCD from a 64bits int

bcd(const int64 p\_value,const int64 p\_restvalue = 0);

// BCD from an unsigned 64bits int

bcd(const uint64 p\_value,const int64 p\_restvalue = 0);

// Copy constructor.

bcd(const bcd& icd);

// BCD from a float

bcd(const float p\_value);

// BCD from a double

bcd(const double p\_value);

// Assignment-constructor of class bcd.

bcd(const CString& p\_string,bool p\_fromDB = false);

// BCD From a character string

bcd(const char\* p\_string,bool p\_fromDB = false);

// BCD from a SQL\_NUMERIC\_STRUCT

bcd(const SQL\_NUMERIC\_STRUCT\* p\_numeric);

// Destructor of class bcd.

~bcd();

// CONSTANTS

static bcd PI(); // Circumference/Radius ratio of a circle

static bcd LN2(); // Natural logarithm of 2

static bcd LN10(); // Natural logarithm of 10

// OPERATORS

// Standard mathematical operators

const bcd operator+(const bcd& p\_value) const;

const bcd operator-(const bcd& p\_value) const;

const bcd operator\*(const bcd& p\_value) const;

const bcd operator/(const bcd& p\_value) const;

const bcd operator%(const bcd& p\_value) const;

// Standard math/assignment operators

bcd& operator+=(const bcd& p\_value);

bcd& operator-=(const bcd& p\_value);

bcd& operator\*=(const bcd& p\_value);

bcd& operator/=(const bcd& p\_value);

bcd& operator%=(const bcd& p\_value);

// Prefix unary minus (negation)

bcd operator-() const;

// Prefix/Postfix increment/decrement operators

bcd operator++(int); // Postfix increment

bcd& operator++(); // Prefix increment

bcd operator--(int); // Postfix decrement

bcd& operator--(); // Prefix decrement

// Assignment operators

bcd& operator=(const bcd& p\_value);

bcd& operator=(const long p\_value);

bcd& operator=(const double p\_value);

bcd& operator=(const CString& p\_value);

// comparison operators

bool operator==(const bcd& p\_value) const;

bool operator!=(const bcd& p\_value) const;

bool operator< (const bcd& p\_value) const;

bool operator> (const bcd& p\_value) const;

bool operator<=(const bcd& p\_value) const;

bool operator>=(const bcd& p\_value) const;

// MAKING AN EXACT NUMERIC value

// Set the mantissa/exponent/sign to the number zero (0)

void Zero();

// Round to a specified fraction (decimals behind the .)

void Round(int p\_precision = 0);

// Truncate to a specified fraction (decimals behind the .)

void Truncate(int p\_precision = 0);

// Change the sign

void Negate();

// MATHEMATICAL FUNCTIONS

// Value before the decimal point

bcd Floor() const;

// Value behind the decimal point

bcd Fraction() const;

// Value after the decimal point

bcd Ceiling() const;

// Square root of the bcd

bcd SquareRoot() const;

// This bcd to the power x

bcd Power(const bcd& p\_power) const;

// Absolute value (ABS)

bcd AbsoluteValue() const;

// Reciproke / Inverse = 1/x

bcd Reciproke() const;

// Natural logarithm

bcd Log() const;

// Exponent e tot the power 'this number'

bcd Exp() const;

// Log with base 10

bcd Log10() const;

// Ten Power

bcd TenPower(int n);

// TRIGONOMETRIC FUNCTIONS

// Sinus of the angle

bcd Sine() const;

// Cosine of the angle

bcd Cosine() const;

// Tangent of the angle

bcd Tangent() const;

// Arc sines (angle) of the ratio

bcd ArcSine() const;

// Arc cosine (angle) of the ratio

bcd ArcCosine() const;

// Arctangent (angle) of the ratio

bcd ArcTangent() const;

// Angle of two points (x,y)

bcd ArcTangent2Points(bcd p\_x) const;

// GET AS SOMETHING DIFFERENT

// Get as a double

double AsDouble() const;

// Get as a short

short AsShort() const;

// Get as an unsigned short

ushort AsUShort() const;

// Get as a long

long AsLong() const;

// Get as an unsigned long

ulong AsULong() const;

// Get as a 64bits long

int64 AsInt64() const;

// Get as an unsigned 64 bits long

uint64 AsUInt64() const;

// Get as a mathematical string

CString AsString(int p\_format = Bookkeeping,bool p\_printPositive = false) const;

// Get as a display string (by desktop locale)

CString AsDisplayString(int p\_decimals = 2) const;

// Get as an ODBC SQL NUMERIC(p,s)

void AsNumeric(SQL\_NUMERIC\_STRUCT\* p\_numeric) const;

// GETTER FUNCTIES

// Is bcd exactly 0.0?

bool IsNull() const;

// Gets the sign 0 (= 0.0), 1 (greater than 0) of -1 (smaller than 0)

int GetSign() const;

// Total length (before and after decimal point)

int GetLength() const;

// Total precision (length after the decimal point)

int GetPrecision() const;

// Get the max size of a bcd

static int GetMaxSize(int p\_precision = 0);

// Gets the fact that it fits in a long

bool GetFitsInLong() const;

// Gets the fact that it fits in an int64

bool GetFitsInInt64() const;

// Decimal part (behind the decimal point) is not "000" (zeros)

bool GetHasDecimals() const;

// Gets the exponent

int GetExponent() const;

// Gets the mantissa

bcd GetMantissa() const;

// FILE STREAM FUNCTIONS

bool WriteToFile (FILE\* p\_fp);

bool ReadFromFile(FILE\* p\_fp);

private:

// INTERNALS

// Sets one integer in this bcd number

void SetValueInt(const int p\_value);

// Sets one or two longs in this bcd number

void SetValueLong(const long p\_value, const long p\_restValue);

// Sets one or two longs in this bcd number

void SetValueInt64 (const int64 p\_value,const int64 p\_restValue);

void SetValueUInt64(const uint64 p\_value,const int64 p\_restValue);

// Sets the value from a double

void SetValueDouble(const double p\_value);

// Sets the value from a string

void SetValueString(const CString& p\_string,bool p\_fromDB = false);

// Sets the value from a SQL NUMERIC

void SetValueNumeric(const SQL\_NUMERIC\_STRUCT\* p\_numeric);

// Take the absolute value of a long

long long\_abs(const long p\_value) const;

// Normalize the mantissa/exponent

void Normalize(int p\_startExponent = 0);

// Multiply the mantissa by 10

void Mult10(int p\_times = 1);

// Divide the mantissa by 10

void Div10(int p\_times = 1);

// Shift mantissa 1 position right

void ShiftRight();

// Shift mantissa 1 position left

void ShiftLeft();

// Convert a string to a single long value

long StringToLong(CString& p\_string) const;

// Convert a long to a string

CString LongToString(long p\_value) const;

// Split the mantissa for floor/ceiling operations

bcd SplitMantissa() const;

// Compare two mantissa

int CompareMantissa(const bcd& p\_value) const;

// Debug print of the mantissa

void DebugPrint(char\* p\_name);

// Stopping criterion for internal iterations

bcd& Epsilon(long p\_fraction) const;

// BASIC OPERATIONS

// Addition operation

bcd Add(const bcd& p\_number) const;

// Subtraction operation

bcd Sub(const bcd& p\_number) const;

// Multiplication

bcd Mul(const bcd& p\_number) const;

// Division

bcd Div(const bcd& p\_number) const;

// Modulo

bcd Mod(const bcd& p\_number) const;

// Helpers for the basic operations

// Position arguments and signs for the next operation

void PositionArguments(bcd& arg1,bcd& arg2,Sign& signResult,Operator& operatorKind) const;

// Calculate the sign for multiplication or division

Sign CalculateSign(const bcd& p\_arg1, const bcd& p\_arg2) const;

// Addition of two mantissa (no signs/exponents)

bcd PositiveAddition(bcd& arg1,bcd& arg2) const;

// Subtraction of two mantissa (no signs/exponents)

bcd PositiveSubtraction(bcd& arg1,bcd& arg2) const;

// Multiplication of two mantissa (no signs)

bcd PositiveMultiplication(const bcd& p\_arg1,const bcd& p\_arg2) const;

// Division of two mantissa (no signs)

bcd PositiveDivision(bcd& p\_arg1,bcd& p\_arg2) const;

// STORAGE OF THE NUMBER

Sign m\_sign; // 0 = Positive, 1 = Negative

short m\_exponent; // +/- 10E32767

long m\_mantissa[bcdLength]; // Up to (bcdDigits \* bcdLength) digits

};