MCF51EM256 Performance Assessment with Algorithms Used in Metering Applications

by: Paulo Knirsch
MSG IMM System and Applications

1 Introduction

This application note’s objective is to demonstrate the implementation of the following algorithms used in metering applications using Freescale MCF51EM256.

- Square root
- Voltage and current RMS values
- Active energy, active power, apparent power, reactive power, and power factor
- Discrete fourier transform (DFT)
- Total harmonic distortion (THD)

The methodology used is to present the formula for the calculation, discuss its implementation, and present execution performance analysis.

The calculation of the performance required to process the algorithms is estimated. The following assumptions are made:

- The power line frequency is 60 Hz.
Test Setup

- The sampling rate is 15.360 kHz, therefore there are 256 samples per power line cycle.
- 256 samples are stored in a double buffer, while one buffer is being updated the other is used for the calculations.

The MCF51EM256 was configured to operate with 25 MHz of bus frequency.

2 Test Setup

The DEMOEM demo board was used. The test software project, EM256_Performance.mcp was developed with CodeWarrior V6.2. The compiler optimization was set to Level 4, faster execution speed. The tests were executed with the CodeWarrior debugger using the Terminal Window application from P&E Micro to visualize the results (configure to 19200 bps, parity none, and 8-bits).

Figure 1. Terminal window application

Figure 2. MCF51EM256 performance test initial window

The test project main files are illustrated in Figure 3.
The metering tests are implemented in the files:

  - Metering_algorithmTests.c
  - Metering_algorithmTests.h

The algorithms tests are implemented in the files:

  - Metering_algorithms.c
  - Metering_algorithms.h

The DFT coefficients are in the file:

  - DFT_coef.h

The input data sets used to perform the test are in the file:

  - inputdata.h

Table 1 describes the datasets used for the tests.
The expected outputs of the algorithms were determined using the excel spreadsheet Performance Analysis.xls. They were obtained doing exactly the same algorithm as implemented in the C code for the MCU.

### 3 Square Root

**Formula**

The square root is calculated using the Babylonian method, see Equation 1.
This method calculates the square root by an interaction. The seed value X₀ needs to be near the desired square root value to reduce the number of interactions necessary for a good precision result.

The example implementation here uses the following seed value for the square root calculations. **Equation 2** is a square root initial guess.

\[
x_0 = 2^{D/2} \quad \text{(here D is the number of binary digits)}
\]

### 3.1 Implementation

```c
word SquareRoot(word A){
    word j; // local variable
    Acc = ASM_FF1(\(A\)); // initial seed
    for(j=0;j<MAX_INT;j++){ // execute maximum of MAX_INT interactions
        Acc_temp = (Acc*Acc + Acc)/2; // Calculate interaction
        if(abs(Acc - Acc_temp) < 0x0001){ // Test is precision is good enough
            break; // break if a good precision was reached
        }
        Acc = Acc_temp; // Save previous interaction
    }
    return((word)Acc<<j); // For debugging purposes
}
```

Figure 4. Square root code implementation

```c
dword ASM_FF1(dword A){
    if(A == 0)
        return(0);
    A = (32-A)>>1; // first one index divided by 2
    A = (unsigned long)1<<A; // multiplies by two
    return(A);
}
```

Figure 5. Square root seed generation

### 3.2 Square Root Test Result

The algorithm performance was tested with four different data sets.
Table 2. Datasets used for a square root test

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Dataset name (check table 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st</td>
<td>InputVec</td>
</tr>
<tr>
<td>2nd</td>
<td>InputVec1</td>
</tr>
<tr>
<td>3rd</td>
<td>InputVec2</td>
</tr>
<tr>
<td>4th</td>
<td>InputVec3</td>
</tr>
</tbody>
</table>

The output of the test obtained with the terminal window can be seen in Figure 6.

Figure 6. Square root test terminal window output

Table 3 summarizes the square root test result.
RMS Calculation

Maximun observed time is 48 μs for one 32-bit square root calculation.

4 RMS Calculation

The RMS equation is as follows:

\[ VRMS = V_{rms} = \frac{1}{N} \times \sum_{i=1}^{N} V^2(i) \]

Eqn. 3
4.1 Implementation

```c
// RMS calculation
word RMS_calc(short *input){
    Sum = 0; // clear accumulator variable
    for(i=0;i<N_SAMPLES;i++) { // execute interaction
        if(ASM_SUM == 1) // Assembly implementation of 64-bit sum
            ADD64bits(&Sum,(*input)*(*input)); // call Assembly function
            input++; // update pointer
        else // if not using assembly
            Sum = Sum + (*input)[i]*(*input)[i]; // multiply and accumulate
    }
    return(SquareRoot dword(Sum)); // divide final sum by number of samples
}
```

Figure 7. RMS algorithm code implementation

The intermediate result was stored with 64-bit precision to avoid intermediate results overflow.

The following implementation optimizations were made to reduce the execution time:

- Implementation of a 32-bit addition with 64-bit accumulation in assembly
- Use of pointers instead of arrays
- Division implemented by shifting

Figure 8 shows the 64-bit addition implemented in assembly.

```asm
// 64-bit accumulate
// Input parameter: signed 64-bits pointer
// Output parameter: signed 32-bits
void ADD64bits(long long * ACC, long A){
    ADD.1 D0.4(A0) // Add 32-bit to 64-bit accumulator
    BCC OUT // Check overflow
    ADD.1 D0.4(A0) // if overflow increase MS 32-bits of ACC
    OUT
    RTS // Return
}
```

Figure 8. 32-bits addition to 64-bit accumulator

A “define” controls if the algorithms use the standard C compiler 64-bit addition or the proposed assembly implementation. The “define” is in the metering_algorithms.h and is shown below. If defined as 1 it uses the assembly 64-bit addition. If defined as 0 use the standard C addition.

```c
#define ASM_SUM 1
```

The RMS implementation allows the configuration of the number of samples to be used for the calculation. The SAMPLE_FOR_RMS define shown below controls and can be found at the top of the metering_algorithms.h file.

```c
#define Sample_FOR_RMS 32
```
4.2 RMS Test Results

The algorithm performance was tested with four different data sets.

Table 5. Datasets used for the RMS test

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Data set name (check table 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st</td>
<td>InputVecRMS</td>
</tr>
<tr>
<td>2nd</td>
<td>InputVecRMS1</td>
</tr>
<tr>
<td>3rd</td>
<td>InputVecRMS2</td>
</tr>
<tr>
<td>4th</td>
<td>InputVecRMS3</td>
</tr>
</tbody>
</table>

Figure 9 illustrates the RMS test terminal window output.

![Figure 9. RMS Test terminal output (# samples = 256)](image)

The RMS tests were repeated using 256, 128, 64, and 32 samples to evaluate the execution time versus precision trade-off. The results are shown in the Table 5 and Table 6.

Table 6. RMS execution time

<table>
<thead>
<tr>
<th>Dataset</th>
<th>256 Samples</th>
<th>128 Samples</th>
<th>64 Samples</th>
<th>32 Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st</td>
<td>277 µs</td>
<td>166 µs</td>
<td>103 µs</td>
<td>72 µs</td>
</tr>
<tr>
<td>2nd</td>
<td>267 µs</td>
<td>156 µs</td>
<td>94 µs</td>
<td>57 µs</td>
</tr>
<tr>
<td>3rd</td>
<td>257 µs</td>
<td>145 µs</td>
<td>83 µs</td>
<td>52 µs</td>
</tr>
<tr>
<td>4th</td>
<td>248 µs</td>
<td>136 µs</td>
<td>74 µs</td>
<td>43 µs</td>
</tr>
</tbody>
</table>
Power Measurements

Observed results are as expected considering the truncation due to quantization.

One additional optimization that can be done is to store the intermediate results in 32-bits precision. This requires normalization of the intermediate values resulting in less precision of the RMS value.

5 Power Measurements

Power measurements is a routine that receives as input the voltage buffer, current buffer, and calculates the following outputs:

1. Total energy
2. Active power
3. Reactive power
4. Apparent power
5. Vrms and Irms
6. Power factor

The buffers must contain 256 samples (N) that must correspond to a complete period of the power main. The outputs are calculated with the following equations:

\[
\text{Energy} = \sum_{i=0}^{N} V[i] \times I[i]
\]

Eqn. 4

\[
\text{ActivePower} = \frac{\text{Energy}}{N}
\]

Eqn. 5

\[
\text{ApparentPower} = V_{\text{rms}} \times I_{\text{rms}}
\]

Eqn. 6

\[
\text{ReactivePower} = \sqrt{\text{ApparentPower}^2 - \text{ActivePower}^2}
\]

Eqn. 7

\[
\text{Power(Factor)} = \text{ActivePower} - \text{ApparentPower}
\]

Eqn. 8

Vrms and Irms are calculated as described in the Section 4, “RMS Calculation.”

Table 7. RMS test results precision

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Expected value</th>
<th>256 Samples result</th>
<th>128 Samples result</th>
<th>64 Sample results</th>
<th>32 Sample results</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st</td>
<td>23169.77</td>
<td>23169</td>
<td>23169</td>
<td>23169</td>
<td>23169</td>
</tr>
<tr>
<td>2nd</td>
<td>707.11</td>
<td>707</td>
<td>707</td>
<td>707</td>
<td>707</td>
</tr>
<tr>
<td>3rd</td>
<td>70.71</td>
<td>70</td>
<td>70</td>
<td>70</td>
<td>70</td>
</tr>
<tr>
<td>4th</td>
<td>7.07</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 7. RMS test results precision

Vrms and Irms are calculated as described in the Section 4, “RMS Calculation.”
5.1 Implementation

Input parameter: - Voltage pointer to 16-bit signed
- Current pointer to 16-bit signed
- Output Vector pointer
- Output Tls vector pointer

Output parameter: unsigned 16-bit

Power_Calc(unsigned *V, unsigned *I, Power_Vec *Out, Power_1sce *in){
  
  L &  // Local var used as ACC
  Vtemp = V;  // Local pointers
  Itemp = I;

  // Active Energy Calculation
  startTimer();
  t = 0;
  for(=0;<N_SAMPLES; i++) { // Start timer to measure execution time
    sum = sum + (*tempV++)/*(*tempI++); // clear accumulator variable
    // multiplication and accumulate
    //
    // Active_Eng = sum; // output result
    // Active Power Calculation
    startTimer();
    startTimer();
    // Start timer to measure execution time
    
  // Voltage RMS Calculation
  startTimer();
  startTimer();
  // Start timer to measure execution time
  // Voltage RMS calculation
  
  startTimer();
  // Stop timer to measure execution time

  // Current RMS Calculation
  startTimer();
  startTimer();
  // Start timer to measure execution time
  // Current RMS calculation
  
  startTimer();
  // Stop timer to measure execution time

  // Power Factor Calculation
  startTimer();
  startTimer();
  // Start timer to measure execution time
  // Check if apparent power is smaller than active
  // Correct output
  
  // Power Factor equals one
  
  // Reactive Energy Calculation
  startTimer();
  // Start timer to measure execution time

  startTimer();
  // Start timer to measure execution time

  startTimer();
  // Stop timer to measure execution time

  // Stop timer to measure execution time

Figure 10. Power calculation implementation code

The energy calculation algorithm used 64-bit intermediate results as well as the Vrms and Irms to ensure maximum precision.

<table>
<thead>
<tr>
<th>Table 8. Algorithms output data types</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Output</strong></td>
</tr>
<tr>
<td>Total energy</td>
</tr>
<tr>
<td>Active Power</td>
</tr>
<tr>
<td>RMS</td>
</tr>
<tr>
<td>Reactive Power1</td>
</tr>
<tr>
<td>Power factor2</td>
</tr>
</tbody>
</table>
The implementation includes the StartTPM() and Stop_Read_TPM() functions. These are used for debug purposes only. These lines must be removed in the final implementation.

The output values are stored in the structure defined as shown in Figure 11.

```c
typedef struct{
  long Act_Eng;  // Active Energy
  long Act_Pwr;  // Active Power
  long React_Pwr;  // Reactive Power
  long Apr_Pwr;  // Apparent Power
  word Vrms;  // Vrms
  word Irms;  // Irms
  word Pwr_fct;  // Power_factor
  word Vec;  // Power_variable_structure
}Power_time;
```

Figure 11. Power variable structure

The Power_time structure was defined to store measured execution times for debugging purposes only. It may be removed from the implementation if it is not required.

### 5.2 Test Results

The algorithm performance was tested with three different data sets each one having two 256 16-bit inputs, one used as the voltage buffer, and the other used as the current buffer.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Data Set Name (check table 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Voltage input</td>
<td>Current Input</td>
</tr>
<tr>
<td>1st</td>
<td>InputVecRMS</td>
</tr>
<tr>
<td>2nd</td>
<td>inputsignal1</td>
</tr>
<tr>
<td>3rd</td>
<td>inputsignal3</td>
</tr>
</tbody>
</table>

Figure 12. Terminal window output for power calculation test (dataset 1)
### Table 10. 1st dataset results table

<table>
<thead>
<tr>
<th>1st Dataset Power Calculation Results</th>
<th>expected</th>
<th>obtained</th>
<th>error</th>
<th>execution time [μS]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Energy</td>
<td>137430225637</td>
<td>137430225637</td>
<td>0.000000%</td>
<td>609</td>
</tr>
<tr>
<td>Act_Pwr</td>
<td>536836819</td>
<td>536836818</td>
<td>0.000000%</td>
<td>3</td>
</tr>
<tr>
<td>Vrms</td>
<td>23170</td>
<td>23169</td>
<td>0.003191%</td>
<td>276</td>
</tr>
<tr>
<td>Irms</td>
<td>23170</td>
<td>23169</td>
<td>0.003191%</td>
<td>277</td>
</tr>
<tr>
<td>Apr_Pwr</td>
<td>536836819</td>
<td>536836818</td>
<td>0.000000%</td>
<td>1</td>
</tr>
<tr>
<td>React_Pwr</td>
<td>0</td>
<td>0</td>
<td>0.000000%</td>
<td>3</td>
</tr>
<tr>
<td>Prw_fct</td>
<td>65536</td>
<td>65535</td>
<td>0.001526%</td>
<td>0</td>
</tr>
<tr>
<td>Total time</td>
<td></td>
<td></td>
<td></td>
<td>1,169</td>
</tr>
</tbody>
</table>

### Table 11. 2nd Dataset results table

<table>
<thead>
<tr>
<th>2nd Dataset Power Calculation Results</th>
<th>expected</th>
<th>obtained</th>
<th>error</th>
<th>execution time [μS]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Energy</td>
<td>97178042060</td>
<td>97178042060</td>
<td>0.0000000%</td>
<td>596</td>
</tr>
<tr>
<td>Act_Pwr</td>
<td>379601727</td>
<td>379601726</td>
<td>0.0000003%</td>
<td>3</td>
</tr>
<tr>
<td>Vrms</td>
<td>23170</td>
<td>23169</td>
<td>0.0043159%</td>
<td>263</td>
</tr>
<tr>
<td>Irms</td>
<td>23170</td>
<td>23169</td>
<td>0.0043159%</td>
<td>263</td>
</tr>
<tr>
<td>Apr_Pwr</td>
<td>536837910</td>
<td>536802561</td>
<td>0.0065847%</td>
<td>0</td>
</tr>
<tr>
<td>React_Pwr</td>
<td>379601727</td>
<td>379453440</td>
<td>0.0390638%</td>
<td>34</td>
</tr>
<tr>
<td>Prw_fct</td>
<td>46341</td>
<td>46344</td>
<td>0.0064737%</td>
<td>87</td>
</tr>
<tr>
<td>Total time</td>
<td></td>
<td></td>
<td></td>
<td>1,246</td>
</tr>
</tbody>
</table>

### Table 12. 3rd Dataset results table

<table>
<thead>
<tr>
<th>3rd Dataset Power Calculation Results</th>
<th>expected</th>
<th>obtained</th>
<th>error</th>
<th>execution time [μS]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Energy</td>
<td>127874757</td>
<td>127874757</td>
<td>0.0000000%</td>
<td>591</td>
</tr>
<tr>
<td>Act_Pwr</td>
<td>499511</td>
<td>499510</td>
<td>0.0002002%</td>
<td>3</td>
</tr>
<tr>
<td>Vrms</td>
<td>11292</td>
<td>11291</td>
<td>0.0088558%</td>
<td>259</td>
</tr>
<tr>
<td>Irms</td>
<td>11292</td>
<td>11291</td>
<td>0.0088558%</td>
<td>261</td>
</tr>
</tbody>
</table>

MCF51EM256 Performance Assessment with Algorithms Used in Metering Applications, Rev. 0
Discrete Fourier Transform (DFT)

A popular method used for energy metering is the discrete fourier transform (DFT) that can estimate the voltage and current phasors. At the same time it eliminates the DC component and harmonics. The phasors are described as follows:

DFT formula:

\[
Z_{rk} = \frac{2}{N} \sum_{r=0}^{N-1} Z_{k-r} \cos \frac{2\pi r}{N}
\]

\[
Z_{ik} = \frac{2}{N} \sum_{r=0}^{N-1} Z_{k-r} \sin \frac{2\pi r}{N}
\]

\[Eqn. 9\]

Where \(Z_{rk}\) is the phasor’s real part, \(Z_{ik}\) is the phasor’s imaginary part and both are expressed as functions of the \(k\) element. \(Z_{k-r}\) is the \(k\) sample, \(N\) is the number of samples, and \(r\) is the angle step defined for the function sine and cosine functions.

Using this approach in Equation 10 voltages are expressed with the DFT formula:

\[
V_{rk} = \frac{2}{N} \sum_{r=0}^{N-1} V_{k-r} \cos \frac{2\pi r}{N}
\]

\[
V_{ik} = \frac{2}{N} \sum_{r=0}^{N-1} V_{k-r} \sin \frac{2\pi r}{N}
\]

\[Eqn. 10\]

The voltage phasor is obtained by:

\[
|\mathcal{V}| = \sqrt{V_r^2 + V_i^2}
\]

\[\theta = \tan^{-1} \frac{V_r}{V_i}\]

\[Eqn. 11\]

The mean and RMS values are obtained by:

\[
V_{mean} = |\mathcal{V}|
\]

\[
V_{RMS} = \frac{|\mathcal{V}|}{\sqrt{2}}
\]

\[Eqn. 12\]
The same mathematical operations are used to obtain the current phasor and are not repeated here. The complex, active, and reactive power can be expressed in terms of the current and voltage phasors as follows.

\[ S = V I^* = P + jQ \]
\[ P = V_1 I_1 + V_1 I_1 \]
\[ Q = V_1 I_1 - V_1 I_1 \]  

**Eqn. 13**

### 6.1 Implementation

The input parameter is a pointer to a 16-bit signed data buffer.

The output parameter is a structure described in **Equation 13**. It returns the real and imaginary values of the output complex vector. Both the real and imaginary values are stored as a 32-bit signed value (word).

The intermediate result was stored with 64-bit precision to avoid intermediate results overflow. The implementation uses 32-bit intermediate results, if 64-bit values are not needed to reduce the execution time.

```
typedef struct {
    long Real;
    long Img;
} Complex;
```

**Figure 13. Complex structure**

The sample frequency \( f_s \) and the number of points in the data buffer, \( N \), determine the fundamental frequency component of the DFT output. The fundamental frequency of the DFT can be calculated using **Equation 14**.

\[ F_k = f_s/N \]  

**Eqn. 14**

In the case considered for this application note, \( f_s = 15.360 \) kHz and \( N = 256 \). Therefore, the \( F_k \) frequency is 60 Hz, exactly the power network frequency being considered. Higher frequency harmonics can be calculated using the formula below, DFT formula for “\( k \)” harmonic:

\[ Z_{\text{real}_k} = \frac{2}{N} \sum_{n=0}^{N-1} X_n \cos\left(\frac{2\pi nk}{N}\right) \]
\[ Z_{\text{img}_k} = \frac{2}{N} \sum_{n=0}^{N-1} X_n \sin\left(\frac{2\pi nk}{N}\right) \]  

**Eqn. 15**

Where “\( k \)” is the harmonic number. In this case, the harmonic frequency is:

\[ F_k = f_s \cdot k/N \]  

**Eqn. 16**
Then for the network third harmonic, \( k = 3 \) and \( F_k = 180 \text{ Hz} \).

```c
Complex DFT(short *input){
    Complex Res;
    short *P1, *P2;
    P1 = input; // pointers
    P2 = &cos_coef_k[0]; // coefficients pointer init (Real)
    Sun_D = 0;
    for(i=0;i<N_SAMPLES;i++)//DEC:
        Sun_D = Sun_D + (((P1)*(P2)>>DFT_SCALING)); // square and accumulate the N_SAMPLES_DFT samples
        P1++; // increment pointers
        P2++; // increment pointers
    Res.Real = (long)((Sun_D)<<(FAC_SAMPLE_FOR_DFT + COEF_MAX - 1 - DFT_SCALING));

    P1 = input; // input pointer initialisation (Imag)
    P2 = &sin_coef_k[0]; // coefficients pointer init (Imag)
    Sun_D = 0;
    for(i=0;i<N_SAMPLES;i++)//DEC:
        Sun_D = Sun_D + (((P1)*(P2)>>DFT_SCALING)); // square and accumulate the N_SAMPLES_DFT samples
        P1++; // increment pointers
        P2++; // increment pointers
    Res.Imag = (long)((Sun_D)<<(FAC_SAMPLE_FOR_DFT + COEF_MAX - 1 - DFT_SCALING));

    return(Res);
}
```

Figure 14. DFT Implementation for first harmonic

To allow faster execution speed, the implementation allows selection of the number of samples used for the DFT calculation.

```c
#define SAMPLE_FOR_DFT 256 // number of samples used for the DFT
```

The less number of samples used, the faster the algorithm. Using 128 samples or less, (apart from less interactions in the loop), has the additional advantage of storing the intermediate results in 32-bit values.

The implementation for calculating other harmonics require a different Sin_coef_k vector. This vector has to be calculated with Equation 11.

### 6.2 Test Result

The algorithm performance was tested with five different data sets, each one having 256 16-bit inputs.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Data Set Name (check table 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1&lt;sup&gt;st&lt;/sup&gt;</td>
<td>Inputsignal1</td>
</tr>
<tr>
<td>2&lt;sup&gt;nd&lt;/sup&gt;</td>
<td>Inputsignal2</td>
</tr>
<tr>
<td>3&lt;sup&gt;rd&lt;/sup&gt;</td>
<td>Inputsignal3</td>
</tr>
<tr>
<td>4&lt;sup&gt;th&lt;/sup&gt;</td>
<td>Inputsignal4</td>
</tr>
<tr>
<td>5&lt;sup&gt;th&lt;/sup&gt;</td>
<td>Inputsignal5</td>
</tr>
</tbody>
</table>
Figure 15. Terminal window output for the DFT, THD, and fundamental RMS tests
Figure 16. Real component calculation with errors

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Expected Value</th>
<th>256 Samples Result</th>
<th>128 Samples Result</th>
<th>64 Samples Result</th>
<th>32 Samples Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st</td>
<td>0 (0.0000%)</td>
<td>0 (−1%)</td>
<td>0 (−1%)</td>
<td>0 (−1%)</td>
<td></td>
</tr>
<tr>
<td>2nd</td>
<td>23169.8</td>
<td>23169 (0.0035%)</td>
<td>23169 (0.0035%)</td>
<td>23168 (0.0078%)</td>
<td>23169 (0.0035%)</td>
</tr>
<tr>
<td>3rd</td>
<td>−8668.5</td>
<td>−8669 (0.0058%)</td>
<td>−8607 (0.7095%)</td>
<td>−8484 (2.1284%)</td>
<td>−8243 (4.9086%)</td>
</tr>
<tr>
<td>4th</td>
<td>11254.9</td>
<td>11254 (0.0080%)</td>
<td>11191 (0.5678%)</td>
<td>11062 (1.7139%)</td>
<td>10797 (4.0685%)</td>
</tr>
<tr>
<td>5th</td>
<td>2586.4</td>
<td>2586 (0.0155%)</td>
<td>2584 (0.0928%)</td>
<td>2578 (0.3248%)</td>
<td>2554 (1.2527%)</td>
</tr>
</tbody>
</table>

Figure 17. Imaginary component calculation with errors

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Expected Value</th>
<th>256 Samples Result</th>
<th>128 Samples Result</th>
<th>64 Samples Result</th>
<th>32 Samples Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st</td>
<td>32767</td>
<td>32765 (0.0061%)</td>
<td>32766 (0.0031%)</td>
<td>32765 (0.0061%)</td>
<td>32765 (0.0061%)</td>
</tr>
<tr>
<td>2nd</td>
<td>23169.8</td>
<td>23169 (0.0035%)</td>
<td>23169 (0.0035%)</td>
<td>23168 (0.0078%)</td>
<td>23169 (0.0035%)</td>
</tr>
<tr>
<td>3rd</td>
<td>11640.8</td>
<td>11640 (0.0069%)</td>
<td>11639 (0.0155%)</td>
<td>11636 (0.0412%)</td>
<td>11624 (0.1443%)</td>
</tr>
<tr>
<td>4th</td>
<td>9054.4</td>
<td>9054 (0.0044%)</td>
<td>9054 (0.0044%)</td>
<td>9057 (0.0287%)</td>
<td>9070 (0.1723%)</td>
</tr>
<tr>
<td>5th</td>
<td>20695.2</td>
<td>20694 (0.0058%)</td>
<td>20694 (0.0058%)</td>
<td>20694 (0.0058%)</td>
<td>20694 (0.0058%)</td>
</tr>
</tbody>
</table>

Table 14. DFT Calculation times

<table>
<thead>
<tr>
<th>Dataset</th>
<th>256 Samples</th>
<th>128 Samples</th>
<th>64 Samples</th>
<th>32 Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st</td>
<td>1318 μs</td>
<td>170 μs</td>
<td>85 μs</td>
<td>46 μs</td>
</tr>
</tbody>
</table>
Total Harmonic Distortion (THD)

The THD of a signal is shown with the following equation:

$$THD = \sqrt{\frac{V_2^2 + V_3^2 + V_4^2 + \ldots + V_n^2}{V_1^2}}$$

\[ Eqn. 17 \]

$V_1$ is the amplitude of the fundamental frequency, and $V_2$, … $V_n$ are the amplitude of the harmonics.

The fundamental RMS voltage was calculated using the following equation:

$$V_{\text{rms, fundamental}} = \sqrt{\frac{V_{\text{real}}^2 + V_{\text{img}}^2}{2}}$$

\[ Eqn. 18 \]

$V_{\text{real}}$ is the real part of the DFT phasor and $V_{\text{img}}$ is the imaginary part of the phasor.

7.1 Implementation

The input parameters for the THD are two 16-bit unsigned data values. One for the total RMS and another for the fundamental RMS. The output value is a 32-bit unsigned number. The output number is multiplied by 65536 to display the decimal values with a 16-bit resolution.

The THD implemented code is illustrated in Figure 18.

```
// THD
word THD_calc(word Total_RMS, word Fund_RMS){
  if(Total_RMS > Fund_RMS)
    return((word)(sqrt((Total_RMS*Total_RMS) - (Fund_RMS*Fund_RMS))>>16) / Fund_RMS);
  else
    return(0);
}
```

Figure 18. THD implementation

The input for the fundamental RMS algorithms are the DFT phasors that consist of two 32-bit signed values. Its implementation is shown in Figure 13. The output parameter is a 16-bit unsigned value.

The fundamental RMS implemented code is illustrated in Figure 19.
7.2 Test Result

The THD and fundamental RMS performance were evaluated with the same datasets used for the DFT. Please refer to Section 6.2, “Test Result” for the dataset reference.

The results are presented in Table 16.

To calculate the THD and fundamental RMS the following inputs are used:
- DFT phasor of the fundamental frequency
- RMS value calculated by equation 3

The THD and fundamental RMS performance are not affected directly by the number of samples used in the input buffer but are affected by the precision of the input parameters.

The precision of these input parameters affect the outputs precision. (THD and fundamental RMS)

The tests were performed using the DFT phasor and RMS value, both were calculated with 256 samples for the maximum precision of the input values. For information regarding the DFT and RMS algorithms precision, please refer to Section 6, “Discrete Fourier Transform (DFT)” and Section 4, “RMS Calculation.”

### Table 15. THD and fundamental RMS test results

<table>
<thead>
<tr>
<th></th>
<th>1st dataset</th>
<th>2nd dataset</th>
<th>3rd dataset</th>
<th>4th dataset</th>
<th>5th dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vrms fund expected</td>
<td>23170</td>
<td>23170</td>
<td>10263</td>
<td>10214</td>
<td>14748</td>
</tr>
<tr>
<td>Vrms fund obtained</td>
<td>23168</td>
<td>23169</td>
<td>10262</td>
<td>10213</td>
<td>14746</td>
</tr>
<tr>
<td>Vrms fund error</td>
<td>0.00763%</td>
<td>0.00331%</td>
<td>0.00789%</td>
<td>0.01042%</td>
<td>0.01052%</td>
</tr>
<tr>
<td>THD expected</td>
<td>0</td>
<td>0</td>
<td>30071</td>
<td>30887</td>
<td>27577</td>
</tr>
<tr>
<td>THD obtained</td>
<td>608</td>
<td>0</td>
<td>30072</td>
<td>30891</td>
<td>27581</td>
</tr>
<tr>
<td>THD error</td>
<td>—</td>
<td>—</td>
<td>–0.00447%</td>
<td>–0.01393%</td>
<td>–0.01489%</td>
</tr>
<tr>
<td>execution time [µS]</td>
<td>65</td>
<td>44</td>
<td>72</td>
<td>80</td>
<td>72</td>
</tr>
</tbody>
</table>
8 Conclusions

The goal of this application note is to supply information regarding the MCF51EM256 capability for processing algorithms commonly used in metering with a quantitative approach. Several algorithms were implemented and its execution time and precision measured.

As per the double buffer approached for storing the input sampled data, the time of filling one buffer is the time available for processing the other buffer. In this application note a buffer is considered to be filled within 16.667 ms. Each buffer contains a full period of a 60 Hz sine wave. Therefore, the MCF51EM256 would have less then 16.667 ms to do all the algorithm calculations in a set of 256 samples per phase. Some portion of this time should be left for the other application functionalities, as updating the LCD, managing the user interface, manage eventual communications protocols, perform data normalization, and others.

Table 16 illustrates the processing capability of the MCF51EM256 for implementing a 3-phase metering system.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Execution Time [uS]</th>
<th># per phase</th>
<th># of phases</th>
<th>Total</th>
<th>% of total available time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Energy per phase</td>
<td>610</td>
<td>1</td>
<td>3</td>
<td>1950</td>
<td>11.7%</td>
</tr>
<tr>
<td>Active power per phase</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>9</td>
<td>0.1%</td>
</tr>
<tr>
<td>Total RMS per signal with 256 samples</td>
<td>280</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0.0%</td>
</tr>
<tr>
<td>Total RMS per signal with 128 samples</td>
<td>170</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0.0%</td>
</tr>
<tr>
<td>Total RMS per signal with 64 samples</td>
<td>110</td>
<td>2</td>
<td>3</td>
<td>780</td>
<td>4.7%</td>
</tr>
<tr>
<td>Total RMS per signal with 32 samples</td>
<td>80</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0.0%</td>
</tr>
<tr>
<td>Apparent Power</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>0.0%</td>
</tr>
<tr>
<td>Reactive Power</td>
<td>40</td>
<td>1</td>
<td>3</td>
<td>120</td>
<td>0.7%</td>
</tr>
<tr>
<td>Power Factor</td>
<td>90</td>
<td>1</td>
<td>3</td>
<td>270</td>
<td>1.6%</td>
</tr>
<tr>
<td>DFT per signal (256 samples) per signal</td>
<td>1320</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0.0%</td>
</tr>
<tr>
<td>DFT per signal (128 samples) per signal</td>
<td>170</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0.0%</td>
</tr>
<tr>
<td>DFT per signal (64 samples) per signal</td>
<td>85</td>
<td>6</td>
<td>3</td>
<td>1800</td>
<td>10.8%</td>
</tr>
<tr>
<td>DFT per signal (32 samples) per signal</td>
<td>46</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0.0%</td>
</tr>
<tr>
<td>THD (includes fundamental RMS) per signal</td>
<td>80</td>
<td>1</td>
<td>3</td>
<td>240</td>
<td>1.4%</td>
</tr>
<tr>
<td>Application free time</td>
<td></td>
<td></td>
<td></td>
<td>11495</td>
<td>69.0%</td>
</tr>
<tr>
<td>Total metering algorithms time</td>
<td></td>
<td></td>
<td></td>
<td>5172</td>
<td>31.0%</td>
</tr>
<tr>
<td>Total time</td>
<td></td>
<td></td>
<td></td>
<td>16667</td>
<td>100.0%</td>
</tr>
</tbody>
</table>
Figure 20. Percentage of CPU performance used for implemented algorithms
Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. “Typical” parameters that may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including “Typicals”, must be validated for each customer application by customer's technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.