DECIMAL	BINARY	BINARY CODE DECIMAL 8 4 2 1
0	0000	0000
1	0001	0001
2	0010	0010
3	0011	0011
4	0100	0100
5	0101	0101
6	0110	0110
7	0111	0111
8	1000	1000
9	1001	1001
10	1010	0001 0000
11	1011	0001 0001
12	1100	0001 0010
13	1101	0001 0011
14	1110	0001 0100
15	1111	0001 0101

Binary Coded Decimal

The case for a bcd type

REFERENCE MANUAL

Version: 1.2.0
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
The ODBC case 3
PREDECESSORS OF BCD 4
COBOL and EBCDIC 4
4GL programming languages 4
The Borland C++ compiler 4
The integer-coded-decimal 4
The arbitrary-floating-point 4
The best of both worlds 4
THE BCD DATATYPE 5
Introduction 5
Construction and initialization 5
Constants 5
Assignments 5
Increments and decrements 6
Operators 6
Comparisons 6
Mathematical functions 6
Trigionometric functions 7
Conversions 7
String display 8
File reading and writing 8
Information and other methods 9
Error handling 9
Enhancement and refinements 9
BCD AND THE ODBC STANDARD 11
The main advantage 11
SQLComponents 11
Open ODBC Querytool 11
PERFORMANCE MEASURING 12

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 book-keeping 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 the chapter “BCD and the ODBC Standard”

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	-1.123.567.890.123.456,00 (in 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.

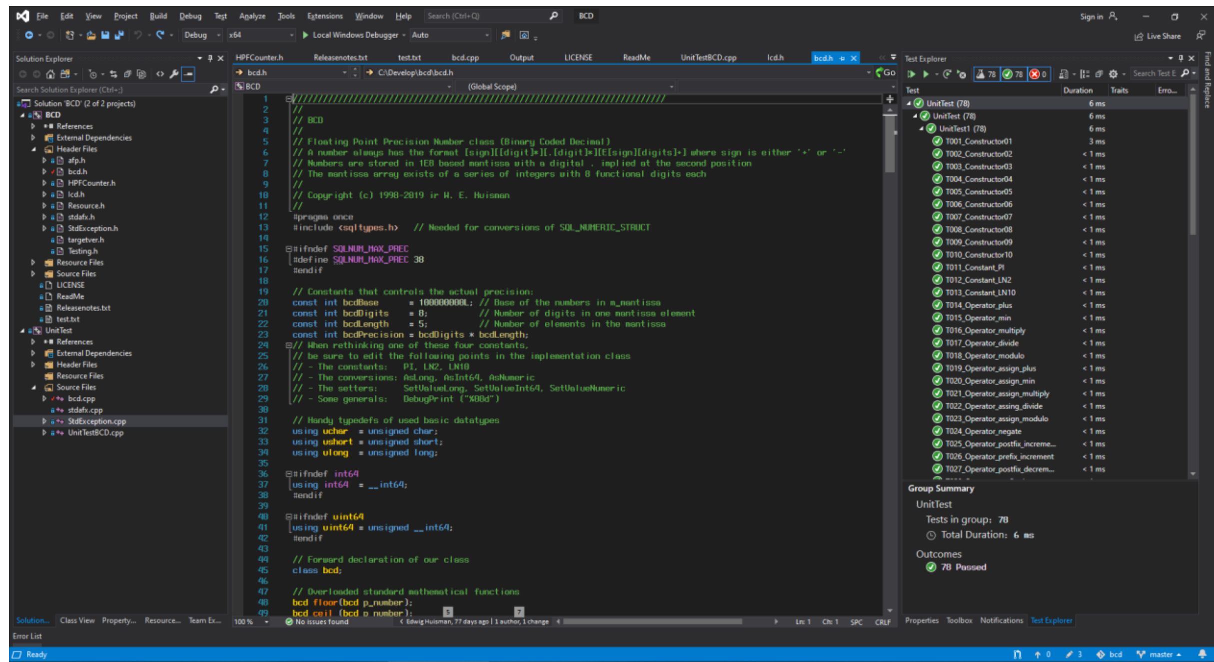
Here is a list of all errors in the bcd class. There descriptions are meant to be self-explanatory:

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…

1) These conversion only convert the standard number format with a decimal point. No exponential numbers can be converted;
2) 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!

SQLComponents

The main application of the bcd class lies within the SQLComponents library. This is a library around the ODBC driver. You can find this library at: https://github.com/edwig/SQLComponents.
In this library, all datarows are bound to a SQLRecord object. The columns of each record in turn are bound to the SQLVariant class.

The SQLVariant class acts as a sorts of variable placeholder for all datatypes that can be obtained from a database row. And of course: one of the datatypes is the bcd class.

The SQLComponents database makes it easier to program with any given ODBC driver. It has been tested with Oracle, MS-SQLServer, MySQL, PostgreSQL, MS-Access and IBM-Informix.

Open ODBC Querytool

Apart from a number of business applications, the one and only killer-app that's using the SQLComponents and bcd class is the Open ODBC-Querytool. You can find this querytool through github on: https://github.com/edwig/ODBCQueryTool and on sourceforge under the following link: https://sourceforge.net/projects/odbcquerytool/. From this last link it has seen more than 50.000 downloads in the last years.

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” 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: 98765432109876543210.12345678901234567890123456789 0

Type         Time Value
------ ---------- -------------------------------------------------------
calc     0.000000 +19.994604968162151965673558368195
double   0.000005 +19.994604968162150
afp      0.982142 +19.99460496816215196567355836819543212297
icd      0.191501 +19.994604968162151965673558368195434979588 5
bcd      0.050899 +19.99460496816215196567355836819543212 29
```

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:

```
C:\Develop\bcd\x64\Debug\BCD.exe

TESTPROGRAM EXACT NUMERICS WITH LARGE PRECISION
===============================================

Legenda:
--------
calc   -> Calculations in MS-Calc (standard calculator)
double -> Build in C++ datatype 'double'
afp    -> Datatype 'Arbitrairy Floating Point' of Henrik Vestermark
icd    -> Datatype 'Integer Coded Decimal'
bcd    -> Datatype 'Binairy Coded Decimal'

Calibrating delta = 0.000001

Floating point constants in [10] iterations are:

Constant Type   Time     Value
-------- ------ -------- ---------------------------------------------
PI       Calc   0.000000 +3.14159265358979323846264338327 95
         afp    0.104825 +3.1415926535897932384626433832795028841972
         icd    0.000766 +3.1415926535897932384626433832795028841972
         bcd    0.000005 +3.141592653589793238462643383279502884197
LN10     Calc   0.000000 +2.3025850929940456840179914546844
         afp    0.097572 +2.3025850929940456840179914546843642076019
         icd    0.000012 +2.3025850929940456840179914546843642076019
         bcd    0.000002 +2.302585092994045684017991454684364207601
LN2      Calc   0.000000 +0.69314718055994530941723212145818
         afp    0.094894 +0.6931471805599453094172321214581765680755 9
         icd    0.000002 +0.6931471805599453094172321214581765680756
         bcd    0.000002 +0.6931471805599453094172321214581765680755
```

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

```
C:\Develop\bcd\x64\Debug\BCD.exe
double   0.000001 +0.874436291285192
                  +-50
afp      0.000094 +7.7665544332211998877665544332211
                  +-16
icd      0.000012 +7.76655443322119988776655 4
                  +-16
bcd      0.000002 +7.766554433221199887766 55
                  +-16


Testing the function [mult-2-power] for a total of [10] iterations:

Input1: 26.5566778899001122334455
Input2: 7.66554433221100998877 66

Type         Time Value
------ ---------- ------------------------------------------------------
calc     0.000000 +3399.2547699072143658810 24
double   0.000000 +3399.254769907214268
afp      0.001791 +3399.2547699072143658810 24
icd      0.000026 +3399.2547699072143658810 24
bcd      0.000014 +3399.2547699072143658810 24


Testing SQL_NUMERIC_STRUCT for a total of [10] iterations

SQL_NUMERIC_STRUCT -> bcd   0.000090 : 10.001
bcd -> SQL_NUMERIC_STRUCT   0.000129 : 10.001

Seen the output? _
```