Freescale Semiconductor
Application Note

Document Number: AN5007
Rev. 0, 10/2014

Application Hints For Using Freescale Metering Libraries in 3-phase Power Meters

by Albert Chen and Shawn Shi

1 Overview

This application note explains the best methods for using Freescale metering libraries in three-phase power meters.

The metering libraries include an FFT library and a Filter library. They both support one-phase, two-phase, and three-phase use cases.

The supported tool chain includes IAR and CodeWarrior.

2 Metering libraries

The metering algorithms perform computations in either the time or frequency domain.

The Filter library calculates all billing and non-billing quantities in the time domain and the FFT library does the same in the frequency domain.

Contents

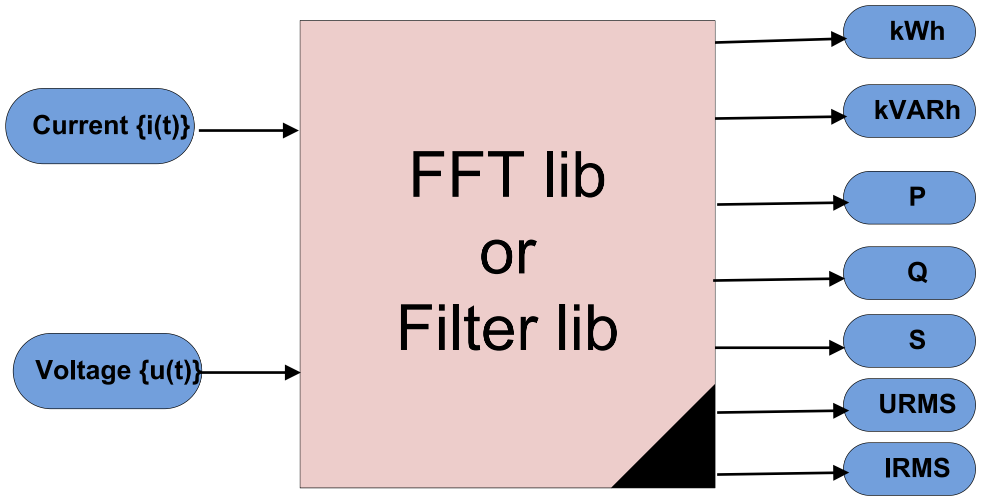

Figure 1. Summary of library

At the application level, a meter can sample the voltage and current signal, send that sampled information to the library. The library can then output useful information such as:

- Active energy (kWh)
- Reactive energy (kVARh)
- Active power (P)
- Reactive power (Q)
- Apparent power (S)
- RMS voltage (URMS)
- RMS current (IRMS)

2.1 Filter library

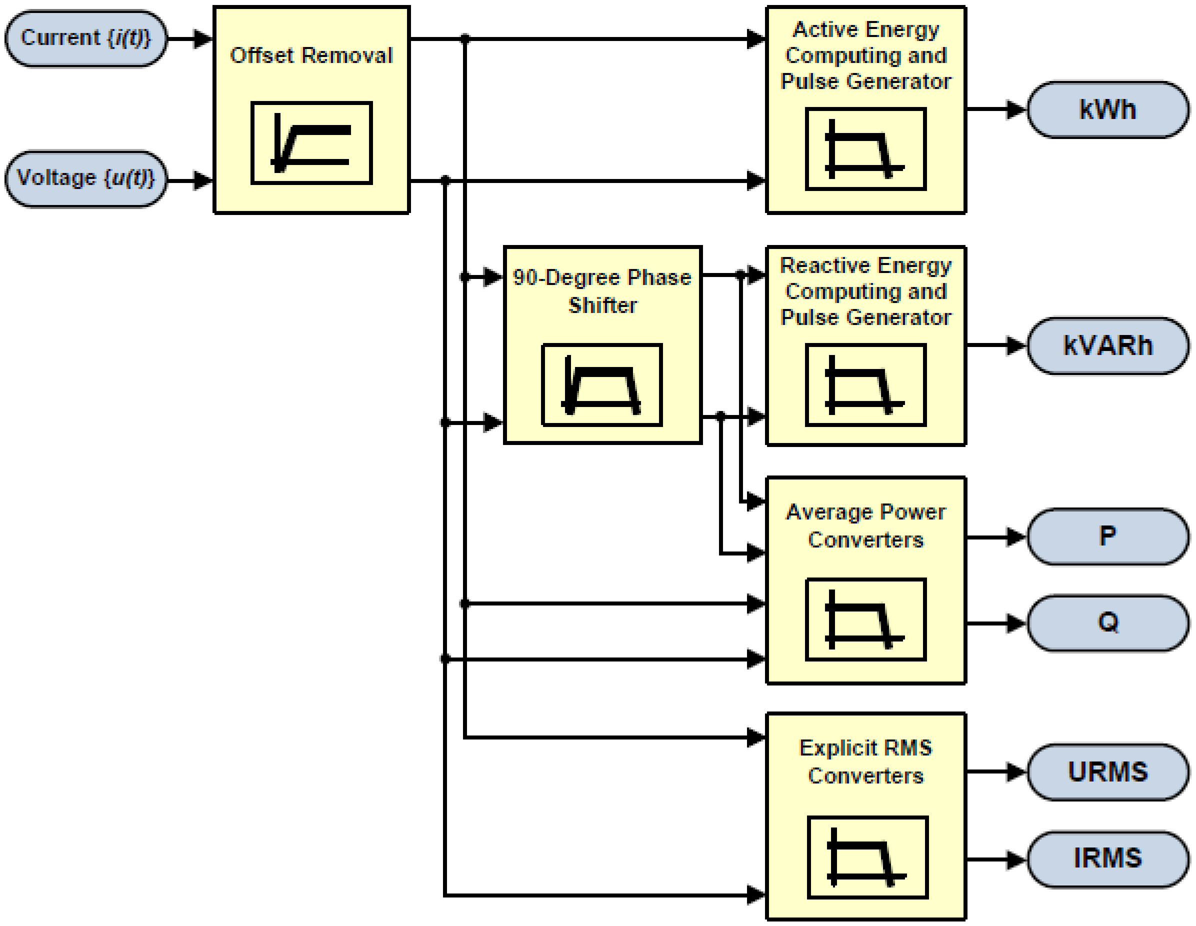

Figure 2. Filter library framework

For more information, please refer to the “KM3x: Kinetis KM3x MCU Family” page at freescale.com. Select the Documentation tab and look for AN4265, “Filter-Based Algorithm for Metering Applications.”

The Filter library includes

- Filter library files: meterlib.a meterlib.h
- Math library files: fraclib.a fraclib.h, …

NOTE

The file names in the library may be changed in a future release. Please refer to the related application note.

2.2 FFT library

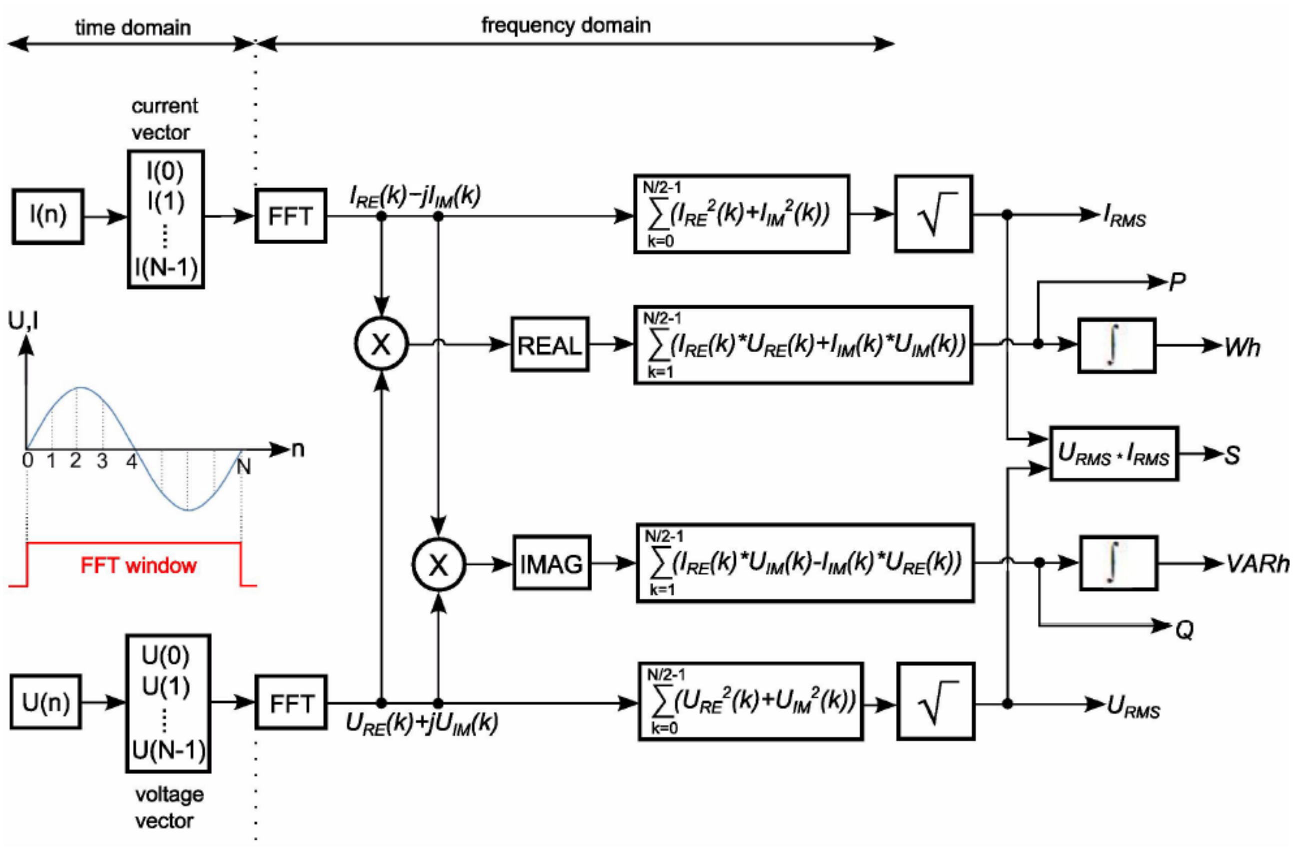
time domain
frequency domain
current vector
I(n)
I(0)
I(1)
I(N-1)
FFT
$I_{RE}(k)-jI_{IM}(k)$
$\sum_{k=0}^{N/2-1}(I_{RE}^2(k)+I_{IM}^2(k))$
I_{RMS}
P
REAL
$\sum_{k=1}^{N/2-1}(I_{RE}(k)*U_{RE}(k)+I_{IM}(k)*U_{IM}(k))$
Wh
$U_{RMS} \cdot I_{RMS}$
S
IMAG
$\sum_{k=1}^{N/2-1}(I_{RE}(k)*U_{IM}(k)-I_{IM}(k)*U_{RE}(k))$
VARh
Q
U,I
n
N
FFT window
U(n)
U(0)
U(1)
U(N-1)
$U_{RE}(k)+jU_{IM}(k)$
$\sum_{k=0}^{N/2-1}(U_{RE}^2(k)+U_{IM}^2(k))$
U_{RMS}
voltage vector

Figure 3. FFT library framework

For more information, please refer to the “KM3x: Kinetis KM3x MCU Family” page at freescale.com. Select the Documentation tab and look for AN4255, “FFT-Based Algorithm for Metering Applications,” and AN4847, “Using an FFT on the Sigma-Delta ADCs.”

The FFT library includes:

- FFT library files: meterlibFFT2.a metering2.h, …
- Math library files: fraclib.a fraclib.h, …

NOTE

The file names in the library may be changed in a future release. Please refer to the related application notes.

2.3 Common note for filter and FFT libraries

Together with the metering libraries we deliver a library with simple filters and math functions implemented in fractional arithmetic. This library is represented by the files:

- fraclib.a
- fraclib_inlines.h

- fraclib.h

All filter and math functions within this library are intended to be used predominantly within the metering libraries, so they may not work correctly with all possible input arguments; for example, when filter coefficients are >1.0 or when dividing negative numbers. These simplifications within the fractional library allowed us to maximize the performance of the metering libraries.

3 Library performance

One important feature of electricity meters, including electronic meters, is metering accuracy.

The accuracy of the FFT and the filter are almost identical.

3.1 Accuracy of FFT/Filter

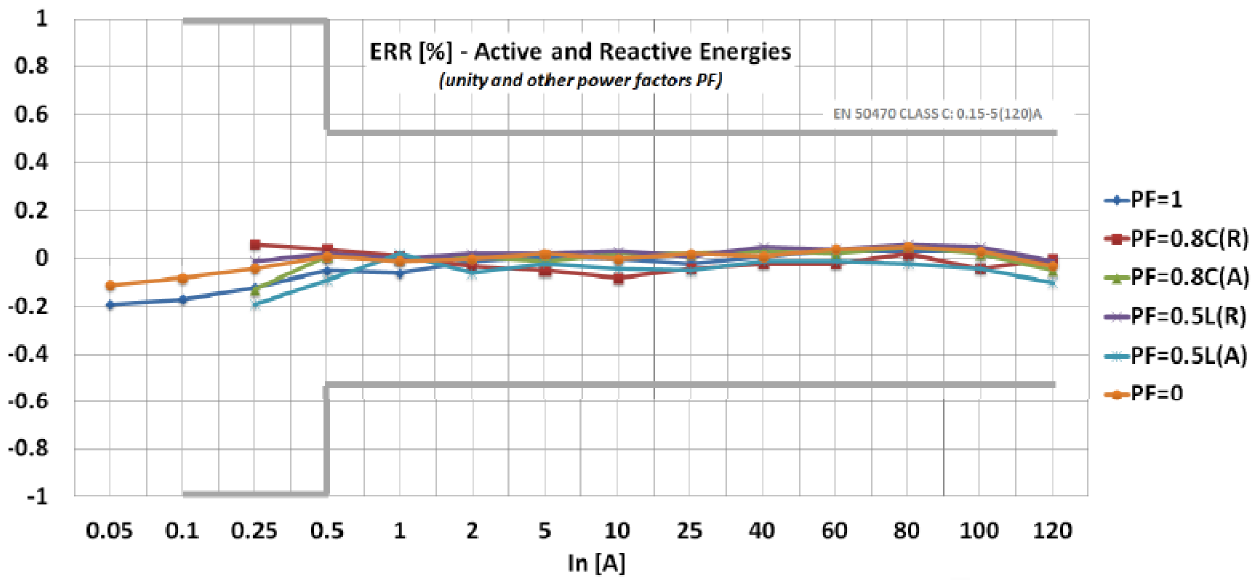

Figure 4. FFT/filter accuracy

3.2 FFT performance

Table 1. Computing performance of FFT library[1]

Number of FFT samples	Computing time [ms]	MCU execution cycles
8	0.24	11513
16	0.55	26384
32	1.13	54208
64	2.42	116093
128	5.36	257131

1 CPUCLK = 47.972352 MHz, Compiler optimization = high speed, f_{inp} = 50 Hz, Cartesian form of the FFT, ARM Cortex-M0+ core.

3.3 Filter performance

Table 2. Computing performance of Filter library[1]

Phase number	Computing time [ms]	MCU execution cycles
1PH	0.20	9586
2PH	0.38	18130
3PH	0.63	30218

[1] CPUCLK = 47.972352 MHz, Compiler optimization = high speed, f = 50 Hz, ARM Cortex-M0+ core.

3.4 Example of resource usage

Here is the resource usage of a typical three-phase power meter design.

Table 3. Resource usage

Resource usage (3PH)	Flash memory	RAM	CPU loading	CAL period
Filter	34 KB	4.3 KB	75%	1200 Hz
FFT (interpolation) N = 64	29 KB	8 KB	70%	50 Hz
FFT (synchronous) N = 64	22 KB	5 KB	51%	50 Hz

NOTE

The flash resource includes only application and metering library code size.

4 Filter library usage

4.1 Requirements

- Sampled values are in 24-bit data format.
  If sampled values are not 24-bit data format, please convert to 24-bit.
  For example, the sampled value for voltage is 16-bit, as in uint16 adc_voltage.
  Please define a new variable such as: Frac32 u24_samplePh2;
  Then convert it to 24-bit: u24_samplePh2 = (Frac32)temp16 << 8.
  Then the variable u24_samplePh2 may be used by meterlib.
- The filter coefficient is based on 1200 Hz.
  Therefore the calculation frequency is 1200 Hz.
  If you want to change to another frequency, please re-create the filter coefficient table using the Filter-Based Metering Algorithm Config Tool. This tool is included with the filter library.

4.2 APIs

1. **KWH_PULS_NUM(x): represents 50% of the energy of one active pulse**

#define KWH_PULS_NUM(x) FRAC48(((5e2/(x))/(U_MAX*I_MAX/3600/CALCFREQ)))

X: constant pulse number—default is 400

CALCFREQ = 1200

For example:

KWH_PULS_NUM(PULSE_NUM)

The real meaning of this macro:

Suppose the sample interval time is t(f = 1/t = CALCFREQ), when condition

P1 × t + P2 × t + P3× t + … = 3600 × 1000 / x = the energy of one active pulse

Therefore

P × t = 3600 × 1000/x

P = 3600 × 1000/x/t = 3600 × 1000 × CALCFREQ/x

Therefore

0.5 × P = 3600 × 500 × CALCFREQ/x = (5e2/x) / (1 / 3600 / CALCFREQ)

U_MAX × I_MAX represents MAX power.

Mathematical normalization result:

KWH_PULS_NUM(x) = 0.5 × P/ (U_MAX × I_MAX)=

= (5e2/x) / (U_MAX × I_MAX / 3600 / CALCFREQ)

2. **KVARH_PULS_NUM(x): represents 50% of the energy of one reactive pulse**

   #define KVARH_PULS_NUM(x) FRAC48(((5e2/(x))/(U_MAX × I_MAX/3600/CALCFREQ)))

   Similar with KWH_PULS_NUM(x)

3. **do_filter: calculate the energy**

   The frequency of this function call is 1200 Hz.

   The AFE module will generate a callback function when data are converted. Within the callback function, data are scaled by gain and offsets found during calibration, then metering library functions are called.

void do_filter(tMETERLIB3PH_DATA *p,

```
Frac24 u1Q, Frac24 i1Q,
Frac24 u2Q, Frac24 i2Q,
Frac24 u3Q, Frac24 i3Q,
Frac32 *whCnt, Frac32 *varCnt,
Frac64 whRes, Frac64 varRes);
```

Input:

```
tMETERLIB3PH_DATA *p: includes the filter coefficient
u1Q, i1Q, …: the calibrated sampled result
whRes, varRes: pulse resolution
```

Input, Output:

```
whCnt, varCnt: output pulse number. They would be changed if energy reach to threshold.
For example:
do_filter(mlib,
-L_mul (u1 - u1_offset, u1_gain),
-L_mul (i1 - i1_offset, i1_gain),
-L_mul (u2 - u2_offset, u2_gain),
-L_mul (i2 - i2_offset, i2_gain),
-L_mul (u3 - u3_offset, u3_gain),
-L_mul (i3 - i3_offset, i3_gain),
&wh_cnt, &varh_cnt,
KWH_PULS_NUM(400), KVARH_PULS_NUM(400));
void do_filter(tMETERLIB3PH_DATA *p,
     Frac24 u1Q, Frac24 i1Q,
     Frac24 u2Q, Frac24 i2Q,
     Frac24 u3Q, Frac24 i3Q,
     Frac32 *whCnt, Frac32 *varCnt,
     Frac64 whRes, Frac64 varRes)
{
     METERLIB3PH_RemoveDcBias(p, u1Q, i1Q, u2Q, i2Q, u3Q, i3Q);
     METERLIB3PH_CalcWattHours(p, whCnt, whRes);
     METERLIB3PH_CalcVarHours(p, varCnt, varRes);
     METERLIB3PH_CalcAuxiliary(p);
}
```

4. **METERLIB3PH_ReadResultsPhx**

```
void METERLIB3PH_ReadResultsPh1(tMETERLIB3PH_DATA *p, double *urms,
     double *irms, double *w, double *var,
     double *va, double *umax, double *imax);
```

Returns auxiliary variables: IRMS, URMS, P, Q, and S

It is okay to call this function anytime.

For example:

```
METERLIB3PH_ReadResultsPh1((tMETERLIB3PH_DATA*)&mlib,
     (double*)&U_RMS_Ph1,
     (double*)&I_RMS_Ph1,
     (double*)&P_Ph1,
     (double*)&Q_Ph1,
     (double*)&S_Ph1,
     (double*)&umax_Ph1,
     (double*)&imax_Ph1);
```

5 FFT library usage

5.1 Requirements

Call FFT function **PowerCalculation** every whole period.

For example:

Suppose the grid frequency is 50 Hz—each period is 20 ms.

In this scenario, call the function **PowerCalculation** every 20 ms.

Suppose FFT N = 64. Then every 20 ms, ensure the sampling of 64 U/I values and send these values to the FFT function.

5.2 APIs

1. **N_SAMPLES**: number of samples for FFT computing (only 8, 16, 32, 64, or 128)
   Defined in fft2.h
   MAX_FFT_SEND = N_SAMPLES / 2
2. **PowerCalculation**: computes the whole power vector

```
typedef long afetype;
typedef unsigned long afeutype;
void PowerCalculation(afetype *voltage, afetype *current, Power_Vector *powers,
afeutype *mU, afeutype *mI);
```

Parameters:

- voltage: [input], array size must be N_SAMPLES
- current: [input], array size must be N_SAMPLES
- Powers: [output], the calculated result
- mU: [output], the voltage magnitudes; array size must be MAX_FFT_SEND
- mI:[output], the current magnitudes; array size must be MAX_FFT_SEND

NOTE

mU and mI are used for harmonic analysis.

For example:

```
afetype v_sample[N_SAMPLES]; //enter the sampled voltage value into it
afetype i_sample[N_SAMPLES]; //enter the sampled current value into it
Power_Vector p_result;
afeutype magnFFTU [MAX_FFT_SEND];
afeutype magnFFTI[MAX_FFT_SEND];
PowerCalculation(v_sample, i_sample, &p_result, magnFFTU, magnFFTI);
```

6 Sampling process

Here is the background for the sampling process.

1. In the current three-phase solution, we used an AFE COC event to trigger the XBAR. The XBAR is connected to the QT timer, and the timer triggers the SAR ADC sample.
2. The CT is used for sampling the current. CT is equal to inductance, so voltage is ahead of current.
3. The AFE samples the current, and the SAR ADC samples the voltage.
4. In the current solution, read U/I values in the AFE interrupt. At this point, the current value is correct, but voltage is the previous point value because the AFE COC event just started to trigger the QT, and the SAR ADC has not started sampling.
5. Read the ADC values in the AFE interrupt.
6. The reserved SAR ADC conversion time is 9 µs.

After calibration, we can confirm the phase compensation between voltage and current.

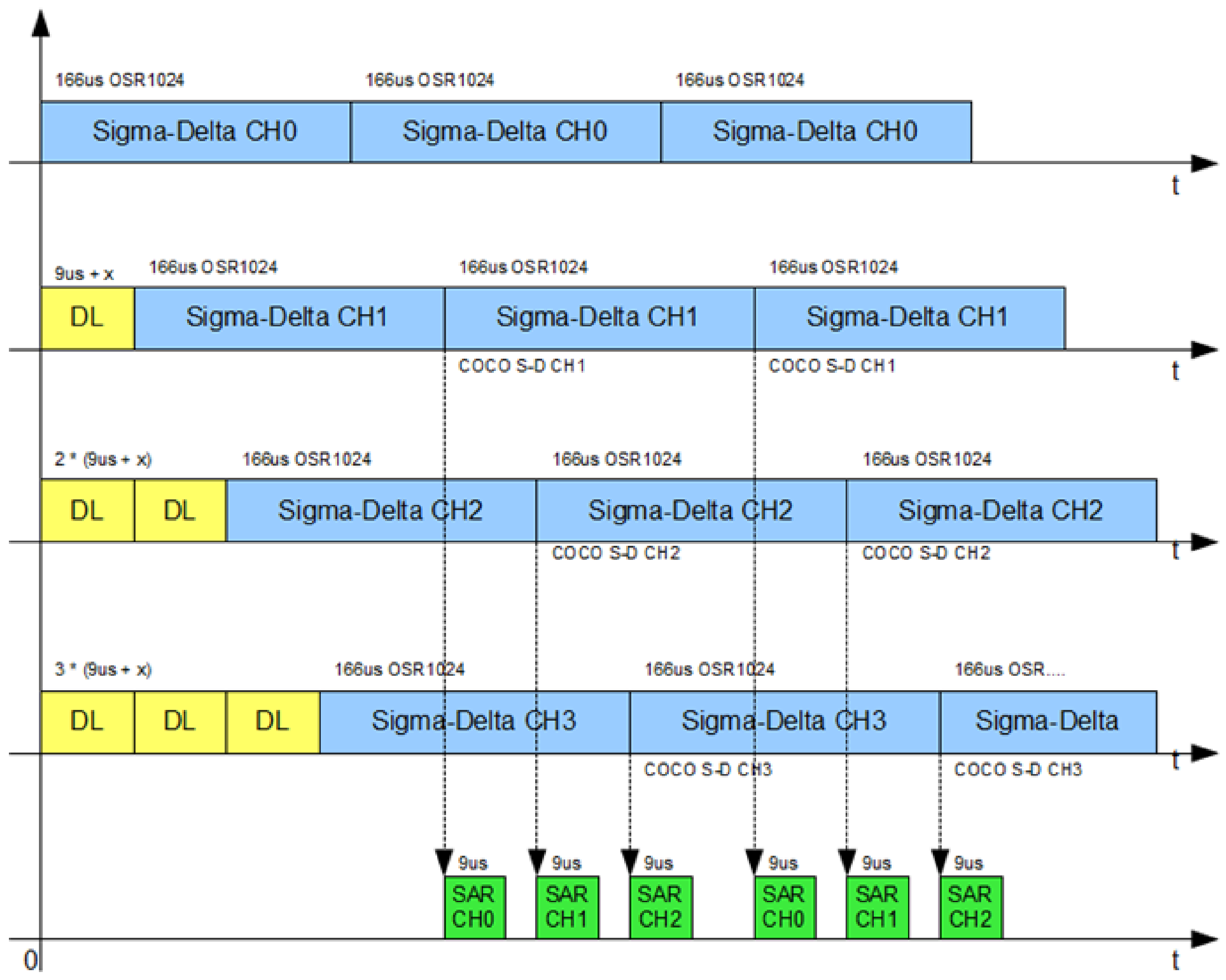

Figure 5. Filter SYNC process

6.1 Filter

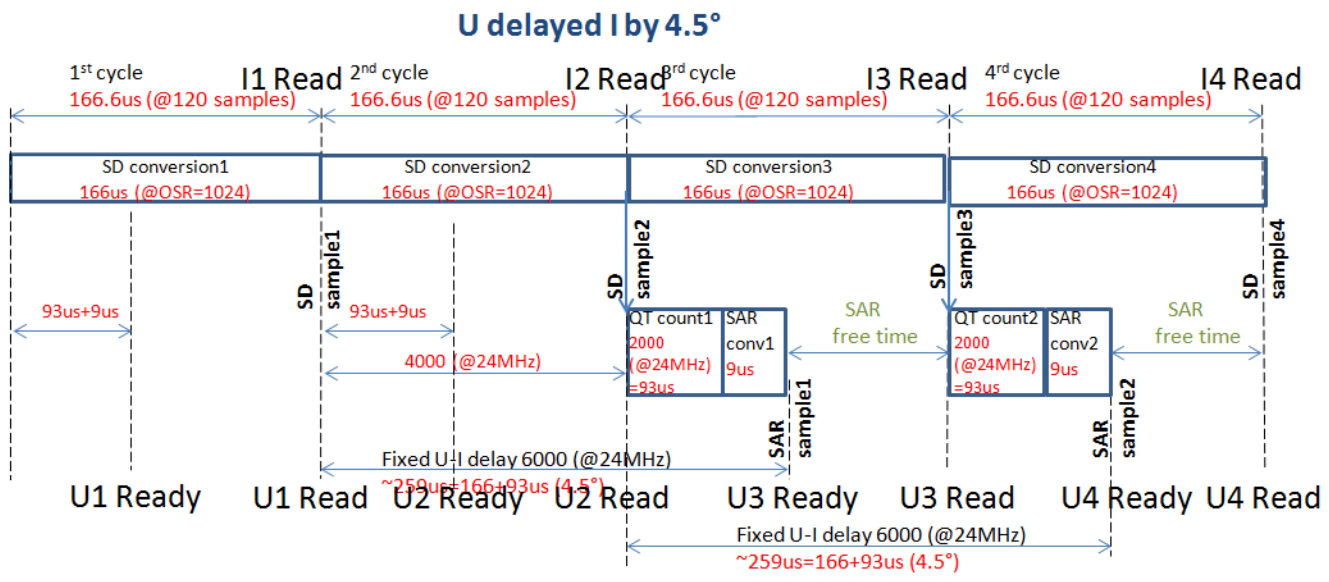

Figure 6. Filter sampling process

Sampling process:

1. In this case the AFE uses continuous mode with a sample rate of 6000 Hz, so the AFE sample period is **166** µs.

   Grid frequency is 50Hz, so every period's sample points equal 6000/50 = 120.

   Samples = **120**

   Degrees between samples: 360 / 120 = **3** degrees
2. A 24 MHz timer is used to implement the U-I delay.
3. Delay counter = delay_angle × 24000000 / (2 * * 50) = delay_angle × 240000 /

   Here the unit of the delay angle is a radian.

   Delay counter = delay_angle × **240000 /**

   If the unit of delay_angle is a degree, then

   Delay counter = delay_angle × 4000 / 3

   Three degrees = 4000 QT counter and 0.1 degree = 133 QT counter
4. **Delay time(us) = delay counter / 24**

   The delay counter is based on a 24 Mhz timer.

   Therefore

   Delay time = delay counter / 24000000 s = delay counter / 24 µs

   For example, suppose the delay counter = 3000; then delay time is 3000/24 = 125 µs.
5. Delay angle (unit is degree) **= delay time(µs)** × **18 / 1000**

   Delay counter = delay_angle × 4000 / 3

   Therefore

delay_angle = Delay counter × 3 / 4000 = delay time × 24 × 3 / 4000
= delay time × 18 / 1000

6. **(U1, I2), (U2, I3)**, … are used for calculation.
U delay I 4.5 degree = 3 degree + 1.5 degree; sample interval is 3 degrees, so U/I pair should use the Ui, Ii+1

6.2 FFT

N = 64, f= 50Hz

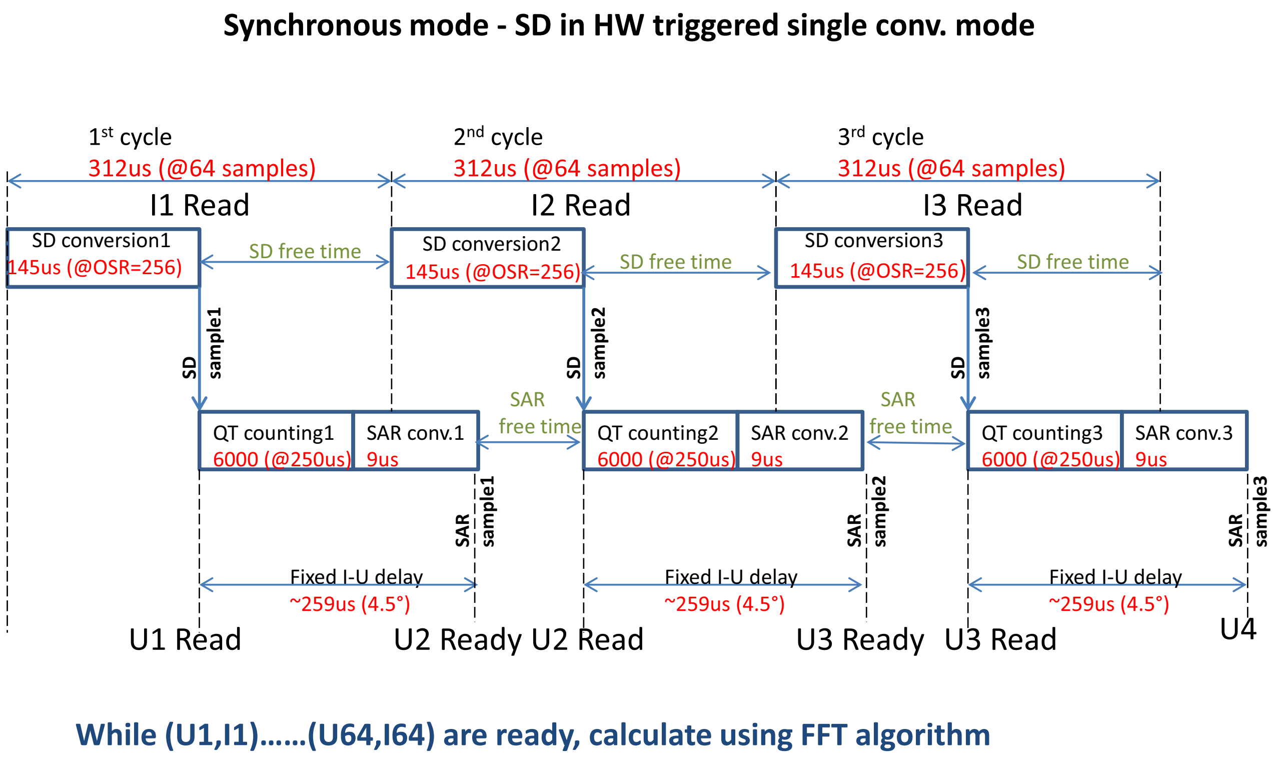

Figure 7. FFT sample process

1. Degrees between samples: 360 / 64 = **5.625 degrees**
   Sample interval = 1000000 μs / N / f, when N=64, f = 50
   Interval = 1000000 μs / 64 / 50 = **312** μs
   If a 24 Mhz timer is used, then
   Interval counter = 24000000 / N / f = 7500
2. Delay counter = 4.5 degrees × 4000/3 = **6000**, delay time = 6000/24 = **250** μs.
3. **(U1, I1), (U2, I2)**, … are used for FFT calculation
4. CT delay angle = Sample interval – delay_time, CT delay angle is constant
   The unit of the CT delay angle is the timer counter.
   Therefore
      CT delay angle = 24000000 / N / f – delay_time

Here we suppose that N = 64, so

CT delay angle = 375000 / f – delay_time

After calibration, we get the CT delay time value under **N=64, f= 50**.

Named QT_DELAY(50 Hz), it is a constant value

CT delay angle = 7500 – QT_DELAY(50) = 375000 / f – delay_time

Therefore

delay_time = 375000 / f – 7500 + QT_DELAY(50)

Here the unit of delay_time is timer counter (based on 24 Mhz QT timer)

With the FFT algorithm we should measure the signal frequency. So **frequency(f)** is a known value, while QT_DELAY(50) was derived from the calibration phase.

In this way it is possible to calculate the delay time with any signal frequency.

7 Pulse output

7.1 Macro PULSE_THRES: energy of one pulse

#define PULSE_THRES (3600 × 1000 × 10000ll / PULSE_NUM)

Here PULSE_NUM is 400, so

PULSE_THRES = 90000000

Only when

P × t ≥ 3600 × 1000 / 400

Output one pulse.

So

P ≥3600 × 1000 × f / 400

Here f = 10000Hz.

So

P ≥ 3600 × 1000 × 10000 / 400 = 90000000

7.2 Pulse output algorithm

Here one timer interrupt is used. The frequency is 10000 (100 μs).

This interrupt priority is lower than the AFE/ADC interrupt, but higher than others.

In every timer interrupt:

```
energy_active += p;
if (energy_active Š PULSE_THRES) {
    energy_active -= PULSE_THRES;
    output active pulse
}else if (energy_active £ (0 - PULSE_THRES)) {
    energy_active += PULSE_THRES;
    output active pulse
}
```

The reactive pulse output is similar to the active pulse output.

8 How to choose a metering library: FFT or Filter

1. If you want **harmonic** information, use the FFT library.
   Only the FFT library supports harmonic information.
2. The FFT library requires the **signal frequency**, so the FFT needs more hardware resources.
   The FFT needs one **CMP** module and at least one **timer channel**.
   Use these resources to measure the signal frequency.
   Filter library doesn’t use the frequency information.
3. FFT uses **more RAM resources**.
   FFT uses a double buffer—one is used for calculation, and the other for saving sampled values.
   Therefore FFT uses more RAM.

9 References

The following documents are available on freescale.com:

- AN4265, *Filter-Based Algorithm for Metering Applications,* Rev. 1, 12/2013
- AN4255, *FFT-Based Algorithm for Metering Applications,* Rev. 1, 10/2013
- AN4847, *Using an FFT on the SD ADCs,* Rev. 0, 12/2013

Document Number: AN5007
Rev. 0
10/2014