

PICAXE Calculator

J. Leach 2006



Author: Jeremy Leach Email: ukc802139700@btconnect.com

June 2006

Introduction

The PICAXE Calculator is an experimental floating-point calculator implemented in code. It is only suitable for PICAXE 18x or higher as it requires around 1100 bytes of program code.

PICAXE BASIC works with integer arithmetic, the largest integer being 65535 (a 2-byte Word value). If floating-point arithmetic is required then a separate floating-point co-processor chip is available that can communicate with attached PICAXE chips and perform all the complex calculations. However, I wondered if it was possible to create simple floating point routines that could run onboard a PICAXE, that might give acceptable performance - and this article describes the result of my efforts to date. I'm pleased to say it gives pretty good performance (Sin A calculated in 1.7 seconds!)

The Calculator presented does not pretend to be a high performance utility. However it's what I believe to be the most efficient example of a method of performing floating-point maths using the BASIC commands available.

Code is available from my web-site:
http://home.btconnect.com/PicAxe_Projects/Home.htm



IN THIS ARTICLE

- 2 Summary of features / Limitations
- 2 Using the Calculator
- 4 Complex Functions
- 6 Code Saving
- 6 Memory Usage
- 6 Performance Tests
- 7 The development of the PICAXE Calculator
- 7 Using Base256
- 8 How to convert the Result to Decimal

ACKNOWLEDGEMENTS

- PICAXE is a trademark of Revolution Education Ltd.
- PICAXE Internet Forum: <http://www.rev-ed.co.uk/picaxe/forum/>
- Article about binary to decimal conversion : <http://www.cs.uiowa.edu/~jones/bcd/decimal.htm>

Summary of features:

- Performs Floating Point arithmetic on a PICAXE chip.
- Uses around 1100 bytes of code. This is about half the available code space for a PICAXE18x. This is obviously a lot of code to give up, however it's quite possible that existing code can be simplified by having the Calculator on board.
- 'Internally' uses Base256 arithmetic, which seems the optimum choice based on the PICAXE BASIC commands available.
- The Calculator is executed via compact command strings in Data memory.
- The Calculator only directly supports the simple Maths functions Add, Subtract and Multiply. However the command language allows more complex maths 'functions' (such as SinA) to be defined, based on the simple functions.
- Includes special display routine to display signed result to LCD display.

Limitations:

- Currently no Divide Routine! Divide is more complex and resource hungry than Add, Subtract and Multiply, and can often be avoided.
- The Calculator Stack is limited to 6 Floating-Point Values.
- Power function can only handle powers $> + 1$

Using the Calculator

The Calculator is used by executing pre-defined Command Strings in Data Memory.

A Command String is a sequence of bytes in an EEPROM statement. Each byte is a predefined Command Value (such as Add) or an Argument Value associated with the command (Such as FPVA).

Note: All Command Strings MUST end with a *CEnd* command.

Figure 1 shows the full list of commands available.

Example Command String

```
EEPROM 53,
(CRecall,FPVB,CRecall,FPVC,CMult,CRecall
,FPVA,CAdd,CDisplayResult,CEnd)
```

This performs: $Result = A + (B * C)$ then displays

the result on the LCD.

Executing the Command String

This is done by setting the CommandAddress then calling the *Execute* subroutine. For example:

```
CommandAddress = 55
Gosub Execute
```

String of Strings

One of the commands is *CCommand*. This allows a command string to include references to other command strings. This is powerful because it allows you to define custom maths functions and refer to them as if they were in-built Calculator functions. However it must be remembered that execution speed is affected by every calculation in a command.

The following example has a reference to a previously defined *SinA* command string:

Figure 1 : Commands

Calculator Command	Argument	Meaning
CAdd	None	Pop the Top and Top-1 stack FP values, add them and Push the result to the top of the Stack.
CSub	None	Pop the [Top] and [Top-1] stack FP values. Subtract [Top-1] value from [Top] value and Push the result to the top of the Stack.
CMultiply	None	Pop the Top and Top-1 stack FP values, multiply them and Push the result to the top of the Stack.
CDivide	None	NOT IMPLEMENTED. Pop the Top and Top-1 stack FP values, divide [Top] value by [Top -1] value and Push the result to the top of the Stack.
CDuplicate	None	Duplicates the FP value at the top of the Stack, so that the top of the stack holds two copies.
CDisplayResult	None	Display the current result value held in RAM as a signed decimal value to the specified number of decimal places.
CEnd	None	Ends the command string. Note if the current command was called from another command this returns execution to the other command. Otherwise it completes execution.
CPower	Power to raise to	Pop the [Top] stack FP value, raise it to the specified power and Push the result to the top of the Stack. Note: This command just performs repeated multiplication of the FP Values. This isn't always the most efficient way to calculate a result.
CSave	FPV Index to save to	Saves a copy of the [Top] FP value to the specified variable address.
CRecall	FPC Index to recall	Pushes the specified FP variable value onto the top of the Stack.
CPushConstant	Constant Index to Stack	Pushes the specified FP constant value onto the top of the Stack.
CPushCConstant	Constant FP Value (5 Bytes)	Pushes the embedded (in the command) FP constant value onto the top of the Stack. Note the embedded constant is written LSB first in the Eeprom statement.
CCommand	Command Address to Execute	Executes the command beginning at the specified address.

EEPROM

60,(CRecall,FPVA,CCommand,SinA,CDisplayResult,CEnd)

Note: Before getting carried away, it is not possible to define a 'string of string of strings' command! There is no Stack for the return addresses, just a single byte for a single return address.

Getting values into the Calculator

In order to perform a calculation, values must first be placed in the calculator. There are 4 ways of doing this:

1. Loading the 'Calculator Memory'

There are 3 RAM locations FPVA, FPVB and FPVC. These each hold a Floating-Point variable value. The value of a variable must be set by calling *Sub SaveWordValue0*. This will save the current value of W0 (signed) into the FP variable

specified by Index1.

Example:

```

W0 = 5
Index1 = FPVA
Gosub SaveWordValue0

```

2. Using predefined constants.

(See section below regarding how to work out constant values)

Data memory addresses 0 to 19 are set aside for predefined constants 0 to 3. Constants are useful for defining common values like Pi.

Example:

```

EEPROM
60,(CpushConstant,Constant0,CRecall,FPVA,C
Mult ,CdisplayResult,CEnd)

```

This multiplies FPVA by Constant0.

3. Using the Calculator Stack

The Calculator has an in-built Stack of 6 FPV Values, which provides a powerful but simple way of manipulating calculations. It effectively provides the ability of using parenthesis in calculations.

In the Calculator, the arithmetic commands Add, Sub and Multiply all operate on the two values held in the Stack and place the result of the calculation on the top of the Stack. The Stack Pointer is reset (ie the Stack Cleared) every time a Command String is executed.

Example:

```

To calculate: Result = A + (B * C):
First save values for A,B and C into FPVA, FPVB,
FPVC using SaveWordValue0 subroutine.
Then write and execute a Command String that will
do the following:
Step 1: Push FPVA and FPVB onto the Stack.
Step 2: Calculate B*C by using the CMultiply
command - the result is automatically added to the
stack.
Step 3: Push A value onto the Stack.

```

```

Step 4: Calculate A + (Result of B*C)
Step 5: Display the Result

```

4. Embedding Constants in the Command String.

(See section below regarding how to work out constant values)

Constants can be defined by embedding the value within the Command string.

Displaying the Calculated Result on the LCD

The CdisplayResult command takes the current Result Value held in RAM and converts it to a signed, decimal value to display on the LCD, to a specified number of decimal places. The display routine is quite complex to achieve and a limitation is that it can only display the whole part of the result, and simply inserts a decimal place a number of digits before the last digit, specified by the constant DecimalPlacesToDisplay.

So, for instance if the Result was 1234 and DecimalPlacesToDisplay was 2 then the displayed result would be 12.34 . This obviously isn't the real result. So it's necessary, prior to displaying the result, to multiply the result by 10 to the power of DecimalPlacesToDisplay, ie in this case by 100.

Complex Functions

It's possible to define complex maths functions based on the low level functions of Add, Subtract , Multiply and Divide.

One way to do this is by using Maclaurin's series:

$$f(x) = f(0) + (x)*f'(0) + (x^2/2!)* f''(0) + (x^3/3!)*f'''(0) + ...$$

The explanation of this formula can be found elsewhere, but the key point is that it allows certain functions to be calculated using 'normal' maths. For example, for Trig functions:

$$\text{Sin } x = x - x^3/3! + x^5/5! - x^7/7! + \dots \text{ where } x \text{ is in Radians}$$

$$\text{Cos } x = 1 - x^2/2! + x^4/4! - x^6/6! + \dots \text{ where } x \text{ is in Radians}$$

Because these series are based on basic maths functions, they can be written in a CommandString.

Careful consideration should be given to the number of terms calculated, because every instruction executed takes valuable time. Also there are often different ways of achieving the same result - some needing far less instructions than others.

Example derivation of SinA

This is a first attempt at the development of a Command String to define SinA, based on the first 3 terms of a Maclaurin series. See the next example for an improved version.

Step1: Do as much pre-calculation as possible.

1/3! = 0.16666666... translates to a FP Constant value (call it C1) : 62,42,170,170,192

1/5! = 0.00833333... translates to a FP Constant value (call it C2) : 62,2,34,34,2

Step2: It helps to write out the formula with brackets to understand the required calculation order and how the Stack should be used to achieve this. In general you need to evaluate expressions enclosed in brackets first.

$\text{SinA} = A - (C1 * A^3) + (C2 * A^5)$

So here we need to evaluate the bracketed expressions first, keep the results on the Stack, then perform the remaining addition and subtraction on those results.

Step 3: Translate this formula into a Command String. It helps to write it out sequentially first:

CRecall,FPVA This pushes A onto the Stack

CDuplicate This copies A, so that there are now two copies of A on the top of the Stack.

Cpower,3 This calculates A^3 It removes the operand from the Stack and replaces it with the Result.

(Note Stack is now <A><Result of A^3 ><TOP>)

CPushCConstant,170,170,42,62 This pushes the value for C1 onto the Stack.

Cmultiply This multiplies A^3 by C1. It removes the operands from the Stack and replaces them with the Result.

(Note Stack is now <A><Result of $(C1 * A^3)$ ><TOP>)

CRecall,FPVA This pushes A onto the Stack , again

Cpower,5 This calculates A^5 It removes the operand from the Stack and replaces it with the Result.

CPushCConstant,2,34,34,2,62 This pushes the value for C2 onto the Stack.

Cmultiply This multiplies A^5 by C2. It removes the operands from the Stack and replaces them with the Result.

(Note Stack is now <A><Result of $(C1 * A^3)$ ><Result of $(C2 * A^5)$ ><TOP>)

Csub This subtracts $(C1 * A^3)$ from $(C2 * A^5)$. It removes the operands from the Stack and replaces them with the Result.

(Note Stack is now <A><Result of $(C2 * A^5) - (C1 * A^3)$ ><TOP>)

Cadd This performs the final addition. It removes the operands from the Stack and replaces them with the Result.

(Note Stack is now <Result of Sin A><TOP>)

Step4: Now put all these Commands into an EEPROM Statement. Declare a Symbol to equal the start address of the EEPROM data, and you now have a function for SinA !.

More efficient version of SinA

Careful consideration should be given to the most efficient way of getting a result and this section describes a better derivation of Sin A:

The original equation is:

$$\text{Sin A} = A - (C1 * A^3) + (C2 * A^5)$$

Careful analysis of the calculations performed in getting the result shows that there are 8 multiplications:

A^3 : This requires 2 multiplications (A x A x A)

$C1 * A^3$: This requires 1 Multiplication

A^5 : This requires 4 multiplications (A x A x A x A x A)

$C2 * A^5$: This requires 1 Multiplication

Multiplication is more time-consuming to execute than addition or subtraction, so should be minimised wherever possible. Also results of calculations should be saved and re-used rather than calculating again, because saving and loading FP Values is relatively easy (just moving bytes

around).

The equation can be re-expressed as:

$$\text{Sin } A = A - A * A^2 * (C1 - C2 * A^2)$$

This looks messy, but careful analysis shows that only 4 multiplications are required, assuming the result of A^2 is stored and reused. There is still only one addition and one subtraction, so we would expect this command to run approaching twice as fast as the original *SinA* command.

This new equation results in the following command string:

```
EEPROM
20,(CDuplicate,CPower,2,CDuplicate,CPushCC
onstant,2,34,34,2,62,CMultiply,
CPushCConstant,192,170,170,42,62,CSub,CMu
ltiply,CRecall,FPVA,CMultiply,CSub, CEnd)
```

This command string is 26 bytes long, which is actually the same as the original Command string.

Code Saving

Although the calculator requires a hefty amount of code, its use may reduce the code requirements in the main program. Consider the following example to display an accurate temperature reading:

A favourite application of a PICAXE is to attach a DS18B20 temperature chip. This chip is capable of displaying temperature to 0.0625 (1/16) of a degree. However it's not that simple to write the code to display a value to this precision (plus negative values).

The following shows how this task is relatively simple using this Calculator. The sub *ReadAndDisplayTemperature* will display a signed temperature value to a predefined number of digits.

```
EEPROM
60,(Recall,FPVA,CPushCConstant,0,0,0,160,62,
CMult, Ccommand, ScaleAndDisplayResult,
Cend)
```

Symbol DisplayTemp = 60

ReadAndDisplayTemperature:

ReadTemp12,1,w0

Index 1 = FPVA

SaveWordValue0

```
CommandAddress = DisplayTemp
```

```
Gosub Execute
```

Memory Usage

Figure 2 and Figure 3 describe how the PICAXE Calculator uses DATA and RAM memory. Do not use the memory locations for anything else, unless you understand how the Calculator works.

Performance Tests

Time to calculate

All tests performed on standard PICAXE 18x at 4MHz:

Test 1:

Calculated Sin A, plus displayed result. Performed this repeatedly 10 times. Took 22 seconds overall. Average of 1.7 seconds.

Test 2: Calculated Sin A (without displaying result): Performed this repeatedly 10 times. Took 17 seconds overall. Average of 0.5 seconds.

Accuracy

The accuracy of the result is heavily dependent on the number of calculations performed. Errors will start creeping in because results are rounded as a calculation proceeds and these errors will accumulate. Nevertheless, the results for SinA above are quite impressive: The results are accurate against exact values of SinA to 2 decimal places or more (especially for angles under 45 degrees).

NOTE: SinA is calculated using an approximated formula. The results compared to the exact results of this approximated formula are accurate to 6 decimal places or more! The error between the approximation of SinA and SinA is shown in the table in Figure 4.

Of course, the approximation of SinA could be improved by including an additional term from the Series, however this improvement in accuracy would come at the price of slower calculation.

Figure 2 : Data Memory Usage

DATA MEMORY ADDRESS	USE
0 to 4	Constant 0
5 to 9	Constant 1
10 to 14	Constant 2
15 to 19	Constant 3
20 to 119	Can be used for user EEPROM Command Strings
120 to 151	Array of fixed data used by DisplayResult routine.
152 to 255	FREE

Figure 3 : RAM Memory Usage

RAM MEMORY ADDRESS	USE
80 to 82	Pointers used by code
83 to 92	Digit Buffer used in formulating the digits to display on the LCD
93 to 119	FREE
120 to 127	Result Mantissa
192 to 194	Pointers used by code
195 to 224	Calculator Stack Space
225 to 239	FP Variables A, B , C

Figure 4: Error in approximation

Degrees	10	20	30	40	50	60	70	80
Radians	0.174533	0.349066	0.523599	0.698132	0.872665	1.047198	1.22173	1.396263
SinA (Approx)	0.173648	0.34202	0.500002	0.642804	0.76612	0.866295	0.940482	0.986806
SinA (Exact)	0.173648	0.34202	0.5	0.642788	0.766044	0.866025	0.939693	0.984808
Error	-4.1E-09	1.15E-07	2.12E-06	1.59E-05	7.57E-05	0.00027	0.00079	0.001998

The development of the PICAXE Calculator

Floating-point arithmetic revolves around defining numbers in terms of Mantissa and Exponent parts, and then working with these values. Standard floating-point formats are based on binary arithmetic with routines manipulating the floating-point values at bit-level and eventually converting the result back to decimal. The routines involve a lot of bit shifting and low-level manipulation.

My initial thoughts were that it's quite code-hungry to manipulate at bit-level to the level required using PICAXE Basic. So a good first step would be to develop FP routines that manipulate all numbers in base10 and not at a binary level at all. I wrote the code and working to Base10 certainly made programming easier to understand, plus a great

advantage was that there wasn't a need to convert between bases to display the final result. However a major downside was very slow execution and wasteful use of memory.

I then realised that the best way forward would be to ditch any standard ideas of floating point routines and go for a format and set of routines that suited PICAXE Basic. This led to the use of Base256 for all calculations.

Using Base256

Base 256 doesn't seem to be a familiar term, but all it means is that each 'digit' in a base256 number system contains a value of 0 to 255, a Byte value. By using this system:

- The core maths functions are simplified and faster because PICAXE Basic maths is based on Byte and Word values.

- The number data is far more compact.
- The number of digits to handle in all the routines is much smaller.

I've devised the following simple floating point format using Base356 numbers. It's 5 Bytes long:

Byte4	Byte3	Point	Byte2	Byte1	Byte0
Sign & Exponent	Digit3	.	Digit2	Digit1	Digit0

Although a 4 digit Mantissa doesn't seem much, it can actually hold a value up to (in decimal) $256 \times 256 \times 256 \times 256 = 4,278,190,080$!

The Sign and Exponent Byte has a leftmost bit indicating the sign of the number (1 is negative, 0 is positive). The remaining 7 bits of form the Exponent, which is offset by 63 (ie an Exponent of 0 is set as 63). The offset makes calculations easier. In reality the offset Exponent is only likely to deviate from 63 by a small amount - so the format isn't fantastic, but it's easy to work with.

Working out Base256 values

To use constant values you need to be able to convert a decimal number into Base256 number format. This is pretty simple for simple integers, but where you have a decimal number with a decimal point things get a lot trickier!

To make life easier, I've written a utility in Excel™ that does the conversion. This can be found on my website.

Rounding

During calculations, results have to be adjusted to conform to the number format used. This means that least significant digits have to be lost. When losing these digits it's desirable to round the result to preserve as much accuracy as possible. If rounding isn't done (the result is truncated) then all results will be skewed. For example 1.9 is almost 2 so it's more accurate to round up to 2 than truncate to 1.

However rounding adds complexity to the code. Looking at the problem simply: If the digit being lost is say, 200 (in Base 256) then the remaining rightmost digit needs to be rounded up. However if the rightmost digit is 255 then not only does this digit

need incrementing, so does the next. I.e. the whole result needs to be incremented.

The code to do this isn't difficult - it's just a version of the *Add* routine I've already got. However it struck me that it's not justified to add the extra code and the processing time to increment the whole result (this subroutine will be called a lot during calculation). Instead increment the remaining rightmost digit if necessary, but ONLY if it's not 255 already. This is obviously an imperfect algorithm, however the chances of the situation of the digit being 255 already are low and the difference in code is very significant. Even if the '255 situation' arises it will only have a minor effect on an overall result. In practice this compromise seems to work very well.

How to convert the Result to Decimal

If the internal calculations are performed in a number base other than decimal this immediately raises the issue of how to translate the end result of calculations back to decimal so humans can understand it!

Converting a base256 result to decimal is not as easy as it may seem. The simplest method would be to repeatedly divide the result by 10. This requires extensive calculation, which must be performed by a calculator that can handle potentially large numbers. However the PICAXE and other micro-controllers can only handle Word integers.

A clever idea that gets round this problem and requires just Word integer calculation is explained on this web site:

<http://www.cs.uiowa.edu/~jones/bcd/decimal.html>

I've taken this idea and developed it into the display routine as part of my PICAXE floating point program. It may look really involved, but it actually reduces to fairly straightforward code.

Computing the Decimal Digits

This is a two-stage technique. The first stage involves making first approximations for the number of units each digit holds. For example the approximation could be 12 units in the "10's" digit.

The second stage calculates the exact decimal digits from these approximations.

In more depth...

If the result is in 4 x base256 digits (R0 to R3) :

R3	R2	R1	R0
-----------	-----------	-----------	-----------

$$\text{Result} = (\text{R0}) + (\text{R1} * 256) + (\text{R2} * 65536) + (\text{R3} * 16777216)$$

$$\text{Highest Result} = 255 + (255 * 256) + (255 * 65536) + (255 * 16777216) = 4,294,967,295 \text{ which is } 10 \text{ decimal digits long.}$$

Expressing the same result as decimal digits:

D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------	-----------

$$\text{Result} = \text{D0} + \text{D1} * 10 + \text{D2} * 100 + \text{D3} * 1000 + \text{D4} * 10000 + \text{D5} * 100000 + \text{D6} * 1000000 + \text{D7} * 10000000 + \text{D8} * 100000000 + \text{D9} * 1000000000$$

The result can also be expressed as:

$$\begin{aligned} \text{Result} &= (\text{R0} + 6 * \text{R1} + 6 * \text{R2} + 6 * \text{R3}) \\ &+ 10 * (5 * \text{R1} + 3 * \text{R2} + \text{R3}) \\ &+ 100 * (2 * \text{R1} + 5 * \text{R2} + 2 * \text{R3}) \\ &+ 1000 * (5 * \text{R2} + 7 * \text{R3}) \\ &+ 10000 * (6 * \text{R2} + 7 * \text{R3}) \\ &+ 100000 * (7 * \text{R3}) \\ &+ 1000000 * (6 * \text{R3}) \\ &+ 10000000 * (\text{R3}) \end{aligned}$$

If we define first approximation values, A, for the decimal digits:

$$\begin{aligned} \text{A0} &= (\text{R0} + 6 * \text{R1} + 6 * \text{R2} + 6 * \text{R3}) \\ \text{A1} &= (5 * \text{R1} + 3 * \text{R2} + \text{R3}) \\ \text{A2} &= (2 * \text{R1} + 5 * \text{R2} + 2 * \text{R3}) \\ \text{A3} &= (5 * \text{R2} + 7 * \text{R3}) \\ \text{A4} &= (6 * \text{R2} + 7 * \text{R3}) \\ \text{A5} &= (7 * \text{R3}) \\ \text{A6} &= (6 * \text{R3}) \\ \text{A7} &= (\text{R3}) \end{aligned}$$

Upper bounds for A values are therefore:

$$\text{A0 upper bound} = 256 + 6 * 256 + 6 * 256 + 6 * 256 = 4864$$

$$\text{A1 upper bound} = 5 * 256 + 3 * 256 + 256 = 2304$$

$$\text{A2 upper bound} = 2 * 256 + 5 * 256 + 2 * 256 = 2304$$

$$\text{A3 upper bound} = 5 * 256 + 7 * 256 = 3072$$

$$\text{A4 upper bound} = 6 * 256 + 7 * 256 = 3328$$

$$\text{A5 upper bound} = 7 * 256 = 1792$$

$$\text{A6 upper bound} = 6 * 256 = 1536$$

$$\text{A7 upper bound} = 256 = 256$$

Therefore all 'A' values can be stored in a Word (2 byte) variable.

However the 'A' values are just approximations to the decimal digits. To compute the exact decimal digit D values, compute Carry C values:

$$\text{C1} = \text{A0} / 10$$

$$\underline{\text{D0}} = \text{A0} \bmod 10$$

$$\text{C2} = (\text{A1} + \text{C1}) / 10$$

$$\underline{\text{D1}} = (\text{A1} + \text{C1}) \bmod 10$$

$$\text{C3} = (\text{A2} + \text{C2}) / 10$$

$$\underline{\text{D2}} = (\text{A2} + \text{C2}) \bmod 10$$

$$\text{C4} = (\text{A3} + \text{C3}) / 10$$

$$\underline{\text{D3}} = (\text{A3} + \text{C3}) \bmod 10$$

$$\text{C5} = (\text{A4} + \text{C4}) / 10$$

$$\underline{\text{D4}} = (\text{A4} + \text{C4}) \bmod 10$$

$$\text{C6} = (\text{A5} + \text{C5}) / 10$$

$$\underline{\text{D5}} = (\text{A5} + \text{C5}) \bmod 10$$

$$\text{C7} = (\text{A6} + \text{C6}) / 10$$

$$\underline{\text{D6}} = (\text{A6} + \text{C6}) \bmod 10$$

$$\text{C8} = (\text{A7} + \text{C7}) / 10$$

$$\underline{\text{D7}} = (\text{A7} + \text{C7}) \bmod 10$$

$$\text{C9} = (\text{C8}) / 10$$

$$\underline{\text{D8}} = (\text{C8}) \bmod 10$$

$$\underline{\text{D9}} = (\text{C9}) \bmod 10$$

The upper bounds for C are:

$$C1 \text{ upper bound} = 4864 / 10 = 486$$

$$C2 \text{ upper bound} = (2304 + 486)/10 = 279$$

$$C3 \text{ upper bound} = (2304 + 279)/10 = 258$$

$$C4 \text{ upper bound} = (3072 + 258)/10 = 333$$

$$C5 \text{ upper bound} = (3328 + 333)/10 = 366$$

$$C6 \text{ upper bound} = (1792 + 366)/10 = 215$$

$$C7 \text{ upper bound} = (1536 + 215)/10 = 175$$

$$C8 \text{ upper bound} = (256 + 175)/10 = 43$$

$$C9 \text{ upper bound} = 4$$

Therefore all C values can comfortably be stored and computed in a Word Value.

Algorithm

Step 0 :

Shift the Result until it's related exponent value is Zero. Assume only displaying the whole part of the result, not the fractional part. If the whole part needs more than 4 base256 digits then raise an error 'display overflow'.

Note: In order to display a decimal result to a specified number of decimal places using this method, it's necessary to perform a suitable multiplication on the result beforehand. E.g. for 2 decimal places multiply by 100 and display the decimal point between D1 and D2.

Step 1:

Compute approximations A0 to A7 for each decimal digit.

Step 2:

Compute and output Digits D0 to D9.

Steps 2 and 3 can in practice be combined in the same For...Next loop.

One difficulty is in calculating the Approximation values. I've found that the most code-memory-efficient method is to hold details of the calculations as an array in data memory.

