NXP Semiconductors
Application Note

Document Number: AN4265
Rev. 4, 04/2016

Filter-Based Algorithm for Metering Applications

By: Martin Mienkina

1. Introduction

High accuracy metering is an essential feature of an electronic power meter application. Metering accuracy is a most important attribute because inaccurate metering can result in substantial amounts of lost revenue. Moreover, inaccurate metering can also undesirably result in overcharging to customers. The common sources of metering inaccuracies, or error sources in a meter, include the sensor devices, the sensor conditioning circuitry, the Analog Front-End (AFE), and the metering algorithm executed either in a digital processing engine or a microcontroller.

Contents

© 2016 NXP B.V.

The critical task for a digital processing engine or a microcontroller in a metering application is accurate computation of active energy, reactive energy, active power, reactive power, apparent power, RMS voltage, and RMS current. The active and reactive energies are sometimes referred to as billing quantities. Their computations must be compliant with the EN50470-1 and EN50470-3 European standards for electronic meters of active energy class B and C, and IEC 62053-21 and IEC 62052-11 international standards for electronic meters of active energy classes 2 and 1, and the IEC 62053-23 international standard for static meters of reactive energy classes 2 and 3.

The remaining quantities are calculated for informative purposes and they are referred as non-billing.

The metering algorithms perform computation in either time or frequency domain. This application note describes an accurate and scalable metering algorithm that is intended for use in electronic meters, further referred to as the Filter-Based Metering Algorithm. This algorithm calculates all billing and non-billing quantities in the time domain, with extensive support of the Finite Impulse Response (FIR) and Infinite Impulse Response (IIR) digital filters [1] and [2].

The Filter-Based Metering Algorithm can be easily integrated into an electronic power meter application. The algorithm requires only instantaneous voltage and current samples to be provided at constant sampling intervals. These instantaneous voltage and current samples are usually measured by an AFE with the help of a resistor divider, in the case of a phase voltage measurement, and a shunt resistor, current transformer or a Rogowski coil in the case of a phase current measurement. All current measurement sensors introduce a phase shift into current measurement. Therefore, it is necessary to align the phases of the instantaneous voltage and current samples using either software phase correction method included in the Filter-Based Metering Algorithm or with the aid of delayed sampling before using them.

The software configuration tool is available to easily set up the Filter-Based Metering Algorithm. The tool is intended to tune digital filters to match the required performance and to generate a C-header file with configuration data specific to the power meter type. The tool automates the procedure of the algorithm setup and optimization, while providing a rough estimate of the required computational load.

There further follows a block diagram and a brief description of the Filter-Based Metering Algorithm in a one-phase power meter configuration.

2. Block diagram

The following figure shows a block diagram of the Filter-Based Metering Algorithm in a typical one-phase power meter application. The current and voltage measurements are represented by $i(t)$ and $u(t)$ signal sources. These sources provide phase-aligned instantaneous current and voltage samples at constant sampling intervals. The new voltage and current samples trigger a recalculation of all the algorithm blocks. After each recalculation, new billing and non-billing quantities will become available. All calculated quantities are usually displayed on the LCD and archived in a database for post-processing and reading through the Automated Meter Reading (AMR) communication interface. In addition, active and reactive energies also drive their respective pulse output LEDs for calibration and testing purposes.

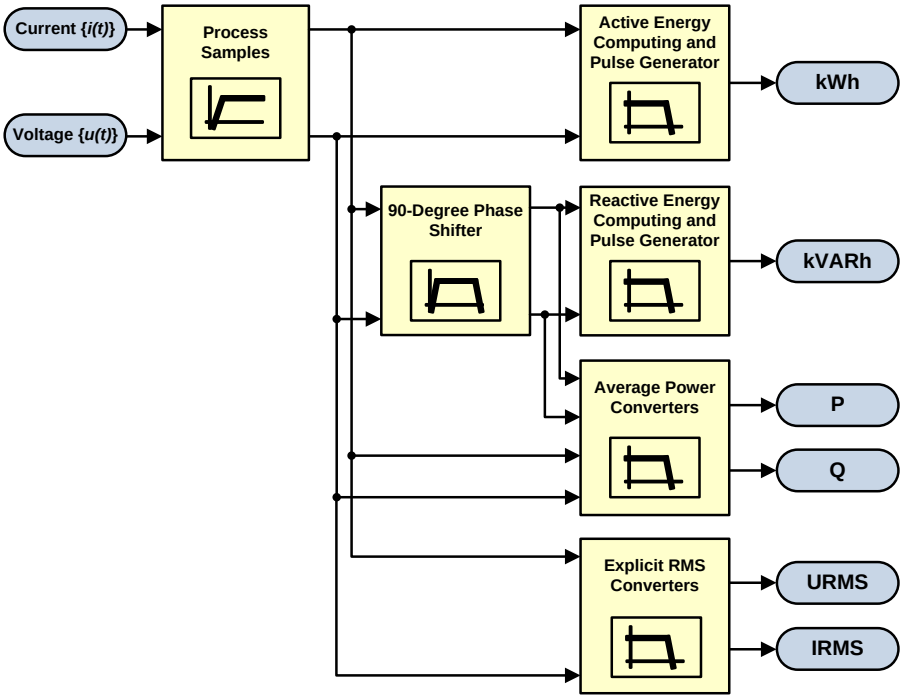


Figure 1. **Block diagram of the filter-based metering algorithm**

The algorithm consists of several blocks mostly comprising the Infinite Impulse Response (IIR) and Finite Impulse Response (FIR) digital filters.

The first block in the signal flow is the samples processing. This block removes offset from the instantaneous voltage and current samples, performs optional sensor phase shift correction, and returns the order of voltage waveform sequences in a three-phase system. If the Direct Current (DC) offset was stable and deterministic, its removal would be performed by simple subtraction. However, in a real application, most analog components unintentionally insert a DC offset as part of the signal conditioning, amplification, and analog-to-digital circuits. Since the DC offset of the analog circuits is not constant but varies with the process, supply voltage, and temperature, a robust algorithm must be used for its removal. Due to this fact, this block represents the high-pass first order IIR filters, which remove any DC and low-frequency components from the alternating voltage and current measurements. For more information, refer to Infinite impulse response filter.

The second block is essential for reactive energy calculation, and is called the 90-degree phase shifter. This block represents two special FIR filters, the first is the N-Tap FIR filter that is an approximation of the Hilbert transformer, and the second is the M-Tap FIR filter that compensates for the group delay introduced by the first N-Tap FIR filter. For more information, refer to Ideal Hilbert transformer.

Following blocks in the signal flow diagram are the active and reactive energy computing and pulse generators. These blocks calculate and smooth the active and reactive energies. The smoothing filters are sometimes required to suppress the 100 Hz (120 Hz) component caused by the multiplication of the instantaneous 50 Hz (60 Hz) voltage and current waveforms. The smoothed energy waveforms result in lower jitter of the generated pulses, and thus, a shortening of the power meter testing and calibration time.

The explicit RMS converters are present to transform alternating voltage and current waveforms into RMS values. This method for the RMS value computation requires the numerical square, average, and

square root functions be called every time a new sample of the analyzed signal is available (see Explicit RMS converter).

Finally, the average power converters calculate the active and reactive powers from the new unbiased phase voltage and phase current samples. This power calculation method leverages the low-pass first order IIR filters extensively (see Average power converter).

The Filter-Based Metering Algorithm allows the use of two sampling intervals. Introducing a short and long sampling interval for calculating billing and non-billing quantities, respectively, will lead to significant savings in the computational power.

The general setting of the algorithm can be easily performed by the configuration tool (see Configuration tool). This tool allows the user to tune the metering algorithm interactively with respect to the power meter hardware and firmware capabilities. The configuration session should always terminate by generating a C-header file containing all the configurations and by saving this file to the hard drive.

3. Theory

The Filter-Based Metering Algorithm comprises of several blocks. These blocks represent the Infinite Impulse Response (IIR) and Finite Impulse Response (FIR) digital filters. The digital filters and other calculations performed by the algorithm are based on elementary fractional and integer calculations, such as addition, subtraction, integration, multiplication and square root.

In order to understand these blocks, the basics and tricks of 2’s complement integer and fractional arithmetic are explained in the following section.

3.1. Basics of fixed-point arithmetic

This section explains how numbers are represented in a microcontroller and processed by the Filter-Based Metering Algorithm. The microcontrollers are integrated with an AFE, which converts an analog input signal into its digital representation and stores it in a result register. This figure shows the result register implementation specific to the Kinetis M microcontroller family of devices:

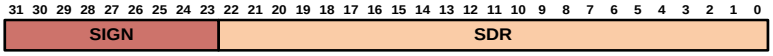


Figure 2. **Kinetis M - AFE result register format**

These devices are integrated with a powerful AFE that produces a digital output scale based on the Oversampling Ratio (OSR). The digital output of each channel is then truncated to a 24-bit signed 2's complement result, which is stored in corresponding channel's result register:

Table 1. **Kinetis M - AFE result register fields**

Field	Description
[31:23]	Sign Bits
SIGN	This field represents sign bits (bits 31 to 23 are filled with sign bits).
[22:0]	Sample Data Result
SDR	This field represents valid sample value in 2’s complement form.

The 2’s complement representation is convenient in implementing DSP algorithms such as IIR and FIR filters. All operations can be performed using 2’s-complement integer or fractional arithmetic [3], [4], and [5].

3.1.1. Signed integer

This format is used for processing data as integers. In this format, the N-bit operand is represented using the Q(N-1).0[1] format (N integer bits). The range of signed integer numbers is as follows:

$$-\mathbf{2}^{(N-1)} \leq \boldsymbol{Integer} \leq \left[\mathbf{2}^{(N-1)} - \mathbf{1}\right] \qquad \textit{Eq. 1}$$

For example, the most negative, signed word that can be represented is –32,768 ($8000), and the most negative, signed long word is –2,147,483,648 (0x80000000). The most positive signed word is 32,767 (0x7FFF), and the most positive signed long word is 2,147,483,647.

Signed integer data format is typically used in controller code, array indexing and address computations, peripheral set-up and handling, bit manipulation, bit-exact algorithms, and other general-purpose tasks.

3.1.2. Signed fractional

In this format, the N-bit operand is represented using the Q0.(N–1) format (1 sign bit, N–1 fractional bits). Signed fractional numbers lie in the following range:

$$-\mathbf{1.0} \leq \boldsymbol{Fractional} \leq \left[+\mathbf{1} - \mathbf{2}^{-(N-1)}\right] \qquad \textit{Eq. 2}$$

For example, the most negative word that can be represented is –1.0, whose internal representation is 0x8000 (word) or 0x80000000 (long word). The most positive word is $1.0\text{-}2^{-15}$ (0x7FFF), and the most positive long word is $1.0\text{-}2^{-31}$ (0x7FFFFFFF).

Using 2's complement signed integers is not convenient for handling to implement digital filters. For example, if two 32-bit words are multiplied, 64 bits are needed to store the result. The size of the required word length increases without bounds as we further multiply numbers together. Although not impossible, it becomes complicated to handle this increase in word-length using signed integer arithmetic.

The problem can be easily handled by using signed fractional numbers in the range −1.0 and $1.0\text{-}2^{-[N-1]}$, instead of signed integers, because the product of two numbers in the range $[-1, 1.0\text{-}2^{-[N-1]}]$ will always be in the same range. Signed fractional data format and arithmetic is typically required for computation-intensive algorithms, such as digital filters, speech coders, vector and array processing, digital control, and other signal processing tasks.

The relationship between the integer interpretation of an N-bit value and the corresponding fractional interpretation is:

$$\boldsymbol{Fractional} = \boldsymbol{Integer}/\mathbf{2}^{(N-1)} \qquad \textit{Eq. 3}$$

The arithmetic operations required by the Filter-Based Metering Algorithm, such as addition, subtraction, multiplication, and square root are discussed in the following subsections.

[1] The Q notation is written as Qm.n, where: Q designates that the number is in the Q format notation (the Texas Instruments representation for signed fixed-point numbers), m is the number of bits set aside to designate the 2's complement integer portion of the number, and n is the number of bits used to designate the fractional portion of the number.

3.1.3. Addition and subtraction

Addition, subtraction, and comparison operations are performed identically for both fractional and integer representations. The Arithmetic Logic Unit (ALU) of the microcontroller does not have to distinguish between the data types for these operations. The source code of the L_add() function that implements 32-bit integer (Q31.0) and fractional (Q0.31) addition, is shown in the following code:

Example 1. **Function for 32-bit Integer and Fractional Addition**

```
static inline frac32 L_add (register frac32 lsrc1, register frac32 lsrc2)
{
  return lsrc1+lsrc2;
}
```

Typical examples of additions are shown in the following table:

Table 2. **Examples of 32-bit addition**

Format	X		Y		Addition, Z=X+Y	
	Signed Fractional	Hexadecimal	Signed Fractional	Hexadecimal	Signed Fractional	Hexadecimal
Q0.31	0.5	0x40000000	0.25	0x20000000	0.75	0x60000000
	0.5	0x40000000	-0.25	0xE0000000	0.25	0x20000000
	-0.5	0xC0000000	-0.25	0xE0000000	-0.75	0xA0000000

The source code of the L_sub() function that implements 32-bit integer and fractional subtraction is shown in the following code:

Example 2. **Function for 32-bit Integer and Fractional Subtraction**

```
static inline frac32 L_sub (register frac32 lsrc1, register frac32 lsrc2)
{
  return lsrc1-lsrc2;
}
```

The following table shows typical examples of subtraction operations:

Table 3. **Examples of 32-bit subtraction**

Format	X		Y		Subtraction, Z=X-Y	
	Signed Fractional	Hexadecimal	Signed Fractional	Hexadecimal	Signed Fractional	Hexadecimal
Q0.31	0.5	0x40000000	0.25	0x20000000	0.25	0x20000000
	0.5	0x40000000	-0.25	0xE0000000	0.75	0x60000000
	-0.5	0xC0000000	-0.25	0xE0000000	-0.25	0xE0000000

NOTE

Addition and subtraction can generate values that are larger than the data format. For example, adding two fractional Q0.15 numbers X=0.55 (0x4666) and Y=0.55(0x4666) causes overflow X+Y= -0.9(0x8CCC). The solution is saturation or data limiting, which is implemented on some DSPs and guarantees that values are always within a given range. On microcontrollers with no hardware support for saturation and data limiting, one has to ensure that algorithms are implemented in a way to prevent overflows. The Filter-Based Metering Algorithm solve this phenomena by using input signals of the phase voltage and phase current samples in 24-bit fractional representation (Q0.23) while performing all mathematical operations in 32-bit fractional format (Q0.31). In this way, the dynamic range for additions and subtractions is extended by eight bits.

3.1.4. Multiplication

The multiplication operation is not same for integer and fractional arithmetic. The result of a fractional multiplication differs from the result of an integer multiplication. The difference amounts to a 1-bit shift of the final result, as illustrated in Figure 3.

Any binary multiplication of two N-bit signed numbers generates a signed result that is 2N–1 bits in length. This (2N–1)-bit result must be properly placed in a field of 2N bits to fit correctly into the on-chip registers. For correct integer multiplication, an extra sign bit is inserted in the MSB to generate a 2N-bit result. For correct fractional multiplication, an extra zero bit is inserted in the LSB to generate a 2N-bit result.

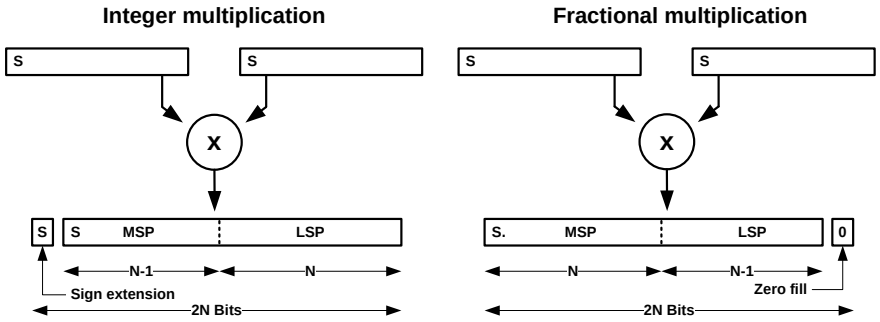


Figure 3. **Comparison of integer and fractional multiplication**

Some DSPs have dedicated instructions to perform integer and fractional multiplication [3]. On general purpose microcontrollers, fractional multiplication can be emulated easily using integer arithmetic. The following code shows the source code of the L_mul() function that implements 32x32=32-bit fractional multiplication in C-language:

Example 3. **Function for 32-bit Fractional Multiplication**

```
static inline frac32 L_mul (register frac32 lsrc1, register frac32 lsrc2)
{
  register frac64 tmp = ((frac64)lsrc1*(frac64)lsrc2);
  return (tmp+tmp)>>32;
}
```

Typical examples of fractional multiplications are shown in the following table:

Table 4. **Examples of 32-bit fractional multiplication**

Format	X		Y		Multiplication, Z=X*Y	
	Signed Fractional	Hexadecimal	Signed Fractional	Hexadecimal	Signed Fractional	Hexadecimal
Q0.31	0.5	0x40000000	0.25	0x20000000	0.125	0x10000000
	0.5	0x40000000	-0.25	0xE0000000	-0.125	0xF0000000
	-0.5	0xC0000000	-0.25	0xE0000000	0.125	0x10000000

3.1.5. Square root

Similarly to the multiplication operation described in previous subsection, square root calculation is also not same for integer and fractional values. The relationship between square root of the fractional and integer N-bit radicands can be expressed as follows:

$$\sqrt{\boldsymbol{Fractional}} = \sqrt{\boldsymbol{Integer}}/\sqrt{\mathbf{2}^{(\boldsymbol{N-1})}} \qquad \textit{Eq. 4}$$

The Filter-Based Metering Algorithm uses square root function for calculating the RMS current and voltage. The square root computation is limited to positive fractional numbers and is based on the *Non-restoring Method* [6]. This method only uses addition, subtraction, and compare operations. The square root is calculated from the known radicand X, the unknown quotient Q, and the unknown remainder R_n, which all satisfy the relation.

$$\boldsymbol{R_n = X - Q^2} \qquad \textit{Eq. 5}$$

The method employs a root extractor that is either added to or subtracted from the partial remainder. The root extractor in the non-restoring method is a function of the quotient digits and constants. The first operation is always subtraction of a constant (0.25). The subsequent operation subtracts or adds the root extractor, depending on whether the remainders are positive or negative. This leads to a new partial remainder. The process continues until the remainder is zero or the desired number of the quotient digit is obtained.

Table 5. **Algorithm for the binary square root by the non-restoring method**

First Reminder	$R_1 = X - 0.25$
Reminder	$R_{n+1} = \begin{cases} R_n - 2^{-n}\left[\sum_{i=1}^{n-1} q_i + 1.25 \times 2^{-n}\right], & R_n \geq 0 \\ R_n + 2^{-n}\left[\sum_{i=1}^{n-1} q_i + 0.75 \times 2^{-n}\right], & R_n < 0 \end{cases}$
Quotient	$Q = \sum_{i=1}^{n} q_i$ $q_i = \begin{cases} 2^{-i}, & R_{i+1} \geq 0 \\ 0, & R_{i=1} < 0 \end{cases}$

The C-language along with a preprocessor guarantees an efficient calculation of the binary square root algorithm on a microcontroller. The source code of the L_sqr() function that implements the 32-bit fractional (Q0.31) square root by the non-restoring method is as follows:

Example 4. **Function for the 32-bit Square Root Calculation by the Non-restoring Method**

```
#define LSQR_STEP(k)                                                \
{                                                                   \
  if(r1>=0)                                                         \
  {                                                                 \
    r1-=((q1+(frac32)FRAC32(1.25/(((frac32)1)<<k)))>>k);            \
    q1+=((frac32)FRAC32(0.5)>>(k-1));                               \
  }                                                                 \
  else                                                              \
  {                                                                 \
    r1+=((q1+(frac32)FRAC32(0.75/(((frac32)1)<<k)))>>k);            \
  }                                                                 \
}

frac32 L_sqr (register frac32 x)
{
  register frac32 q1 = 0l;
  register frac32 r1 = x-(frac32)FRAC32(0.25);

  /* input parameter conditions                                                */
  if (x <= 0l) { return FRAC32(0.0); }

  /* square root calculation using non-restoring method                        */
  LSQR_STEP( 1); LSQR_STEP( 2); LSQR_STEP( 3); LSQR_STEP( 4); LSQR_STEP( 5);
  LSQR_STEP( 6); LSQR_STEP( 7); LSQR_STEP( 8); LSQR_STEP( 9); LSQR_STEP(10);
  LSQR_STEP(11); LSQR_STEP(12); LSQR_STEP(13); LSQR_STEP(14); LSQR_STEP(15);
  LSQR_STEP(16); LSQR_STEP(17); LSQR_STEP(18); LSQR_STEP(19); LSQR_STEP(20);
  LSQR_STEP(21); LSQR_STEP(22); LSQR_STEP(23); LSQR_STEP(24); LSQR_STEP(25);
  LSQR_STEP(26); LSQR_STEP(27); LSQR_STEP(28); LSQR_STEP(29); LSQR_STEP(30);
  LSQR_STEP(31);

  return q1;
}
```

Typical examples of the square root computation for fractional radicands are shown in the following table:

Table 6. **Examples of 32-bit square root**

Format	X		Square Root, Z=SQRT(X)	
	Signed Fractional	Hexadecimal	Signed Fractional	Hexadecimal
Q0.31	0.5	0x40000000	0.7071068	0x5A827999
	0.25	0x20000000	0.5000000	0x40000000
	0.125	0x10000000	0.3535534	0x2D413CCC

The digital filter theory and derivation formulas for calculation of the filter coefficients are discussed in subsequent section.

3.2. Infinite impulse response filter

Infinite Impulse Response (IIR) digital filters have a transfer function of the form:

$$H(z) = \frac{a_0 + a_1 z^{-1} + \cdots + a_M z^{-M}}{1 + b_1 z^{-1} + \cdots + b_N z^{-N}} \qquad \textit{Eq. 6}$$

where, H(z) is the z-Transform, N is filter order and M ≤ N.

The most common technique used for designing IIR digital filters involves the following steps:

1. The designing of an analog prototype filter.
2. Transforming the prototype to the digital representation.

The most common designs for the analog filter are Butterworth, Chebyshev, and Elliptic. The Chebyshev and Elliptic filters are characterized by more rapid transitions from pass-band to stop-band than the Butterworth filter.

In a metering application, the monotonic and smooth overall filter response is preferred so as not to distort the magnitudes of the phase voltage and current harmonics in the band of interest.

In addition, neither an attenuation slope nor a sharp transition from the pass-band to the stop-band is critical. Obviously, moderate attenuation and transition band of the filter will cause slight magnitude error at frequency of the mains 50 Hz (60 Hz). This filter error along with other inaccuracies of the power meter's measurement and calculation chain are calibrated on the production line. Due to relaxed requirements on a steep attenuation slope but the necessity of a smooth overall filter response, both the low-pass and high-pass first order digital filters were derived from the transfer function $H(s)$ of the normalized first order Butterworth analog filter.

$$\boldsymbol{H(s) = \frac{1}{s+1}} \qquad \textit{Eq. 7}$$

The normalized transfer function $H(s)$ represents the case for the cut-off frequency $\omega_C = 1\ [\mathrm{rad/s}]$. To obtain Butterworth filters with different cut-off frequencies, it is convenient to use the normalized transfer function $H(s)$ as prototype and apply the analog-to-analog transformations $s \rightarrow s/\omega_C$. By applying this transformation, we get the transfer function for the low-pass first order Butterworth filter.

$$\boldsymbol{H_{LP}(s) = \frac{\omega_c}{s+\omega_c}} \qquad \textit{Eq. 8}$$

where, ω_c is the low-pass filter cut-off frequency in $[\mathrm{rad/s}]$.

The following figure shows magnitude and phase responses of the low-pass first order Butterworth filter for $\omega_c = 1\ [\mathrm{rad/s}]$. In electronics, the frequency responses are often described in terms of "per decade". The example Bode plot shows a slope of -20 dB/decade in the stop-band, which means that for every factor-of-ten increase in frequency (going from 10 rad/s to 100 rad/s in the figure), the gain decreases by 20 dB.

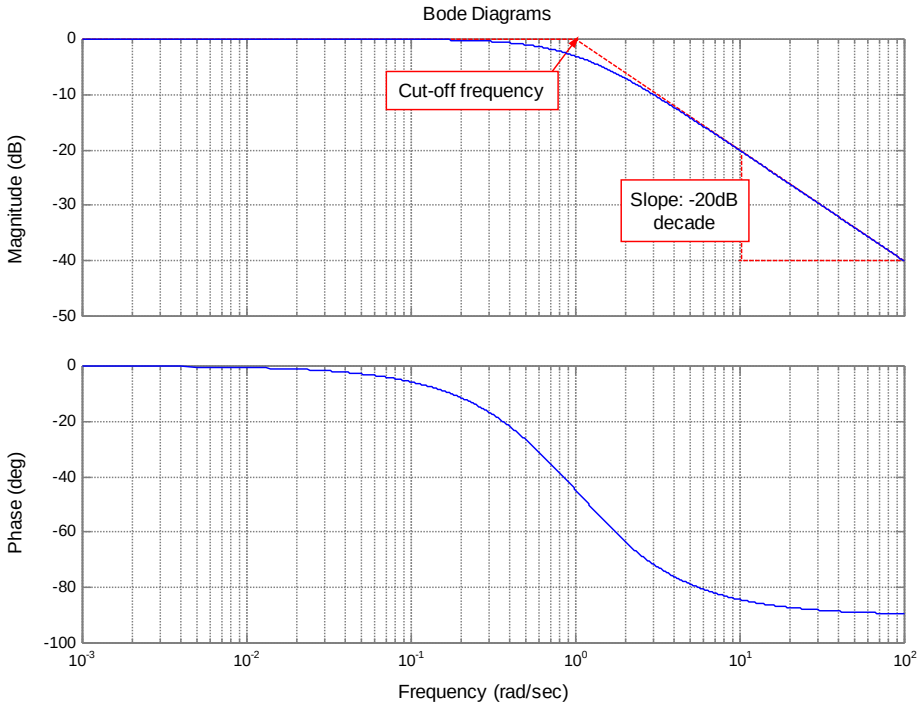


Figure 4. **Bode diagram of the low-pass first order Butterworth filter**

The low-pass filter step response $c_{LP}(s)$ to unit step function $L\{u(t)\} = 1/s$ is as follows:

$$\boldsymbol{C_{LP}(s) = \frac{1}{s}\frac{\omega_c}{s+\omega_c}} \qquad \textit{Eq. 9}$$

Taking the Inverse Laplace transform, the step response is given by:

$$\boldsymbol{c_{LP}(t) = 1 - e^{-\omega_c t}} \qquad \textit{Eq. 10}$$

The filter settling time is defined as the time of the step response to reach, and stay within, 2% of its final value 1.0. Thus, solving Eq. 10 for the time parameter the low-pass filter settling time t is expressed as follows:

$$\boldsymbol{t = \left[\frac{-2log\left(\sqrt{1-c_{LP}(t)}\right)}{\omega_c}\right]_{c_{LP}(t)=0.98}} \qquad \textit{Eq. 11}$$

Similarly, to the previously derived low-pass filter, the high-pass first order Butterworth filter can be derived by applying the analog-to-analog transformation $s \rightarrow \omega_C/s$.

$$\boldsymbol{H_{HP}(s) = \frac{s}{\omega_c + s}} \qquad \textit{Eq. 12}$$

where, ω_c is the high-pass filter cut-off frequency in $[rad/s]$.

The following figure shows magnitude and phase responses of the high-pass first order Butterworth filter for $\omega_c = 1$ $[rad/s]$. The example Bode plot shows a slope of -20 dB/decade in the stop-band, which means that for every factor-of-ten decrease in frequency (going from 0.1 rad/s to 0.01 rad/s in the figure), the gain decreases by 20 dB.

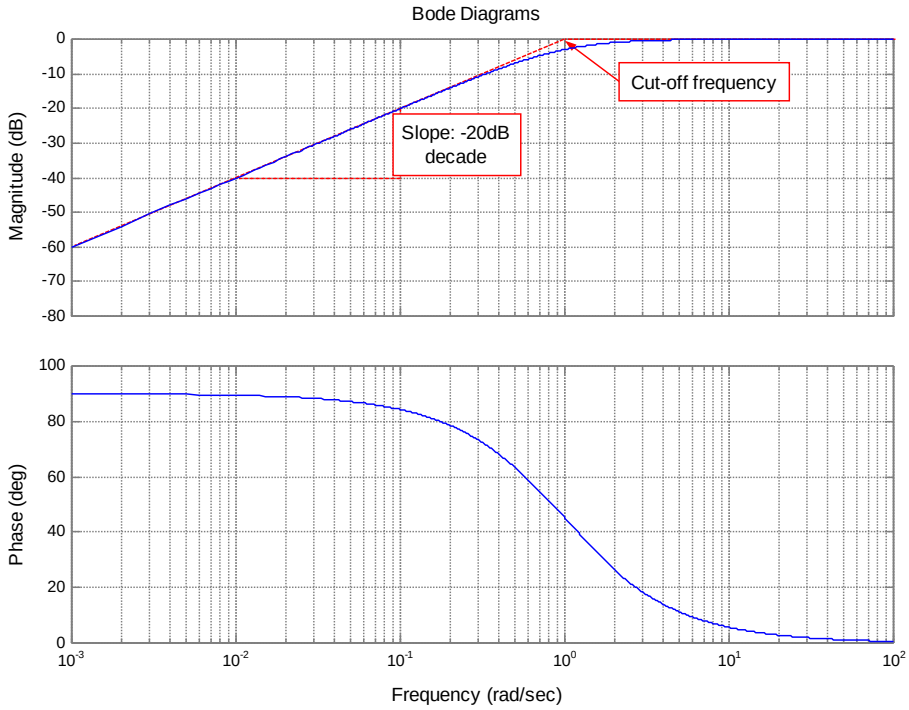


Figure 5. **Bode diagram of the high-pass first order Butterworth filter**

Also, the settling time of the high-pass filter is defined as the time of the response to reach, and stay within, 2% of its final value 0.0. By Inverse Laplace transform of the high-pass filter transfer function Eq. 12, combined with the unit step function $L\{u(t)\} = 1/s$ and solving the equation for the time parameter, the settling time is given by

$$t = \left[\frac{-log(c_{HP}(t))}{\omega_c}\right]_{c_{HP}(t)=0.02} \qquad \textit{Eq. 13}$$

The magnitude responses of the high-pass and low-pass Butterworth filters are monotonic overall with magnitude $|H_{LP}(j\omega)| = |H_{HP}(j\omega)| = 1/\sqrt{2}$ (magnitude down by 3 dB) at $\omega_c = 1$.

Further in this section, the digital representation of analog filters will be derived. The analog filter prototypes given by Eq. 8 and Eq. 12 must be transformed into a digital representation using analog-to-digital mapping. This generally involves a transformation between the s-plane and the z-plane mapping. Several transformations exist, see [1].

This section outlines the bilinear transformation that transforms $H(s)$ into $H(z)$ via the relation:

$$H(z) = [H(s)]_{s=(2/T)(1-z^{-1})/(1+z^{-1})} \qquad \textit{Eq. 14}$$

where, T is the sampling period in seconds.

The next step is to obtain the discrete transfer function of the low-pass first order Butterworth filter by applying a bilinear transformation to the low-pass analog filter transfer function Eq. 8.

$$H(z) = \frac{\left(\frac{\omega_C T}{2+\omega_C}\right)z + \left(\frac{\omega_C T}{2+\omega_C T}\right)}{z - \left(\frac{2-\omega_C T}{2+\omega_C T}\right)} \qquad \textit{Eq. 15}$$

In order to match the magnitude responses of the digital filter transfer function Eq. 15 and the analog filter prototype transfer function Eq. 8, the cut-off frequency of the analog filter ω_C must be shifted relative to the digital filter cut-off frequency ω_D [6].

$$\omega_C = \frac{2}{T}\tan\left(\frac{\omega_D T}{2}\right) \quad \textit{Eq. 16}$$

The difference equation of the first order filter expressed in a general form is:

$$y(n) = b_1 x(n) + b_2 x(n-1) - a_2 y(n-1) \quad \textit{Eq. 17}$$

Substituting the frequency pre-warping Eq. 16 into Eq. 15, and by further applying the Inverse z-transform, the coefficients of the difference equation representing the low-pass first order Butterworth digital filter can be calculated:

$$b_1 = 2\tan(\omega_D T/2)/[2 + 2\tan(\omega_D T/2)] \quad \textit{Eq. 18}$$

$$b_2 = 2\tan(\omega_D T/2)/[2 + 2\tan(\omega_D T/2)] \quad \textit{Eq. 19}$$

$$a_2 = [2\tan(\omega_D T/2) - 2]/[2 + 2\tan(\omega_D T/2)] \quad \textit{Eq. 20}$$

where, $\omega_D = 2\pi f_D$ is the cut-off frequency of the digital filter in $[\mathrm{rad/s}]$, and T is the sampling period in seconds.

Similarly, by substitution of the bilinear transformation into the high-pass analog filter transfer function (Eq. 12), applying frequency prewarping and the Inverse z-transform, the coefficients of the difference equation for the high-pass first order Butterworth digital filter can be derived:

$$b_1 = 2/[2 + 2\tan(\omega_D T/2)] \quad \textit{Eq. 21}$$

$$b_2 = -2/[2 + 2\tan(\omega_D T/2)] \quad \textit{Eq. 22}$$

$$a_2 = [2\tan(\omega_D T/2) - 2]/[2 + 2\tan(\omega_D T/2)] \quad \textit{Eq. 23}$$

3.3. Explicit RMS converter

The Root Mean Square (RMS) is a fundamental measurement of the magnitude of an alternating signal. In mathematics, the RMS is known as the standard deviation, which is a statistical measure of the magnitude of a varying quantity. It measures only the alternating portion of the signal as opposed to the RMS value, which measures both the direct and alternating components. In electrical engineering, the RMS or effective value of a current (IRMS) is, by definition, such that the heating effect is the same for equal values of alternating or direct current. The basic equation for straightforward computation of the RMS current from the signal function is:

$$IRMS = \sqrt{\frac{1}{T}\int_0^T [i(t)]^2\, dt} \quad \textit{Eq. 24}$$

where, $i(t)$ denotes the function of the analyzed waveform in the time domain, and the period T is the time it takes for one complete signal cycle to be produced.

The proposed solution for RMS current calculation overcomes the inherent limitation of the straightforward computation Eq. 24, such as the need for determining precisely the limits for the finite integration. It is known in technical literature as an explicit RMS converter, and has been used for many years primarily for monolithic RMS/DC converters [7] and [8].

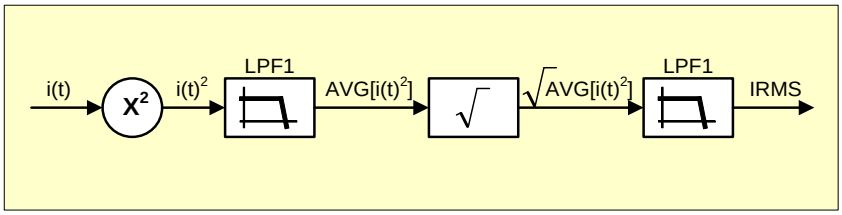


Figure 6. **Explicit RMS current converter**

This method for computing the RMS value requires numerical square, average and square root functions to be called every time a new sample of the analyzed signal is obtained. Figure 6 shows the explicit RMS converter implementation for RMS current computation.

The Filter-Based Metering Algorithm uses the explicit RMS converter method for calculating the RMS current (IRMS) and RMS voltage (URMS). The next section describes a similar method for the calculation of active and reactive power.

3.4. Average power converter

As opposed to the RMS current, where the heating effect is the same for equal values of alternating or direct current, the RMS value of power is not equivalent to heating power and, in fact, it does not represent any useful physical quantity. The equivalent heating power of a waveform is the average power and can be calculated using the average power converter. This converter can calculate both the active (P) and reactive (Q) powers.

The active power (P) is measured in watts (W) and is expressed as the product of the voltage and the in-phase component of the alternating current. In fact, the average power of any whole number of cycles is the same as the average power value of just one cycle. So, we can easily find the average power of a very long-duration periodic waveform simply by calculating the average value of one complete cycle.

$$P = \frac{1}{T}\int_0^T u(t)\, i(t) dt \qquad \textit{Eq. 25}$$

where, u(t) and i(t) denote alternating voltage and current waveforms, and the time T is the waveform period.

The average power converter is, to some extent, similar to the explicit RMS converter. The power is calculated by multiplying instantaneous voltage and current samples and passing the product through a two-stage low-pass first order Butterworth filter as shown in the following figure:

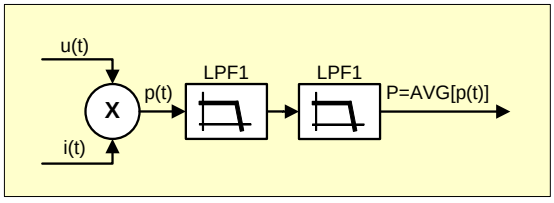


Figure 7. **Average active power converter**

The Filter-Based Metering Algorithm uses the average power converter for calculation of active power (P) and reactive power (Q). The reactive power (Q) is measured in units of volt-amperes-reactive (VAR) and is the product of the voltage and current and the sine of the phase angle between them. The reactive power (Q) is calculated in the same manner as active power (P), but in reactive power the voltage input waveform is 90 degrees shifted with respect to the current input waveform.

The Hilbert filter, a special FIR filter for shifting a phase voltage waveform by 90 degrees, is explained in the following section.

3.5. Ideal Hilbert transformer

The Ideal Hilbert transformer is a special class of transformation which is characterized by phase shifting all the pass-band frequencies of the input signal by 90 degrees.

$$H(e^{j\omega}) = \begin{cases} -j, & 0 < \omega < \pi \\ j, & -\pi > \omega > 0 \end{cases} \qquad \textit{Eq. 26}$$

The following figure shows the magnitude and phase response of the Ideal Hilbert transformer. The frequency response of the ideal analog Hilbert transformer has unity magnitude, a phase angle of $-\pi/2$ for $0 < \omega < \pi$ and a phase angle $+\pi/2$ for $-\pi < \omega < 0$.

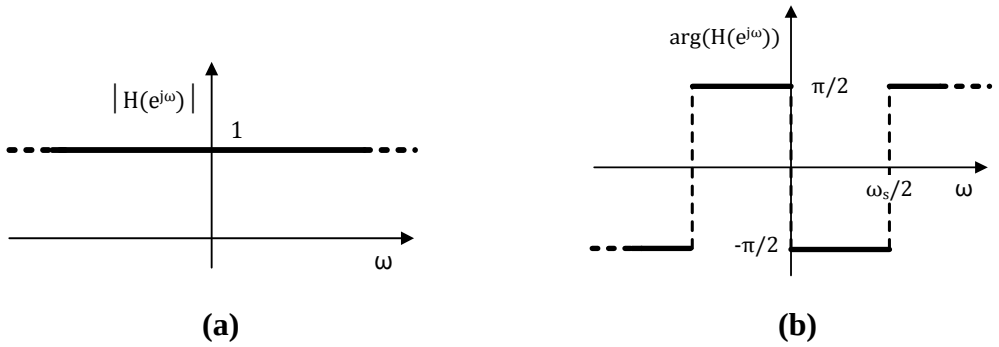


Figure 8. **Magnitude and phase responses of the ideal Hilbert transformer**

The impulse response h(n) of the Ideal Hilbert transformer is [3] and [9].

$$h[n] = \begin{cases} \frac{2}{\pi}\frac{\sin^2(\pi n/2)}{n}, & n \neq 0 \\ 0, & n = 0 \end{cases} \qquad \textit{Eq. 27}$$

The impulse response infinitely extends in both directions (see Figure 9). Moreover, the output of the Ideal Hilbert transformer starts responding to the Dirac Impulse in advance. The infinite length and predictive nature of the impulse response mean that the ideal Hilbert transformation cannot be implemented in practice - an approximation is therefore necessary.

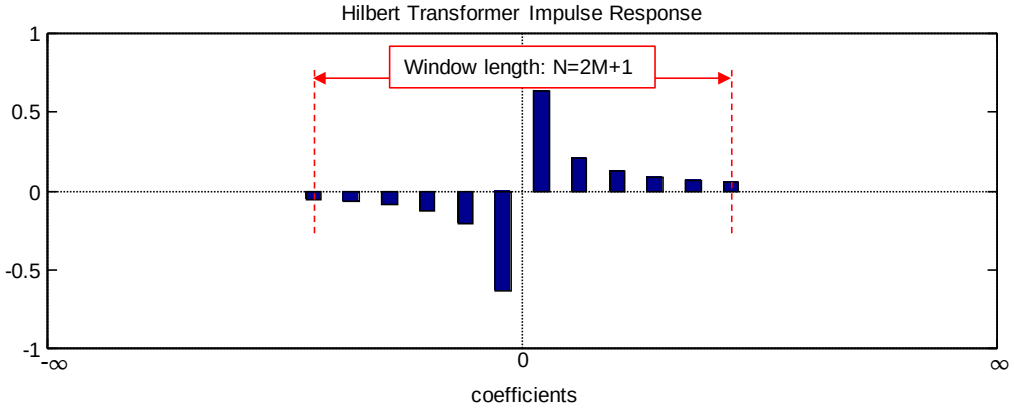


Figure 9. **Ideal Hilbert transformer impulse response**

The Filter-Based Metering Algorithm uses FIR approximation and Kaiser Window to restrict the impulse response length of the Ideal Hilbert transformer. This procedure can be compared to placing a window of width $N = 2M + 1$ over all of the coefficients. All the coefficients within the window are retained and multiplied with the window weight coefficient, and all coefficients outside the window are discarded.

The Kaiser Window coefficients of the Hilbert FIR filter of length N are expressed by equation:

$$w[n] = \begin{cases} \dfrac{I_0\left\{\beta\sqrt{(1-[(n-n_d)/n_d]^2)}\right\}}{I_0\{\beta\}}, & 0 \le n \le N-1 \\ 0, & otherwise \end{cases} \qquad \textit{Eq. 28}$$

where, $n_d = M/2$, I_0 is the zeroth order Modified Bessel function of the first kind, β is an arbitrary real number that determines the shape of the Kaiser Window, and $N = 2M + 1$ is the length of the Hilbert FIR filter.

Furthermore, the impulse response is shifted by a constant group delay to make the system casual. The Hilbert FIR filter, the closest approximation of the Ideal Hilbert transformer, with a finite number of coefficients shifted by constant group delay M, is shown in the following figure:

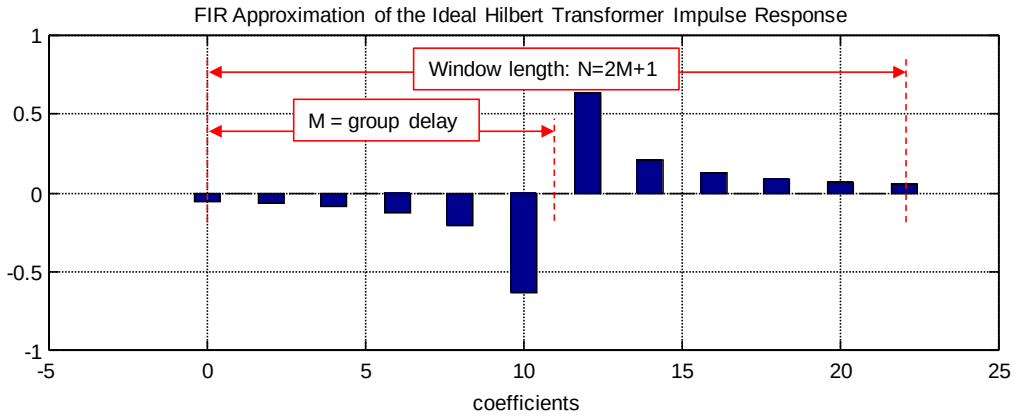


Figure 10. **FIR approximation of the ideal Hilbert transformer impulse response**

Typically, FIR filters are implemented as causal filters, so the actual phase response of the Hilbert FIR filter will be the approximate Hilbert phase response plus a linear phase term with a slope equal to M considering filter length $N = 2M + 1$.

Therefore, when a signal passes through such a Hilbert FIR filter, the output is not the Hilbert transform of the input, but rather it is the Hilbert transform of the input delayed by M samples. If the filter length is even, this will yield a non-integer sample delay. Thus, an odd-valued filter length is usually desirable so that the input signal x[n] can be passed both through the Hilbert FIR filter and through an integer sample delay to yield two signals $y_{90}[n]$ and $y_{del}[n]$ that are related through the Hilbert transform as shown in the following figure:

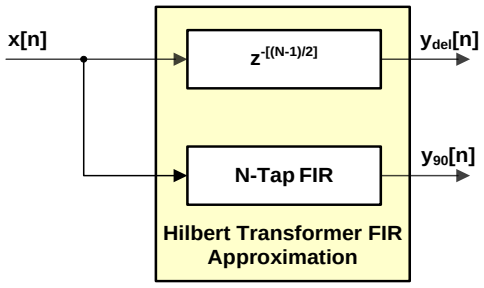


Figure 11. **FIR approximation block of the ideal Hilbert transformer**

The magnitude and phase response of the FIR Hilbert filter designed using a Kaiser Window (N=23 and β =0, 4 and 8) is shown in the following figure.

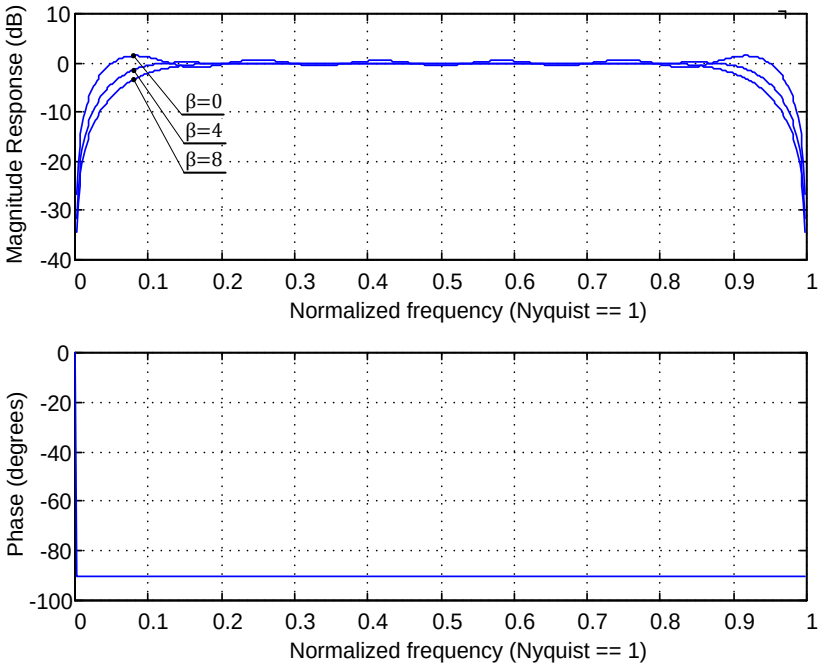


Figure 12. **FIR approximation block magnitude and phase response**

NOTE

The case with β = 0 corresponds to use of the Rectangular Window with all the weights within the window set to one and the remaining weights outside the window set to zero.

As already indicated, the Hilbert FIR filter is used by the Filter-Based Metering Algorithm to phase shift the voltage input waveform by 90 degrees with respect to the current input waveform. The shifted waveforms are then used for calculating the reactive power (Q) and reactive energy (kVARh).

3.6. Rogowski coil sensor signal processing

Rogowski coils, typically represented by an air core coil, provide linear measurement within a high current dynamic range. The voltage that is induced in the Rogowski coil is proportional to the rate of change (derivative) of the measured current. Because the output from the Rogowski coil is a derivative of current di/dt, an integrator is needed to convert it back to the original format i(t) [10].

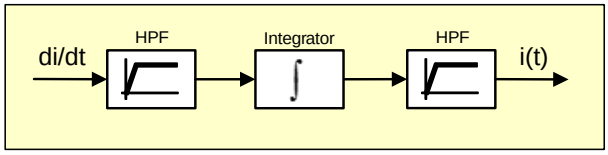


Figure 13. **Rogowski coil digital integrator**

Figure 13 shows the calculation path of the digital integrator implemented by the Filter-Based Metering Algorithm. The calculation path comprises an integrator block and two high-pass first order IIR filter blocks. The first high-pass filter in the computation chain is required to prevent the periodic overflows of the integrator which would otherwise occur due to DC offset of the input signal.

As already indicated, the integrator block converts a derivative of the current back to the original format. In the frequency domain, an output of the integrator block can be viewed as a -20dB/decade attenuation and a constant –90° phase shift (see Figure 14).

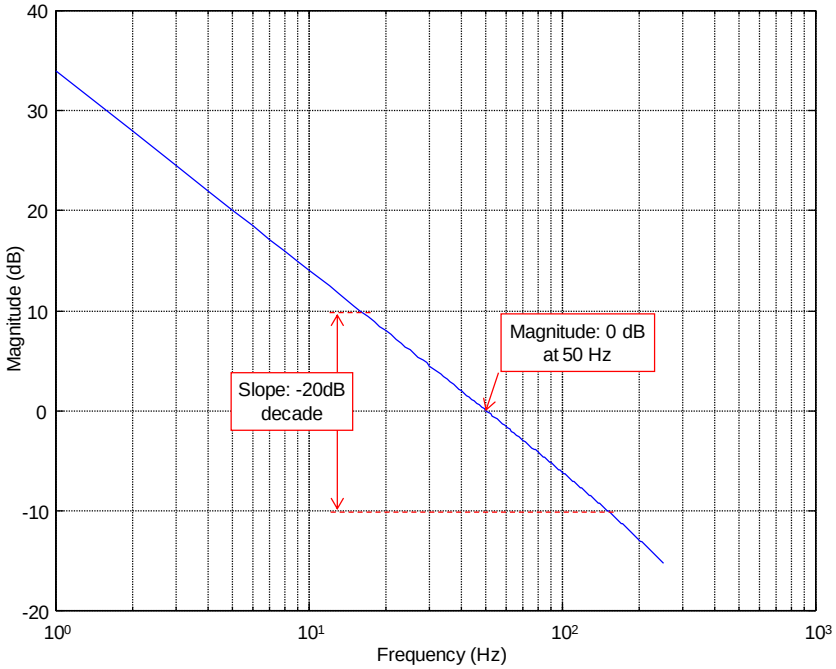


Figure 14. **Magnitude response of the integrator block from 1 to 250 Hz with gain 0 dB at 50 Hz**

The second high-pass filter is required to remove DC offset from the output of the integrator block.

When both high-pass filters are used, then the output signal of the digital integrator is proportional to the input current, even if Rogowski coil outputs are distorted by DC offset or by slowly varying signals. The only difference between the measured current and the Rogowski coil output voltage processed by the digital integrator is in the small phase shift. The small phase shift error is caused by the high-pass filters in the calculation path together with the integrator block, and may be corrected by propagating the phase voltage samples through the same high-pass filters.

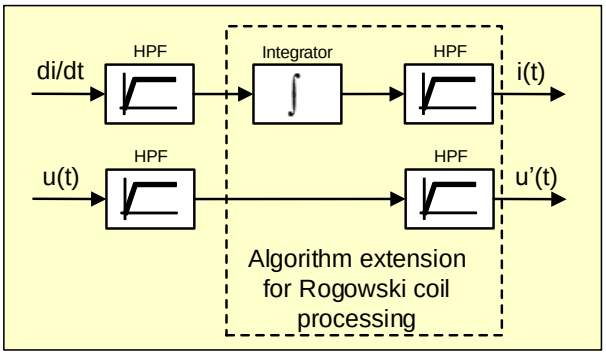


Figure 15. **Offset removal block combined with a Rogowski coil digital integrator**

Figure 15 shows an offset removal block combined with a Rogowski coil digital integrator. This block removes DC offset from the input signal samples and converts the rate of change of the current, measured by the Rogowski coil sensor, into the original format.

NOTE

Even if other current sensor types, such as a current transformer or shunt resistor are used, it is always recommended to eliminate DC offset and slowly varying signals before energy computing. In such cases, neither the integrator block nor the pair of high-pass IIR filters is required, and thus the relatively complex block, for Rogowski coil processing, transforms into a high-pass first order IIR filter in each signal path.

4. Power meter application development

Mastering a power meter application and achieving the accuracy classes with minimal computational resources and a low-power budget might be a never-ending process. More than designer diligence, usually it's the time to market that drives power meter development milestones. Specifically, the metrology portion of the power meter must be robust and behave deterministically under all conditions. Therefore, in order to accelerate power meter development, the designers may familiarize themselves with algorithms offered by the semiconductor vendors and select and adopt the best solution.

Besides the Filter-Based Metering Algorithm theory, this application note also describes the software functions which serve as an interface into the algorithm and its capabilities. All software functions are built into the metering library that must be integrated within the firmware application during project compilation and linking. These software functions shall be called preferably at fixed sampling intervals. In fact, existing implementation allows the use of two sampling intervals. Introducing short and long sampling intervals for calculating billing and non-billing quantities, respectively, will lead to significant savings in the computational power. Recalculating all non-billing quantities at a lower update rate is technically acceptable and highly recommended.

NOTE

The application note is delivered together with the metering library and test applications. The library is provided in object format and the test applications in C-source code.

The general setting of the algorithm can be easily performed by the configuration tool. This tool runs on a personal computer and it allows the user to tune algorithm behavior in an interactive way and matching the required performance. The configuration session completes by generating a C-header file with algorithm configuration data specific to the selected power meter topology.

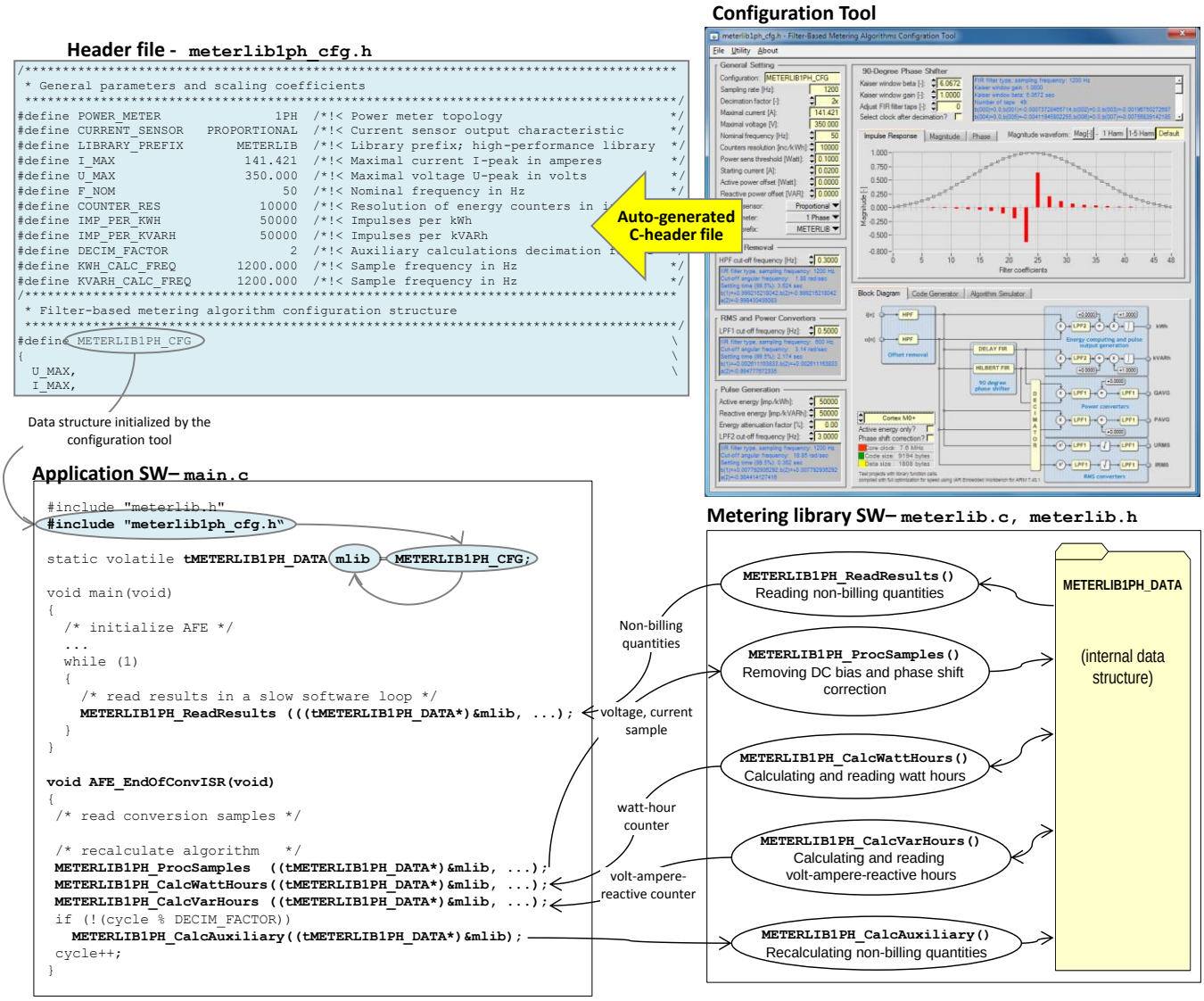


Figure 16. **Power meter development and user interactions**

The software needed to perform basic metering functionality can be divided into two parts:

- **Application software** – this part includes configuration of the on-chip peripheral modules for high-precision analog measurement and low jitter pulse output generation, reading phase voltage and current samples and passing them to the metering library functions.

- **Metering library** – comprises a set of highly optimized functions for calculating the billing and non-billing quantities from the measured phase voltage and current samples. The behavior of the Filter-Based Metering Algorithm is configured with the help of configuration tool.

Figure 16 depicts usage of the metering library and configuration tool in a simple one-phase power meter test application.

Initially, necessary hardware initialization, including the AFE, is performed in the main() function.

Consecutively, all processing takes place in the AFE_ EndOfConvISR() interrupt service routine (ISR). In this routine, the phase voltage and phase current samples are read from AFE and passed to the metering algorithm via the METERLIB1PH_ProcSamples() function.

The following two functions, METERLIB1PH_CalcWattHours() and METERLIB1PH_CalcVarHours(), can be called whenever new conversion samples are available. Practically, these functions shall be called at least 1200 times per second in order to calculate active and reactive energies in the frequency bandwidth up to 10th harmonic. The increasing calling frequency of these functions makes sense only if the billing quantities need to be calculated over a higher frequency bandwidth. In a standard power meter application, the frequency bandwidth of calculations up to 10th harmonics is usually sufficient and a further increasing sampling rate will not bring any advantage.

The additional function METERLIB1PH_CalcAuxiliary() is called at a lower update rate and it recalculates all non-billing quantities. The calling frequency for this particular function is even less demanding than for calculating billing quantities.

Finally, the information stored within the metering library's internal data structure can be read by the METERLIB1PH_ReadResults() function. This function is usually called from the main() function or from a low-frequent software task. The typical calling frequency is in the range from 100 to 250 milliseconds depending on the update rate of non-billing quantities on the LCD.

Figure 16 shows that the metering library operates almost independently, it only requires that conversion samples of the phase voltage and phase current waveforms be provided by the user application. Due to this design methodology, the library can be very easily incorporated into various power meter applications. Further advantages come along with the configuration tool. This tool allows metering algorithms to be set up and filters tuned in an interactive way. The configuration shall be stored in the C-header file (for example, meterlib1ph_cfg.h) which is included in the compilation process of the application and defines algorithm behavior.

The metering library and configuration tool support one-phase, two-phase (Form-12S) and three-phase power meter applications. These deliverables are discussed in the following sections.

4.1. Metering libraries

This section describes the metering library's implementation of the Filter-Based Metering Algorithm. The library comprises several functions with a unique Application Programming Interface (API) for the most frequent power meter topologies; that is, one-phase, two-phase (Form-12S), and three-phase. More precisely, two function sets are available. The first function set is optimized to compute metering quantities at high precision, generating a significant computational load. The second function set has been designed to support low-power applications. It computes metering quantities at a moderate to low precision, but generates only 35% of the computation load required by the high-precision functions. Both the high-precision and low-power function sets are accessible from the meterlib.lib and

meterliblp.lib library files, respectively. Similarly to library files, the function prototypes and internal data structures of both function sets are also declared in the meterlib.h and meterliblp.h header files.

NOTE

The IAR Embedded Workbench for ARM® (version 7.40.1) tool was used to obtain performance data for all library functions. The code was compiled with full optimization for execution speed for the MKM34Z128, an ARM® Cortex®-M0+ core based target [11]. The device was clocked at 48 MHz using the Frequency-Locked Loop (FLL) module operating in FLL Engaged External (FEE) mode, driven by an external 32.768 kHz crystal. Measured execution times were recalculated to core clock cycles. The flash and RAM requirements are represented in bytes.

The simple block diagrams of the computing process, split by the functions realized by the high-precision and low-power libraries in a typical one-phase power meter application, are depicted in the following figures.

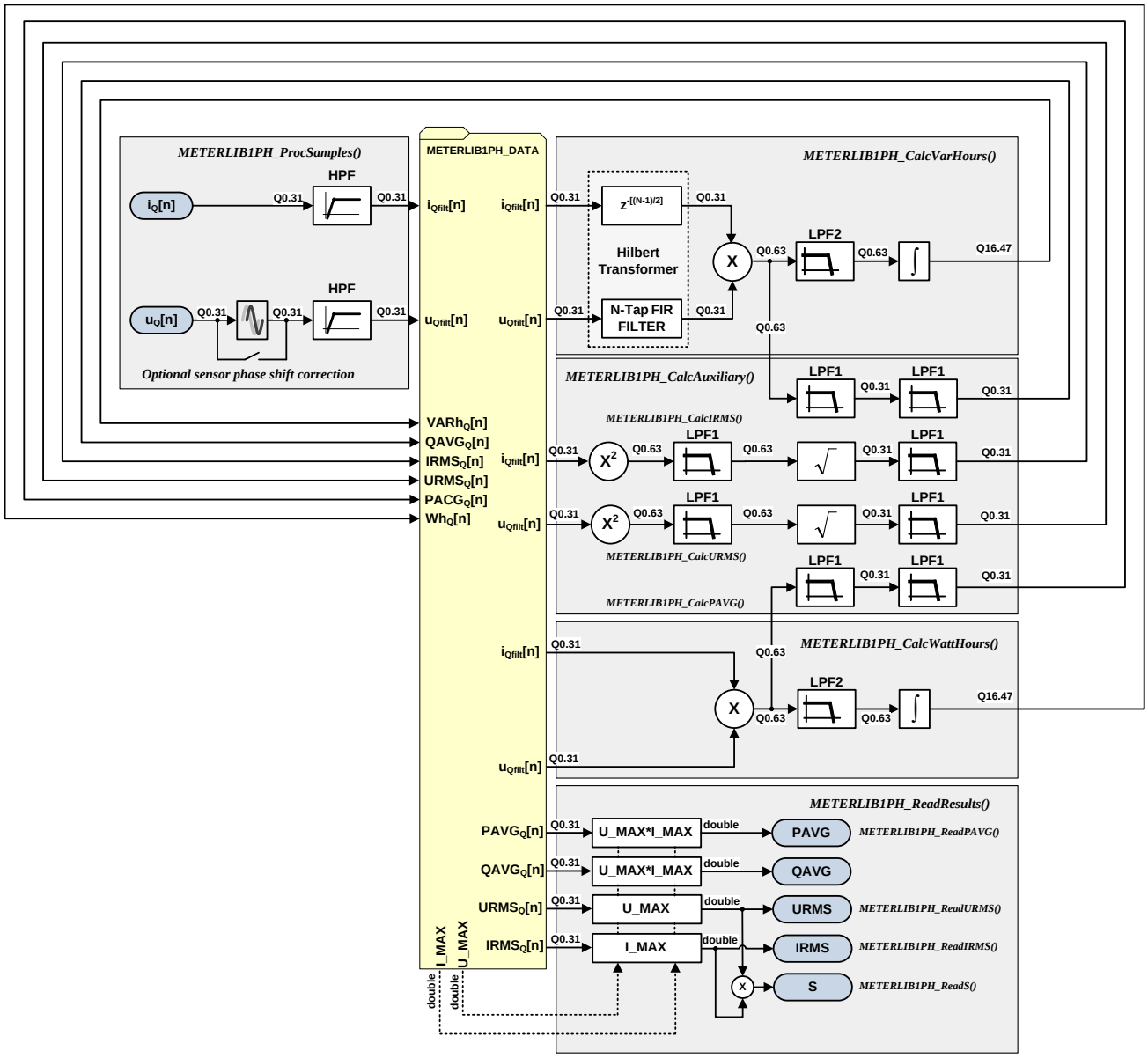


Figure 17. **Block diagram of the one-phase power meter computing using the high-precision library**

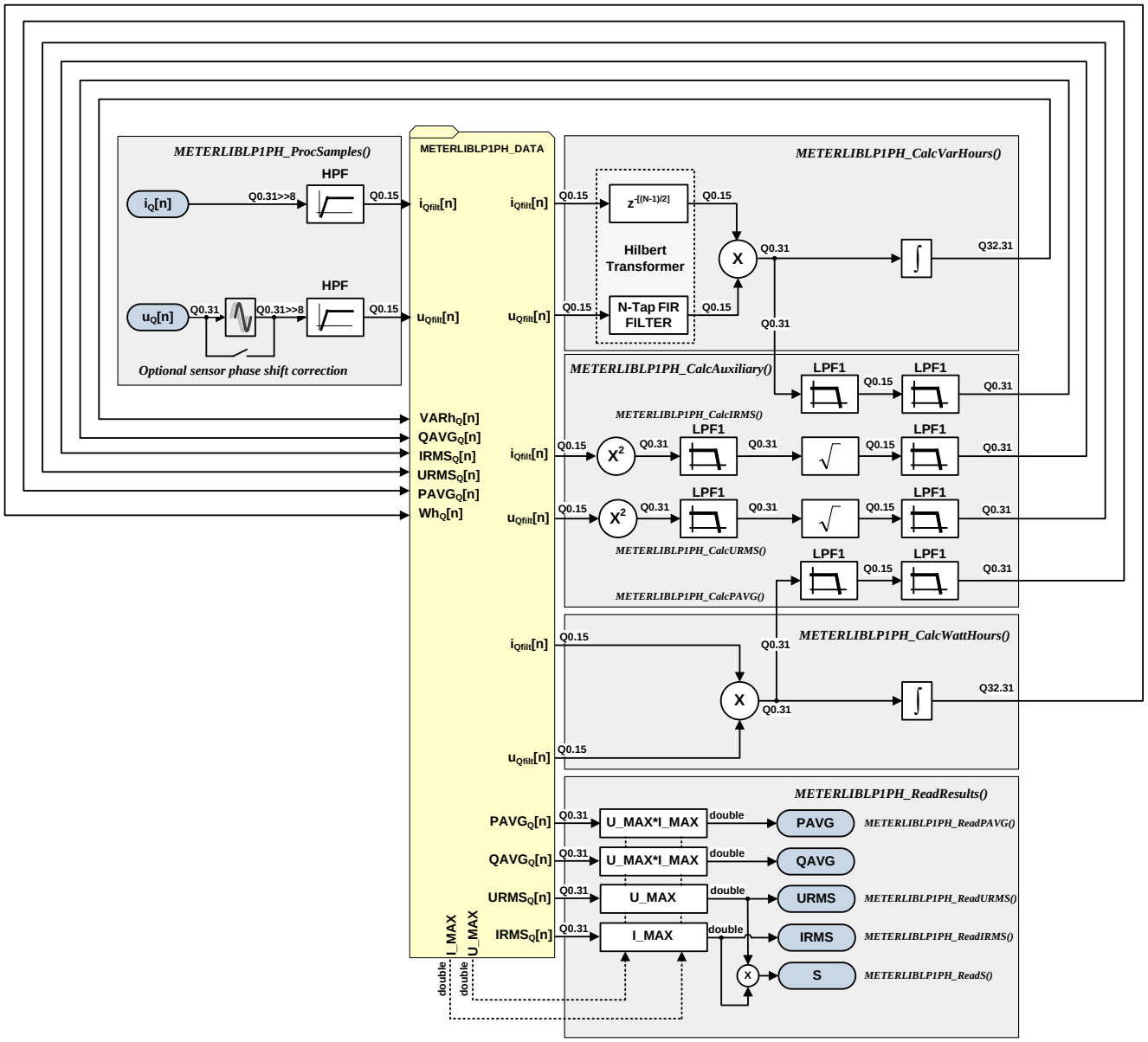


Figure 18. **Block diagram of the one-phase power meter computing using the low-power library**

The low-power library functions are implemented using 16-bit fractional math to the contrary of the high-precision library, which performs most of computation using either 32-bit or 64-bit fractional math. Additional performance savings of the low-power library were achieved by not computing the energy smoothing "LPF2" low-pass filter. This filter helps to speed up power meter calibration by eliminating energy ripples at twice the load frequency, developed by the multiplication of two 50/60 Hz sinusoidal waveforms. If this filter is not computed then energy accuracy during power meter calibration and the testing phases must be measured by accumulating the energy output pulses in a 5 to 10 second window.

A detailed description of the libraries' exported data types, their functions and APIs, is given in the following subsections. The "METERLIB" or "METERLIBLP" prefix to a function name indicates membership of the function to either the high-precision library file meterlib.lib or the low-power library file meterliblp.lib.

NOTE

Use exclusively high-precision or low-power library functions in your application. An attempt to call a low-power library function with a high-precision library internal data structure argument, or vice versa, is not allowed and will terminate by an error in the project compilation phase.

4.1.1. Core architecture and compiler support

The high-precision and low-power libraries support ARM® Cortex®-M0+ and Cortex-M4 cores. In addition to standard cores, the libraries also support the Memory-Mapped Arithmetic Unit (MMAU), a hardware math module designed by Freescale to accelerate the execution of specific metering algorithms.

The default installation folder of the filter-based metering libraries and the graphical configuration tool is C:\Freescale\METERLIB_R4_1_0.

The following table lists all the necessary header files, library files, and their locations, relative to the default installation folder. Add these files and paths into your project workspace to successfully integrate the high-precision metering library into your application.

Table 7. **High-precision library integration**

Include files and libraries			METERLIB		
			Cortex-M0+ w/o MMAU	Cortex-M0+ w/ MMAU	Cortex-M4
include	files	iar armcc gcc	fraclib.h meterlib.h		
	paths	iar	..\lib\fraclib\inc ..\lib\fraclib\inc\cm0p ..\lib\meterlib\inc	..\lib\fraclib\inc ..\lib\fraclib\inc\cm0p_mmau ..\lib\fraclib\inc\cm0p_mmau\iar ..\lib\meterlib\inc	..\lib\fraclib\inc ..\lib\fraclib\inc\cm4 ..\lib\meterlib\inc
		armcc		..\lib\fraclib\inc ..\lib\fraclib\inc\cm0p_mmau ..\lib\fraclib\inc\cm0p_mmau\armcc ..\lib\meterlib\inc	
		gcc		..\lib\fraclib\inc ..\lib\fraclib\inc\cm0p_mmau ..\lib\fraclib\inc\cm0p_mmau\gcc ..\lib\meterlib\inc	
library	files	iar	fraclib_cm0p_iar.a meterlib_cm0p_iar.a	fraclib_cm0p_mmau_iar.a meterlib_cm0p_mmau_iar.a	fraclib_cm4_iar.a meterlib_cm4_iar.a
		armcc	fraclib_cm0p_armcc.lib meterlib_cm0p_armcc.lib	fraclib_cm0p_mmau_armcc.lib meterlib_cm0p_mmau_armcc.lib	fraclib_cm4_armcc.lib meterlib_cm4_armcc.lib
		gcc	fraclib_cm0p_gcc.a meterlib_cm0p_gcc.a	fraclib_cm0p_mmau_gcc.a meterlib_cm0p_mmau_gcc.a	fraclib_cm4_gcc.a meterlib_cm4_gcc.a
	paths	iar armcc gcc	..\lib\fraclib ..\lib\meterlib		

The following table lists all the necessary header and library files together with their relative paths to add into your project workspace to successfully integrate the low-power metering library into your application.

Table 8. **Low-power library integration**

Include files and libraries			METERLIBLP		
			Cortex-M0+ w/o MMAU	Cortex-M0+ w/ MMAU	Cortex-M4
include	files	iar	fraclib.h meterliblp.h		
		armcc			
		gcc			
	paths	iar	..\lib\fraclib\inc ..\lib\fraclib\inc\cm0p ..\lib\meterliblp\inc	..\lib\fraclib\inc ..\lib\fraclib\inc\cm0p_mmau ..\lib\fraclib\inc\cm0p_mmau\iar ..\lib\meterliblp\inc	..\lib\fraclib\inc ..\lib\fraclib\inc\cm4 ..\lib\meterliblp\inc
		armcc		..\lib\fraclib\inc ..\lib\fraclib\inc\cm0p_mmau ..\lib\fraclib\inc\cm0p_mmau\armcc ..\lib\meterliblp\inc	
		gcc		..\lib\fraclib\inc ..\lib\fraclib\inc\cm0p_mmau ..\lib\fraclib\inc\cm0p_mmau\gcc ..\lib\meterliblp\inc	
library	files	iar	fraclib_cm0p_iar.a meterliblp_cm0p_iar.a	fraclib_cm0p_mmau_iar.a meterliblp_cm0p_mmau_iar.a	fraclib_cm4_iar.a meterliblp_cm4_iar.a
		armcc	fraclib_cm0p_armcc.lib meterliblp_cm0p_armcc.lib	fraclib_cm0p_mmau_armcc.lib meterliblp_cm0p_mmau_armcc.lib	fraclib_cm4_armcc.lib meterliblp_cm4_armcc.lib
		gcc	fraclib_cm0p_gcc.a meterliblp_cm0p_gcc.a	fraclib_cm0p_mmau_gcc.a meterliblp_cm0p_mmau_gcc.a	fraclib_cm4_gcc.a meterliblp_cm4_gcc.a
	paths	iar	..\lib\fraclib ..\lib\meterliblp		
		armcc			
		gcc			

4.1.2. High-precision library function API

This section summarizes the functions' API defined in the high-precision metering library meterlib.lib. Prototypes of all functions and internal data structures are declared in the meterlib.h header file.

4.1.2.1. One-Phase power meter

- void **METERLIB1PH_ProcSamples** (tMETERLIB1PH_DATA *p, frac24 u1Q, frac24 i1Q, frac16 *shift);

  Remove DC bias from phase voltage and phase current samples, together with performing an optional sensor phase shift correction.
- void **METERLIB1PH_CalcWattHours** (tMETERLIB1PH_DATA *p, tENERGY_CNT *pCnt, frac64 puRes);

  Recalculate active energy using new voltage and current samples.
- void **METERLIB1PH_CalcVarHours** (tMETERLIB1PH_DATA *p, tENERGY_CNT *pCnt, frac64 puRes);
  Recalculate reactive energy.

- void **METERLIB1PH_CalcAuxiliary** (tMETERLIB1PH_DATA *p);
  Recalculate auxiliary variables; IRMS, URMS, P, Q, and S.
- void **METERLIB1PH_CalcURMS** (tMETERLIB1PH_DATA *p);
  Recalculate URMS.
- void **METERLIB1PH_CalcIRMS** (tMETERLIB1PH_DATA *p);
  Recalculate IRMS.
- void **METERLIB1PH_CalcPAVG** (tMETERLIB1PH_DATA *p);
  Recalculate PAVG.
- void **METERLIB1PH_ReadResults** (tMETERLIB1PH_DATA *p, double *urms, double *irms, double *pavg, double *qavg, double *s);
  Return non-billing measurements: IRMS, URMS, PAVG, QAVG, and S.
- void **METERLIB1PH_ReadURMS** (tMETERLIB1PH_DATA *p, double *urms1);
  Return URMS.
- void **METERLIB1PH_ReadIRMS** (tMETERLIB1PH_DATA *p, double *irms1);
  Return IRMS.
- void **METERLIB1PH_ReadPAVG** (tMETERLIB1PH_DATA *p, double *pavg1);
  Return PAVG.
- void **METERLIB1PH_ReadS** (tMETERLIB1PH_DATA *p, double *s1);
  Return S.

4.1.2.2. Two-Phase power meter

- void **METERLIB2PH_ProcSamples** (tMETERLIB2PH_DATA *p, frac24 u1Q, frac24 i1Q, frac24 u2Q, frac24 i2Q, frac16 *shift);
  Remove DC bias from phase voltage and phase current samples, together with performing an optional sensor phase shift correction.
- void **METERLIB2PH_CalcWattHours** (tMETERLIB2PH_DATA *p, tENERGY_CNT *pCnt, frac64 puRes);
  Recalculate active energy using new voltage and current samples.
- void **METERLIB2PH_CalcVarHours** (tMETERLIB2PH_DATA *p, tENERGY_CNT *pCnt, frac64 puRes);
  Recalculate reactive energy.
- void **METERLIB2PH_CalcAuxiliary** (tMETERLIB2PH_DATA *p);
  Recalculate auxiliary variables; IRMS, URMS, P, Q, and S.
- void **METERLIB2PH_CalcURMS** (tMETERLIB2PH_DATA *p);
  Recalculate URMS.
- void **METERLIB2PH_CalcIRMS** (tMETERLIB2PH_DATA *p);
  Recalculate IRMS.
- void **METERLIB2PH_CalcPAVG** (tMETERLIB2PH_DATA *p);
  Recalculate PAVG.

- void **METERLIB2PH_ReadResultsPh1** (tMETERLIB2PH_DATA *p, double *urms, double *irms, double *pavg, double *qavg, double *s);

  Return non-billing measurements for Phase1: IRMS, URMS, PAVG, QAVG, and S.
- void **METERLIB2PH_ReadResultsPh2** (tMETERLIB2PH_DATA *p, double *urms, double *irms, double *pavg, double *qavg, double *s);

  Return non-billing measurements for Phase2: IRMS, URMS, PAVG, QAVG, and S.
- void **METERLIB2PH_ReadURMS** (tMETERLIB2PH_DATA *p, double *urms1, double *urms2);

  Return URMS.
- void **METERLIB2PH_ReadIRMS** (tMETERLIB2PH_DATA *p, double *irms1, double *irms2);

  Return IRMS.
- void **METERLIB2PH_ReadPAVG** (tMETERLIB2PH_DATA *p, double *pavg1, double *pavg2);

  Return PAVG.
- void **METERLIB2PH_ReadS** (tMETERLIB2PH_DATA *p, double *s1, double *s2);

  Return S.

4.1.2.3. Three-Phase power meter

- int **METERLIB3PH_ProcSamples** (tMETERLIB3PH_DATA *p, frac24 u1Q, frac24 i1Q, frac24 u2Q, frac24 i2Q, frac24 u3Q, frac24 i3Q, frac16 *shift);
  Remove DC bias from phase voltage and phase current samples, together with determining the phase sequence and performing an optional sensor phase shift correction.
- void **METERLIB3PH_CalcWattHours** (tMETERLIB3PH_DATA *p, tENERGY_CNT *pCnt, frac64 puRes);

  Recalculate active energy using new voltage and current samples.
- void **METERLIB3PH_CalcVarHours** (tMETERLIB3PH_DATA *p, tENERGY_CNT *pCnt, frac64 puRes);
  Recalculate reactive energy.
- void **METERLIB3PH_CalcAuxiliary** (tMETERLIB3PH_DATA *p);

  Recalculate auxiliary variables; IRMS, URMS, P, Q, and S.
- void **METERLIB3PH_CalcURMS** (tMETERLIB3PH_DATA *p);

  Recalculate URMS.
- void **METERLIB3PH_CalcIRMS** (tMETERLIB3PH_DATA *p);

  Recalculate IRMS.
- void **METERLIB3PH_CalcPAVG** (tMETERLIB3PH_DATA *p);

  Recalculate PAVG.
- void **METERLIB3PH_ReadResultsPh1** (tMETERLIB3PH_DATA *p, double *urms, double *irms, double *pavg, double *qavg, double *s);

  Return non-billing measurements for Phase1: IRMS, URMS, PAVG, QAVG, and S.

- void **METERLIB3PH_ReadResultsPh2** (tMETERLIB3PH_DATA *p, double *urms, double *irms, double *pavg, double *qavg, double *s);

  Return non-billing measurements for Phase2: IRMS, URMS, PAVG, QAVG, and S.
- void **METERLIB3PH_ReadResultsPh3** (tMETERLIB3PH_DATA *p, double *urms, double *irms, double *pavg, double *qavg, double *s);

  Return non-billing measurements for Phase3: IRMS, URMS, PAVG, QAVG, and S.
- void **METERLIB3PH_ReadURMS** (tMETERLIB3PH_DATA *p, double *urms1, double *urms2, double *urms3);

  Return URMS.
- void **METERLIB3PH_ReadIRMS** (tMETERLIB3PH_DATA *p, double *irms1, double *irms2, double *irms3);

  Return IRMS.
- void **METERLIB3PH_ReadPAVG** (tMETERLIB3PH_DATA *p, double *pavg1, double *pavg2, double *pavg3);

  Return PAVG.
- void **METERLIB3PH_ReadS** (tMETERLIB3PH_DATA *p, double *s1, double *s2, double *s3);
  Return S.

4.1.2.4. Auxiliary macros

- #define **METERLIB_KWH_PD** (p);

  Return fine delay of the active energy pulse output transition…
- #define **METERLIB_KWH_PS** (p);

  Return raw state of the active energy pulse output.
- #define **METERLIB_KVARH_PD** (p);

  Return fine delay of the reactive energy pulse output transition…
- #define **METERLIB_KVARH_PS** (p);

  Return raw state of the reactive energy pulse output.
- #define **METERLIB_KWH_PR** (x);

  This macro converts imp/kWh number to numeric representation required by the high-precision library.
- #define **METERLIB_KVARH_PR** (x);

  This macro converts imp/kVARh number to numeric representation required by the high-precision library.
- #define **METERLIB_DEG2SH** (x,fn);

  This macro converts U-I phase shift in degrees to a 16-bit fractional number.
- #define **METERLIB_RAD2SH** (x,fn);

  This macro converts U-I phase shift in radians to a 16-bit fractional number.

4.1.3. Low-power library function API

This section summarizes functions API defined in the low-power metering library meterliblp.lib. Prototypes of all functions and internal data structures are declared in the meterliblp.h header file.

4.1.3.1. One-Phase power meter

- void **METERLIBLP1PH_ProcSamples** (tMETERLIBLP1PH_DATA *p, frac24 u1Q, frac24 i1Q, frac16 *shift);

  Remove DC bias from phase voltage and phase current samples, together with performing an optional sensor phase shift correction.
- void **METERLIBLP1PH_CalcWattHours** (tMETERLIBLP1PH_DATA *p, tENERGY_CNT *pCnt, frac64 puRes);

  Recalculate active energy using new voltage and current samples.
- void **METERLIBLP1PH_CalcVarHours** (tMETERLIBLP1PH_DATA *p, tENERGY_CNT *pCnt, frac64 puRes);
  Recalculate reactive energy.
- void **METERLIBLP1PH_CalcAuxiliary** (tMETERLIBLP1PH_DATA *p);

  Recalculate auxiliary variables; IRMS, URMS, P, Q, and S.
- void **METERLIBLP1PH_CalcURMS** (tMETERLIBLP1PH_DATA *p);

  Recalculate URMS.
- void **METERLIBLP1PH_CalcIRMS** (tMETERLIBLP1PH_DATA *p);

  Recalculate IRMS.
- void **METERLIBLP1PH_CalcPAVG** (tMETERLIBLP1PH_DATA *p);

  Recalculate PAVG.
- void **METERLIBLP1PH_ReadResults** (tMETERLIBLP1PH_DATA *p, double *urms, double *irms, double *pavg, double *qavg, double *s);

  Return non-billing measurements: IRMS, URMS, PAVG, QAVG, and S.
- void **METERLIBLP1PH_ReadURMS** (tMETERLIBLP1PH_DATA *p, double *urms);

  Return URMS.
- void **METERLIBLP1PH_ReadIRMS** (tMETERLIBLP1PH_DATA *p, double *irms);

  Return IRMS.
- void **METERLIBLP1PH_ReadPAVG** (tMETERLIBLP1PH_DATA *p, double *pavg);

  Return PAVG.
- void **METERLIBLP1PH_ReadS** (tMETERLIBLP1PH_DATA *p, double *s);

  Return S.

4.1.3.2. Two-Phase power meter

- void **METERLIBLP2PH_ProcSamples** (tMETERLIBLP2PH_DATA *p, frac24 u1Q, frac24 i1Q, frac24 u2Q, frac24 i2Q, frac16 *shift);
  Remove DC bias from phase voltage and phase current samples, together with performing an optional sensor phase shift correction.
- void **METERLIBLP2PH_CalcWattHours** (tMETERLIBLP2PH_DATA *p, tENERGY_CNT *pCnt, frac64 puRes);
  Recalculate active energy using new voltage and current samples.
- void **METERLIBLP2PH_CalcVarHours** (tMETERLIBLP2PH_DATA *p, tENERGY_CNT *pCnt, frac64 puRes);
  Recalculate reactive energy.
- void **METERLIBLP2PH_CalcAuxiliary** (tMETERLIBLP2PH_DATA *p);
  Recalculate auxiliary variables; IRMS, URMS, P, Q, and S.
- void **METERLIBLP2PH_CalcURMS** (tMETERLIBLP2PH_DATA *p);
  Recalculate URMS.
- void **METERLIBLP2PH_CalcIRMS** (tMETERLIBLP2PH_DATA *p);
  Recalculate IRMS.
- void **METERLIBLP2PH_CalcPAVG** (tMETERLIBLP2PH_DATA *p);
  Recalculate PAVG.
- void **METERLIBLP2PH_ReadResultsPh1** (tMETERLIBLP2PH_DATA *p, double *urms, double *irms, double *pavg, double *qavg, double *s);
  Return non-billing measurements for Phase1: IRMS, URMS, PAVG, QAVG, and S.
- void **METERLIBLP2PH_ReadResultsPh2** (tMETERLIBLP2PH_DATA *p, double *urms, double *irms, double *pavg, double *qavg, double *s);
  Return non-billing measurements for Phase2: IRMS, URMS, PAVG, QAVG, and S.
- void **METERLIBLP2PH_ReadURMS** (tMETERLIBLP2PH_DATA *p, double *urms1, double *urms2);
  Return URMS.
- void **METERLIBLP2PH_ReadIRMS** (tMETERLIBLP2PH_DATA *p, double *irms1, double *irms2);
  Return IRMS.
- void **METERLIBLP2PH_ReadPAVG** (tMETERLIBLP2PH_DATA *p, double *pavg1, double *pavg2);
  Return PAVG.
- void **METERLIBLP2PH_ReadS** (tMETERLIBLP2PH_DATA *p, double *s1, double *s2);
  Return S.

4.1.3.3. Three-Phase power meter

- int **METERLIBLP3PH_ProcSamples** (tMETERLIBLP3PH_DATA *p, frac24 u1Q, frac24 i1Q, frac24 u2Q, frac24 i2Q, frac24 u3Q, frac24 i3Q, frac16 *shift);
  Remove DC bias from phase voltage and phase current samples, together with determining the phase sequence and performing an optional sensor phase shift correction.
- void **METERLIBLP3PH_CalcWattHours** (tMETERLIBLP3PH_DATA *p, tENERGY_CNT *pCnt, frac64 puRes);
  Recalculate active energy using new voltage and current samples.
- void **METERLIBLP3PH_CalcVarHours** (tMETERLIBLP3PH_DATA *p, tENERGY_CNT *pCnt, frac64 puRes);
  Recalculate reactive energy.
- void **METERLIBLP3PH_CalcAuxiliary** (tMETERLIBLP3PH_DATA *p);
  Recalculate auxiliary variables; IRMS, URMS, P, Q, and S.
- void **METERLIBLP3PH_CalcURMS** (tMETERLIBLP3PH_DATA *p);
  Recalculate URMS.
- void **METERLIBLP3PH_CalcIRMS** (tMETERLIBLP3PH_DATA *p);
  Recalculate IRMS.
- void **METERLIBLP3PH_CalcPAVG** (tMETERLIBLP3PH_DATA *p);
  Recalculate PAVG.
- void **METERLIBLP3PH_ReadResultsPh1** (tMETERLIBLP3PH_DATA *p, double *urms, double *irms, double *pavg, double *qavg, double *s);
  Return non-billing measurements for Phase1: IRMS, URMS, PAVG, QAVG, and S.
- void **METERLIBLP3PH_ReadResultsPh2** (tMETERLIBLP3PH_DATA *p, double *urms, double *irms, double *pavg, double *qavg, double *s);
  Return non-billing measurements for Phase2: IRMS, URMS, PAVG, QAVG, and S.
- void **METERLIBLP3PH_ReadResultsPh3** (tMETERLIBLP3PH_DATA *p, double *urms, double *irms, double *pavg, double *qavg, double *s);
  Return non-billing measurements for Phase3: IRMS, URMS, PAVG, QAVG, and S.
- void **METERLIBLP3PH_ReadURMS** (tMETERLIBLP3PH_DATA *p, double *urms1, double *urms2, double *urms3);
  Return URMS.
- void **METERLIBLP3PH_ReadIRMS** (tMETERLIBLP3PH_DATA *p, double *irms1, double *irms2, double *irms3);
  Return IRMS.
- void **METERLIBLP3PH_ReadPAVG** (tMETERLIBLP3PH_DATA *p, double *pavg1, double *pavg2, double *pavg3);
  Return PAVG.
- void **METERLIBLP3PH_ReadS** (tMETERLIBLP3PH_DATA *p, double *s1, double *s2, double *s3);
  Return S.

4.1.3.4. Auxiliary macros

- #define **METERLIBLP_KWH_PD** (p);
  Return fine delay of the active energy pulse output transition…
- #define **METERLIBLP_KWH_PS** (p);
  Return raw state of the active energy pulse output.
- #define **METERLIBLP_KVARH_PD** (p);
  Return fine delay of the reactive energy pulse output transition…
- #define **METERLIBLP_KVARH_PS** (p);
  Return raw state of the reactive energy pulse output.
- #define **METERLIBLP_KWH_PR** (x);
  This macro converts imp/kWh number to numeric representation required by the low-power library.
- #define **METERLIBLP_KVARH_PR** (x);
  This macro converts imp/kVARh number to numeric representation required by the low-power library.
- #define **METERLIBLP_DEG2SH** (x,fn);
  This macro converts U-I phase shift in degrees to a 16-bit fractional number.
- #define **METERLIBLP_RAD2SH** (x,fn);
  This macro converts U-I phase shift in radians to a 16-bit fractional number.

4.1.4. Data structures

This section describes the data structures for accessing those state variables calculated by both the high-precision and low-power metering libraries.

4.1.4.1. tCNT

Structure containing energy counter.

Reference

```
#include "meterlib.h"
#include "meterliblp.h"
```

Data fields

Type	Name	Description
unsigned long	ex	Counter for exported energy.
unsigned long	im	Counter for imported energy.
unsigned long	Q[4]	Reactive energy counters in four quadrant system

4.1.4.2. tENERGY_CNT

Structure containing energy counters for three three-phase system.

Reference

```
#include "meterlib.h"
#include "meterliblp.h"
```

Data fields

Type	Name	Description
tCNT	ph[3]	Energy counters in phases of the three-phase system.

4.1.5. METERLIB_ProcSamples

These functions remove DC bias from measured phase voltage and phase current samples, together with performing an optional sensor phase shift correction. In addition, the function for three-phase system determines and returns the phase sequence.

Syntax

```
#include "meterlib.h"
void METERLIB1PH_ProcSamples (tMETERLIB1PH_DATA *p, frac24 u1Q, frac24 i1Q, frac16 *shift);
void METERLIB2PH_ProcSamples (tMETERLIB2PH_DATA *p, frac24 u1Q, frac24 i1Q, frac24 u2Q,
frac24 i2Q, frac16 *shift);
int  METERLIB3PH_ProcSamples (tMETERLIB3PH_DATA *p, frac24 u1Q, frac24 i1Q, frac24 u2Q,
frac24 i2Q, frac24 u3Q, frac24 i3Q, frac16 *shift);

#include "meterliblp.h"
void METERLIBLP1PH_ProcSamples (tMETERLIBLP1PH_DATA *p, frac24 u1Q, frac24 i1Q, frac16
*shift);
void METERLIBLP2PH_ProcSamples (tMETERLIBLP2PH_DATA *p, frac24 u1Q, frac24 i1Q, frac24 u2Q,
frac24 i2Q, frac16 *shift);
int  METERLIBLP3PH_ProcSamples (tMETERLIBLP3PH_DATA *p, frac24 u1Q, frac24 i1Q, frac24 u2Q,
frac24 i2Q, frac24 u3Q, frac24 i3Q, frac16 *shift);
```

Arguments

Type	Name	Direction	Description
tMETERLIB1PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB1PH_DATA.
tMETERLIB2PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB2PH_DATA.
tMETERLIB3PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB3PH_DATA.
tMETERLIBLP1PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP1PH_DATA.
tMETERLIBLP2PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP2PH_DATA.
tMETERLIBLP3PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP2PH_DATA.
frac24	u1Q	in	Phase 1 instantaneous voltage sample in Q0.31 data format.
frac24	i1Q	in	Phase 1 instantaneous current sample in Q0.31 data format.
frac24	u2Q	in	Phase 2 instantaneous voltage sample in Q0.31 data format.
frac24	i2Q	in	Phase 2 instantaneous current sample in Q0.31 data format.
frac24	u3Q	in	Phase 3 instantaneous voltage sample in Q0.31 data format.
frac24	i3Q	in	Phase 3 instantaneous current sample in Q0.31 data format.
frac16	shift	in	Pointer to the values for U-I phase shift correction. Set each value in the range -32768...32767 to phase shift the voltage with resolution of 1/(32768*KWH_CALC_FREQ) seconds – for more details, refer to METERLIBLP_DEG2SH() and METERLIBLP_RAD2SH() macros. Use the NULL pointer to disable software sensor phase shift correction.

Return

Only a function for three-phase system returns the phase sequence. Functions for 1- and two-phase systems do not return any arguments.

Description

These functions remove the DC offset and low-frequency drift from the measured phase voltage and phase current samples. They also perform an optional sensor phase shift correction. In addition, the functions for three-phase system determine and return the phase sequence.

Figure 19 shows block diagram of the METERLIB1PH_ProcSamples() function defined by the high-precision metering library.

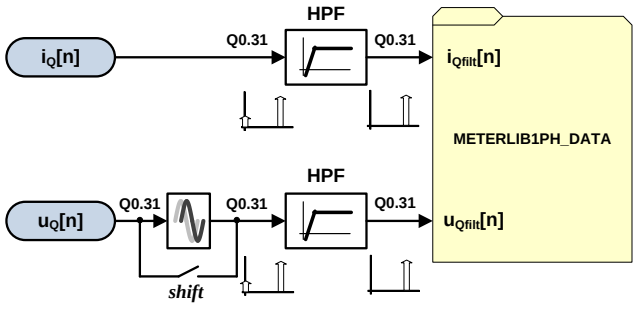


Figure 19. **METERLIB1PH_ProcSamples function block diagram**

Figure 20 shows block diagram of the METERLIBLP1PH_ProcSamples() function defined by the low-power metering library.

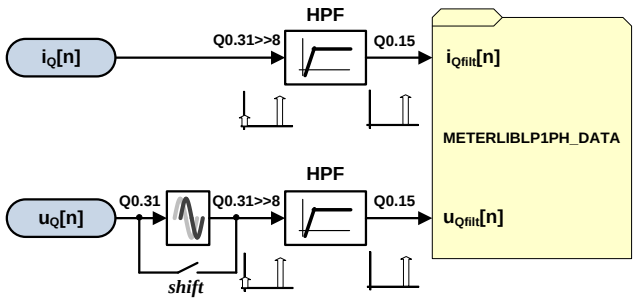


Figure 20. **METERLIBLP1PH_ProcSamples function block diagram**

These functions are specific to a one-phase power meter. They process phase current samples $i_Q[n]$ and phase voltage samples $u_Q[n]$ in Q0.31 format to attenuate any DC offsets and low-frequency drifts. The "HPF" blocks represent the high-pass first order Butterworth IIR filters (see Infinite impulse response filter). The physical quantities of the phase voltage u[n] and phase current i[n] are greater than one, therefore, substitutions are introduced to allow use of fractional arithmetic. The actual phase voltage and phase current are scaled into fractional representation by U_MAX and I_MAX.

$$u_Q = \frac{u}{U\_MAX} \qquad \textit{Eq. 29}$$

$$i_Q = \frac{i}{I\_MAX} \qquad \textit{Eq. 30}$$

where, U_MAX and I_MAX are the maximum physical values of the phase voltage and phase current that correspond to the full AFE analog input range.

NOTE

The fractional representation of the phase voltage u_Q and phase current i_Q can be read directly from the AFE, provided data in the result registers is represented in the right justified 2's complement 32-bit data format (see Figure 2).

Performance

Function	Code size	Stack size	Clock cycles[2]
METERLIB1PH_ProcSamples	936	72	448 (535)
METERLIB2PH_ProcSamples	1880	72	872 (1048)
METERLIB3PH_ProcSamples	2982	120	1429 (1699)
METERLIBLP1PH_ProcSamples	294	28	132 (215)
METERLIBLP2PH_ProcSamples	664	48	274 (483)
METERLIBLP3PH_ProcSamples	1044	44	442 (745)

4.1.6. METERLIB_CalcWattHours

These functions recalculate active energy using new unbiased phase voltage and phase current samples.

[2] Clock cycles in brackets denote function execution time with sensor phase shift correction enabled; i.e. "shift" pointer is not NULL but points to values in frac16 representation.

Syntax

```
#include "meterlib.h"
void METERLIB1PH_CalcWattHours (tMETERLIB1PH_DATA *p, tENERGY_CNT  *pCnt, frac64 puRes);
void METERLIB2PH_CalcWattHours (tMETERLIB2PH_DATA *p, tENERGY_CNT  *pCnt, frac64 puRes);
void METERLIB3PH_CalcWattHours (tMETERLIB3PH_DATA *p, tENERGY_CNT  *pCnt, frac64 puRes);

#include "meterliblp.h"
void METERLIBLP1PH_CalcWattHours (tMETERLIBLP1PH_DATA *p, tENERGY_CNT  *pCnt, frac64 puRes);
void METERLIBLP2PH_CalcWattHours (tMETERLIBLP2PH_DATA *p, tENERGY_CNT  *pCnt, frac64 puRes);
void METERLIBLP3PH_CalcWattHours (tMETERLIBLP3PH_DATA *p, tENERGY_CNT  *pCnt, frac64 puRes);
```

Arguments

Type	Name	Direction	Description
tMETERLIB1PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB1PH_DATA.
tMETERLIB2PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB2PH_DATA.
tMETERLIB3PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB3PH_DATA.
tMETERLIBLP1PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP1PH_DATA.
tMETERLIBLP2PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP2PH_DATA.
tMETERLIBLP3PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP2PH_DATA.
tENERGY_CNT	pCnt	out	Pointer to LCD active energy counter structure.
frac64	puRes	in	Pulse output resolution; calculated by METERLIB_KWH_PR() macro.

Return

These functions do not return any arguments.

Description

These functions compute active energy in watt-hours (Wh) and return the state of active energy pulse output. The active energy in a typical one-phase power meter application is computed as an infinite integral of the unbiased instantaneous phase voltage $u(t)$ and phase current $i(t)$ waveforms.

$$\boldsymbol{Wh = \frac{1}{3600}\int_0^{\infty} u(t)\, i(t) dt} \qquad \textit{Eq. 31}$$

The Backward Euler approximation of the integral term will transform the basic equation Eq. 31 into a difference form:

$$\boldsymbol{Wh[n] = Wh[n-1] + \frac{u[n] * i[n] * \Delta t}{3600}} \qquad \textit{Eq. 32}$$

where, $\Delta t = 1/f_s$ is the sampling interval, index $[n]$ represents the current value and index $[n-1]$ represents the previous value calculated in the previous calculation step.

In equation Eq. 32, the physical quantities $u[n]$, $i[n]$, $Wh[n]$ and $Wh[n-1]$ are scaled by U_MAX, I_MAX and Δt to allow implementation of fractional arithmetic.

$$\boldsymbol{Wh_Q[n] = Wh_Q[n-1] + u_Q[n] * i_Q[n]} \qquad \textit{Eq. 33}$$

The active energy in fractional representation is mapped into the fractional range $-1 \leq Wh_Q \leq 1 - 2^{(N-1)}$ using scaling:

$$Wh_Q = Wh\frac{3600 * f_s}{U\_MAX * I\_MAX} \quad \textit{Eq. 34}$$

Figure 21 shows a block diagram of the METERLIB1PH_CalcWattHours() function defined by the high-precision library and the way of calculating active energy in a typical one-phase power meter.

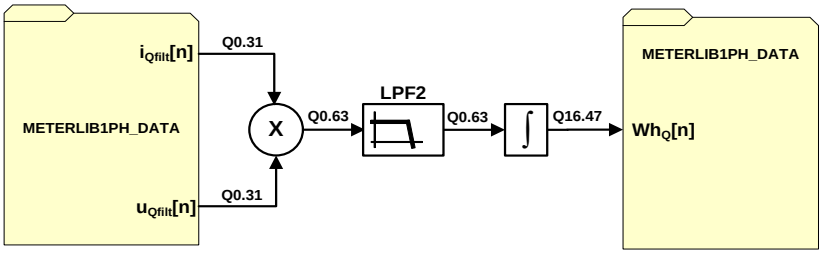


Figure 21. **METERLIB1PH_CalcWattHours function block diagram**

The new active energy sample $Wh_Q[n]$ is computed according to Eq. 33 using fractional addition and multiplication functions. The "Multiplier" block represents a 32x32=64 bit fractional multiplication and an "Integrator" block represents a 64+64=64 bit fractional addition. The new active energy sample is smoothed by the "LPF2" low-pass first order Butterworth IIR filter. This low-pass filter attenuates the alternating active energy component that is developed by multiplication of two 50/60 Hz sinusoidal waveforms, the alternating instantaneous voltage and current, and that ripples at twice the load frequency (100/120 Hz).

Figure 22 shows a block diagram of the METERLIBLP1PH_CalcWattHours() function defined by the low-power library. The new active energy sample $Wh_Q[n]$ is also computed according to Eq. 33 using fractional addition and multiplication functions. The "Multiplier" block represents a 16x16=32 bit fractional multiplication and an "Integrator" block represents a 64+64=64 bit fractional addition. The new active energy sample is not smoothed to save computation power.

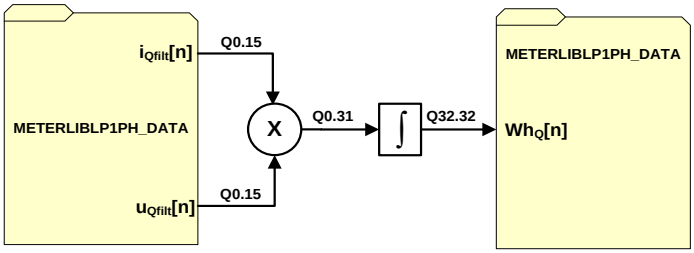


Figure 22. **METERLIBLP1PH_CalcWattHours function block diagram**

NOTE

The active energy calculations in two-phase (Form-12S) and three-phase power meters are performed in a similar way as in the one-phase power meter, with the one exception that the infinite integral is calculated over the sum of the unbiased instantaneous phase voltage u(t) and phase current i(t) waveforms of all the phases.

Performance

Function	Code size	Stack size	Clock cycles
METERLIB1PH_CalcWattHours	710	104	893
METERLIB2PH_CalcWattHours	1124	112	1439
METERLIB3PH_CalcWattHours	1496	120	2052
METERLIBLP1PH_CalcWattHours	472	56	450
METERLIBLP2PH_CalcWattHours	678	88	594
METERLIBLP3PH_CalcWattHours	900	64	704

4.1.7. METERLIB_CalcVarHours

These functions recalculate the reactive energy using new phase voltage and phase current samples.

Syntax

```
#include "meterlib.h"
void METERLIB1PH_CalcVarHours (tMETERLIB1PH_DATA *p, tENERGY_CNT  *pCnt, frac64 puRes);
void METERLIB2PH_CalcVarHours (tMETERLIB2PH_DATA *p, tENERGY_CNT  *pCnt, frac64 puRes);
void METERLIB3PH_CalcVarHours (tMETERLIB3PH_DATA *p, tENERGY_CNT  *pCnt, frac64 puRes);

#include "meterliblp.h"
void METERLIBLP1PH_CalcVarHours (tMETERLIBLP1PH_DATA *p, tENERGY_CNT  *pCnt, frac64 puRes);
void METERLIBLP2PH_CalcVarHours (tMETERLIBLP2PH_DATA *p, tENERGY_CNT  *pCnt, frac64 puRes);
void METERLIBLP3PH_CalcVarHours (tMETERLIBLP3PH_DATA *p, tENERGY_CNT  *pCnt, frac64 puRes);
```

Arguments

Type	Name	Direction	Description
tMETERLIB1PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB1PH_DATA.
tMETERLIB2PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB2PH_DATA.
tMETERLIB3PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB3PH_DATA.
tMETERLIBLP1PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP1PH_DATA.
tMETERLIBLP2PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP2PH_DATA.
tMETERLIBLP3PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP2PH_DATA.
tENERGY_CNT	pCnt	out	Pointer to LCD reactive energy counter structure.
frac64	puRes	in	Pulse output resolution; calculated by METERLIB_KVARH_PR() macro.

Return

These functions do not return any arguments.

Description

These functions compute reactive energy in the unit of volt-ampere-reactive hours (VARh) and return the state of reactive energy pulse output. The reactive energy in a typical one-phase power meter is computed as an infinite integral of the unbiased instantaneous shifted phase voltage $u(t-90°)$ and phase current $i(t)$ waveforms.

$$VARh = \frac{1}{3600}\int_0^\infty u(t-90°)\,i(t)\,dt \qquad \textit{Eq. 35}$$

The Backward Euler approximation of the integral term will transform the basic equation Eq. 35 into a difference form:

$$VARh[n] = VARh[n-1] + \frac{u90[n] * idel[n] * \Delta t}{3600} \quad \textit{Eq. 36}$$

where, $\Delta t = 1/f_s$ is the sampling time, index [n] represents the current value and index $[n-1]$ represents the old value calculated in the previous calculation step.

In equation Eq. 37, the physical quantities u90[n], idel[n], VARh[n], and VARh$[n-1]$ are scaled by U_MAX, I_MAX, and Δt to allow usage of fractional arithmetic.

$$VARh_Q[n] = VARh_Q[n-1] + u90_Q[n] * idel_Q[n] \quad \textit{Eq. 37}$$

The reactive energy in fractional representation is mapped into the fractional range $-1 \leq VARh_Q \leq 1 - 2^{(N-1)}$ using scaling:

$$VARh_Q = VARh \frac{3600 * f_s}{U\_MAX * I\_MAX} \quad \textit{Eq. 38}$$

NOTE

Reactive energy Eq. 37 can be efficiently computed on the microcontroller using fractional addition and multiplication functions. The phase voltage instantaneous sample $u90_Q[n]$ is shifted by 90-degrees from the delayed phase current instantaneous sample $idel_Q[n]$ using the Hilbert transformer.

Figure 23 shows a block diagram of the METERLIB1PH_CalcVarHours() function defined by the high-precision metering library.

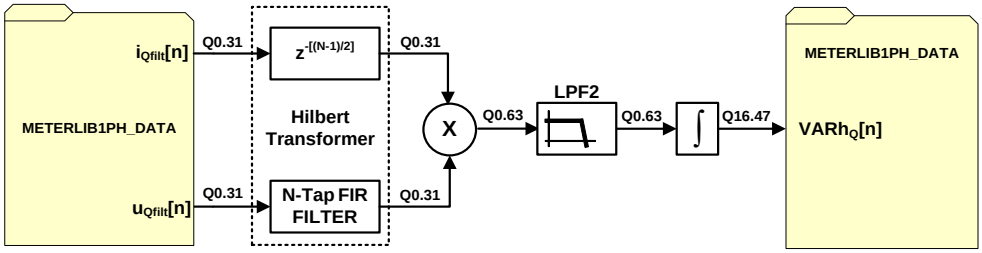


Figure 23. **METERLIB1PH_CalcVarHours function block diagram**

Figure 24 shows a block diagram of the METERLIBLP1PH_CalcVarHours() function defined by the low-power metering library.

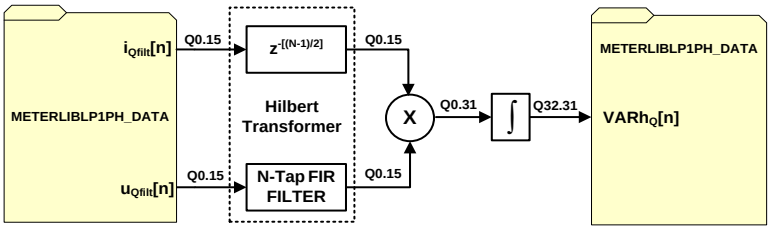


Figure 24. **METERLIBLP1PH_CalcVarHours function block diagram**

These functions calculate reactive energy in a typical one-phase power meter application. The Hilbert transformer block represents an N-Tap FIR filter with a 90-degree constant phase response plus a linear

phase response with the group delay $M = (N - 1)/2$. Therefore, when the phase voltage instantaneous sample $u_Q[n]$ passes through the N-Tap FIR filter, the output sample $u90_Q[n]$ is related to the delayed phase current sample $idel_Q[n]$ through the Hilbert transform (for more information, refer Ideal Hilbert transformer). The reactive energy $VARh_Q[n]$ is computed according to Eq. 37. Similarly to the active energy processing, the reactive energy is also smoothed using a low-pass filter to attenuate the alternating energy component.

The high-precision metering library defines three functions for calculating reactive energy. The first, METERLIB1PH_CalcVarHours() function is intended to calculate reactive energy in a one-phase power meter application. Others, METERLIB2PH_CalcVarHours() and METERLIB3PH_CalcVarHours() functions shall be called in two-phase (Form-12S) and three-phase metering power meter use cases. The low-power metering library defines the same set of functions.

NOTE

Functions for calculating reactive energy in multiple phases calculate the Hilbert transform and reactive energy contribution of each phase and return the sum of all reactive energies.

Performance

Function	Code size	Stack size	Clock cycles[3]
METERLIB1PH_CalcVarHours	882	104	3463
METERLIB2PH_CalcVarHours	1484	112	6578
METERLIB3PH_CalcVarHours	2128	136	9730
METERLIBLP1PH_CalcVarHours	630	64	2088
METERLIBLP2PH_CalcVarHours	1050	72	3849
METERLIBLP3PH_CalcVarHours	1428	72	5608

4.1.8. METERLIB_CalcAuxiliary

These functions recalculate non-billing variables such as active power (P), reactive power (Q), RMS voltage (URMS), and RMS current (IRMS).

Syntax

```
#include "meterlib.h"
void METERLIB1PH_CalcAuxiliary (tMETERLIB1PH_DATA *p);
void METERLIB2PH_CalcAuxiliary (tMETERLIB2PH_DATA *p);
void METERLIB3PH_CalcAuxiliary (tMETERLIB3PH_DATA *p);

#include "meterliblp.h"
void METERLIBLP1PH_CalcAuxiliary (tMETERLIBLP1PH_DATA *p);
void METERLIBLP2PH_CalcAuxiliary (tMETERLIBLP2PH_DATA *p);
void METERLIBLP3PH_CalcAuxiliary (tMETERLIBLP3PH_DATA *p);
```

[3] Performance obtained for 49-Tap FIR filter (default filter length for 1200 Hz sampling frequency).

Arguments

Type	Name	Direction	Description
tMETERLIB1PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB1PH_DATA.
tMETERLIB2PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB2PH_DATA.
tMETERLIB3PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB3PH_DATA.
tMETERLIBLP1PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP1PH_DATA.
tMETERLIBLP2PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP2PH_DATA.
tMETERLIBLP3PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP2PH_DATA.

Return

These functions do not return any arguments.

Description

These functions calculate active power (P), reactive power (Q), RMS voltage (URMS), and RMS current (IRMS) in one-phase, two-phase (Form-12S), and three-phase power meter applications.

The active power is measured in watts (W) and is symbolized by the capital letter P. The reactive power is measured in volt-amperes-reactive (VAR) and is symbolized by the capital letter Q. The library function uses the average power converter to calculate active and reactive powers (see Average power converter).

The following figure shows block diagram of the METERLIB1PH_CalcAuxiliary() function (power calculation portion) defined by the high-precision metering library.

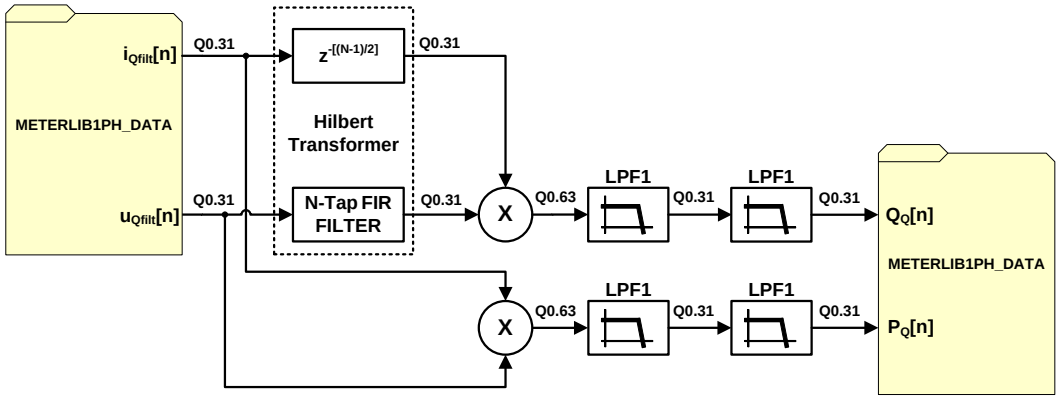


Figure 25. **METERLIB1PH_CalcAuxiliary function block diagram – powers calculation**

This figure shows block diagram of the METERLIBLP1PH_CalcAuxiliary() function (power calculation portion) defined by the low-power metering library:

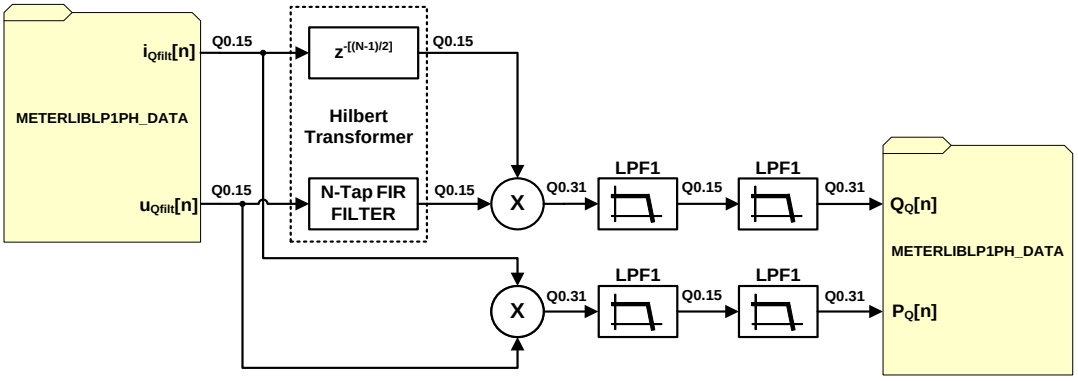


Figure 26. **METERLIBLP1PH_CalcAuxiliary function block diagram – powers calculation**

These portions update the active power $P_Q[n]$ and reactive power $Q_Q[n]$ samples based on new unbiased phase voltage and phase current samples. Needless to say, both powers are in fractional data format to accelerate and simplify the calculations. They are mapped into the physical representation using scaling:

$$P_Q = \frac{P}{U\_MAX * I\_MAX} \qquad \textit{Eq. 39}$$

$$Q_Q = \frac{Q}{U\_MAX * I\_MAX} \qquad \textit{Eq. 40}$$

The RMS values are calculated using the explicit RMS converter approach (see Explicit RMS converter). This figure shows block diagram of the METERLIB1PH_CalcAuxiliary() function (RMS calculation portion) defined by the high-precision metering library:

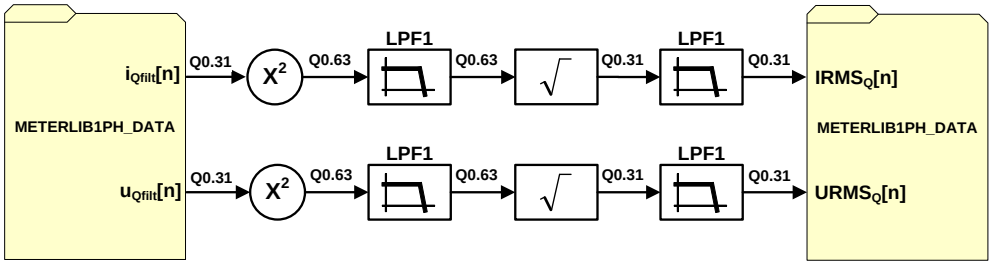


Figure 27. **METERLIB1PH_CalcAuxiliary function block diagram – RMS calculation**

This figure shows block diagram of the METERLIBLP1PH_CalcAuxiliary() function (RMS calculation portion) defined by the low-power metering library:

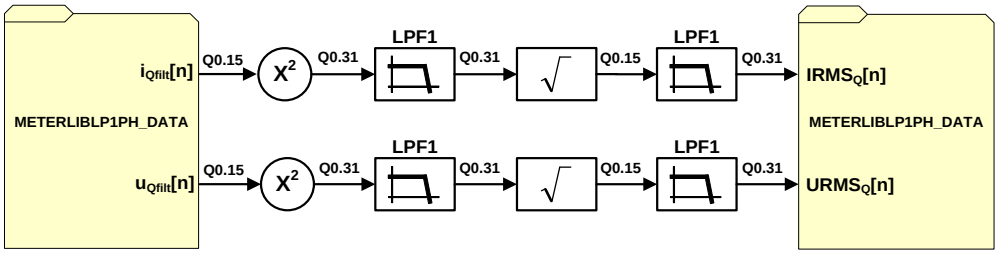


Figure 28. **METERLIBLP1PH_CalcAuxiliary function block diagram – RMS calculation**

Both implementations are based on fractional data format and use of fractional multiplication, addition, and square-root functions. The library implementations differ in dynamic range of calculations. The high-precision library uses 32 and 64 bit fractional data types. On the contrary, the low-power library uses 16 and 32 bit fractional data types. Both implementations use square root calculation based on the *Non-restoring Method* (see Square root).

These functions calculate the RMS voltage and RMS current samples in fractional data format, denoted as $URMS_Q[n]$ and $IRMS_Q[n]$, respectively. They are mapped into the physical representation using scaling:

$$URMS_Q = \frac{URMS}{U\_MAX} \quad \textit{Eq. 41}$$

$$IRMS_Q = \frac{IRMS}{I\_MAX} \quad \textit{Eq. 42}$$

The high-precision metering library defines three functions for calculating non-billing quantities. The first, METERLIB1PH_CalcAuxiliary() function is intended to calculate non-billing quantities in a one-phase power meter application.

Others, METERLIB2PH_ CalcAuxiliary () and METERLIB3PH_ CalcAuxiliary () functions shall be called in two-phase (Form-12S) and three-phase power meter use cases. The low-power metering library defines the same set of functions.

Performance

Function	Code size	Stack size	Clock cycles
METERLIB1PH_CalcAuxiliary	1522	128	3427
METERLIB2PH_CalcAuxiliary	3252	144	6851
METERLIB3PH_CalcAuxiliary	4828	144	10190
METERLIBLP1PH_CalcAuxiliary	738	48	997
METERLIBLP2PH_CalcAuxiliary	1460	48	1991
METERLIBLP3PH_CalcAuxiliary	2128	64	2939

4.1.9. METERLIB_CalcURMS

These functions recalculate RMS voltage (URMS).

Syntax

```
#include "meterlib.h"
void METERLIB1PH_CalcURMS (tMETERLIB1PH_DATA *p);
void METERLIB2PH_CalcURMS (tMETERLIB2PH_DATA *p);
void METERLIB3PH_CalcURMS (tMETERLIB3PH_DATA *p);

#include "meterliblp.h"
void METERLIBLP1PH_CalcURMS (tMETERLIBLP1PH_DATA *p);
void METERLIBLP2PH_CalcURMS (tMETERLIBLP2PH_DATA *p);
void METERLIBLP3PH_CalcURMS (tMETERLIBLP3PH_DATA *p);
```

Arguments

Type	Name	Direction	Description
tMETERLIB1PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB1PH_DATA.
tMETERLIB2PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB2PH_DATA.
tMETERLIB3PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB3PH_DATA.
tMETERLIBLP1PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP1PH_DATA.
tMETERLIBLP2PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP2PH_DATA.
tMETERLIBLP3PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP2PH_DATA.

Return

These functions do not return any arguments.

Description

These functions calculate RMS voltage (URMS) in one-phase, two-phase (Form-12S), and three-phase power meter applications. These functions are complementary to METERLIB_CalcAuxiliary() functions that calculates all non-billing variables including active power (P), reactive power (Q), RMS voltage (URMS) and RMS current (IRMS).

This figure shows block diagram of the METERLIB1PH_CalcURMS() function defined by the high-precision metering library:

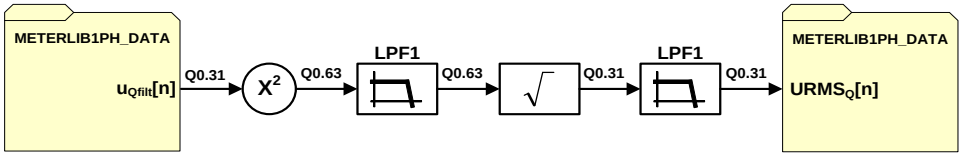


Figure 29. **METERLIB1PH_CalcURMS function block diagram**

This figure shows block diagram of the METERLIBLP1PH_CalcURMS() function defined by the low-power metering library:

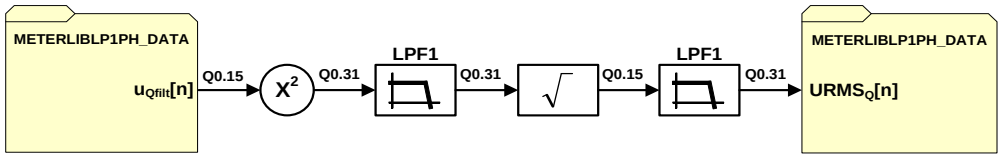


Figure 30. **METERLIBLP1PH_CalcURMS function block diagram**

Both implementations are based on fractional data format and use of fractional multiplication, addition, and square-root functions. These functions calculate the RMS voltage samples in fractional data format, denoted as $URMS_Q[n]$. They are mapped into the physical representation using scaling:

$$URMS_Q = \frac{URMS}{U\_MAX} \qquad \textit{Eq. 43}$$

The high-precision metering library defines three functions for calculating RMS voltages. The first, METERLIB1PH_CalcURMS() function is intended to calculate RMS voltage in a one-phase power meter application. Others, METERLIB2PH_CalcURMS() and METERLIB3PH_CalcURMS() functions

shall be called in two-phase (Form-12S) and three-phase metering power meter use cases. The low-power metering library defines the same set of functions.

Performance

Function	Code size	Stack size	Clock cycles
METERLIB1PH_CalcURMS	364	96	1148
METERLIB2PH_CalcURMS	688	96	2244
METERLIB3PH_CalcURMS	1040	96	3336
METERLIBLP1PH_CalcURMS	170	24	386
METERLIBLP2PH_CalcURMS	354	32	769
METERLIBLP3PH_CalcURMS	498	32	1105

4.1.10. METERLIB_CalcIRMS

These functions recalculate RMS current (IRMS).

Syntax

```
#include "meterlib.h"
void METERLIB1PH_CalcIRMS (tMETERLIB1PH_DATA *p);
void METERLIB2PH_CalcIRMS (tMETERLIB2PH_DATA *p);
void METERLIB3PH_CalcIRMS (tMETERLIB3PH_DATA *p);

#include "meterliblp.h"
void METERLIBLP1PH_CalcIRMS (tMETERLIBLP1PH_DATA *p);
void METERLIBLP2PH_CalcIRMS (tMETERLIBLP2PH_DATA *p);
void METERLIBLP3PH_CalcIRMS (tMETERLIBLP3PH_DATA *p);
```

Arguments

Type	Name	Direction	Description
tMETERLIB1PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB1PH_DATA.
tMETERLIB2PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB2PH_DATA.
tMETERLIB3PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB3PH_DATA.
tMETERLIBLP1PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP1PH_DATA.
tMETERLIBLP2PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP2PH_DATA.
tMETERLIBLP3PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP2PH_DATA.

Return

These functions do not return any arguments.

Description

These functions calculate RMS current (IRMS) in one-phase, two-phase (Form-12S), and three-phase power meter applications. These functions are complementary to METERLIB_CalcAuxiliary() functions that calculates all non-billing variables including active power (P), reactive power (Q), RMS voltage (URMS) and RMS current (IRMS).

This figure shows block diagram of the METERLIB1PH_CalcIRMS() function defined by the high-precision metering library:

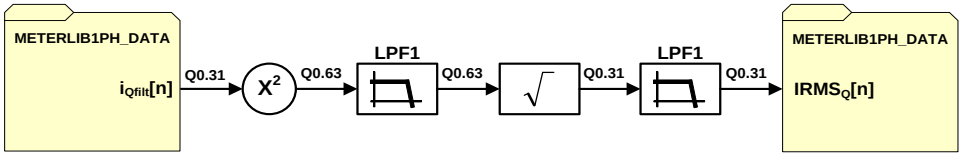


Figure 31. **METERLIB1PH_CalcIRMS function block diagram**

This figure shows block diagram of the METERLIBLP1PH_CalcIRMS() function defined by the low-power metering library:

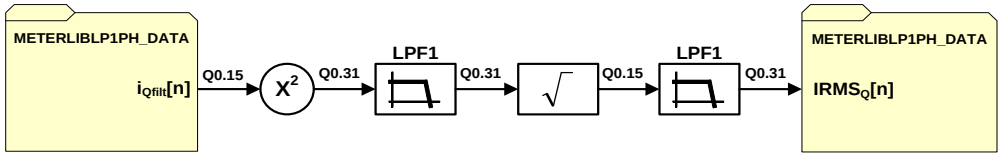


Figure 32. **METERLIBLP1PH_CalcIRMS function block diagram**

Both implementations are based on fractional data format and use extensively fractional multiplication, addition, and square-root functions to calculate RMS values. These functions calculate the RMS current samples in fractional data format, denoted as $IRMS_Q[n]$. They are mapped into the physical representation using scaling:

$$IRMS_Q = \frac{IRMS}{I\_MAX} \qquad \textit{Eq. 44}$$

The high-precision metering library defines three functions for calculating RMS currents. The first, METERLIB1PH_CalcIRMS() function is intended to calculate RMS current in a one-phase power meter application. Others, METERLIB2PH_CalcIRMS() and METERLIB3PH_CalcIRMS() functions shall be called in two-phase (Form-12S) and three-phase metering power meter use cases. The low-power metering library defines the same set of functions.

Performance

Function	Code size	Data size	Clock cycles
METERLIB1PH_CalcIRMS	364	96	1137
METERLIB2PH_CalcIRMS	690	96	2231
METERLIB3PH_CalcIRMS	1042	96	3314
METERLIBLP1PH_CalcIRMS	170	24	366
METERLIBLP2PH_CalcIRMS	354	32	744
METERLIBLP3PH_CalcIRMS	500	32	1069

4.1.11. METERLIB_CalcPAVG

These functions recalculate active power (P).

Syntax

```
#include "meterlib.h"
void METERLIB1PH_CalcPAVG (tMETERLIB1PH_DATA *p);
void METERLIB2PH_CalcPAVG (tMETERLIB2PH_DATA *p);
void METERLIB3PH_CalcPAVG (tMETERLIB3PH_DATA *p);

#include "meterlibLP.h"
```

```
void METERLIBLP1PH_CalcPAVG (tMETERLIBLP1PH_DATA *p);
void METERLIBLP2PH_CalcPAVG (tMETERLIBLP2PH_DATA *p);
void METERLIBLP3PH_CalcPAVG (tMETERLIBLP3PH_DATA *p);
```

Arguments

Type	Name	Direction	Description
tMETERLIB1PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB1PH_DATA.
tMETERLIB2PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB2PH_DATA.
tMETERLIB3PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB3PH_DATA.
tMETERLIBLP1PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP1PH_DATA.
tMETERLIBLP2PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP2PH_DATA.
tMETERLIBLP3PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP2PH_DATA.

Return

These functions do not return any arguments.

Description

These functions calculate active power (P) in one-phase, two-phase (Form-12S), and three-phase power meter applications. The active power is measured in watts (W) and is symbolized by the capital letter P. The library function uses the average power converter to calculate active and reactive powers (see Average power converter).

This figure shows block diagram of the METERLIB1PH_CalcPAVG() function defined by the high-precision metering library:

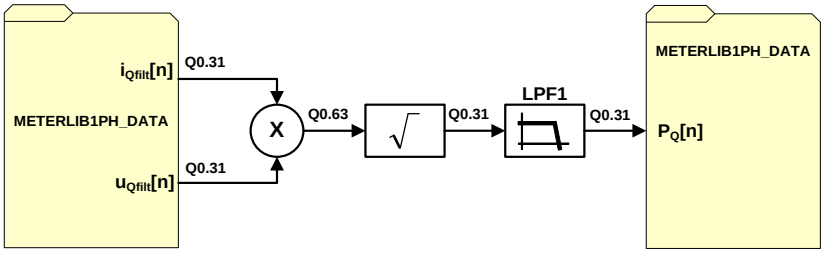


Figure 33. **METERLIB1PH_CalcPAVG function block diagram**

This figure shows block diagram of the METERLIBLP1PH_CalcPAVG() function defined by the low-power metering library:

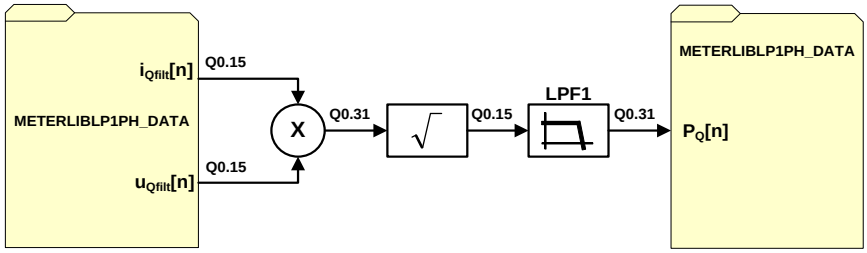


Figure 34. **METERLIBLP1PH_CalcPAVG function block diagram**

These functions update the active power $P_Q[n]$ samples based on new unbiased phase voltage and phase current samples. The active power samples are represented in fractional data format and they're mapped into the physical representation using scaling:

$$P_Q = \frac{P}{U\_MAX * I\_MAX} \qquad \textit{Eq. 45}$$

The high-precision metering library defines three functions for calculating active powers. The first, METERLIB1PH_CalcPAVG() function is intended to calculate active power in a one-phase power meter application. Others, METERLIB2PH_CalcPAVG() and METERLIB3PH_CalcPAVG() functions shall be called in two-phase (Form-12S) and three-phase metering power meter use cases. The low-power metering library defines the same set of functions for computing active power in various power meters.

Performance

Function	Code size	Stack size	Clock cycles
METERLIB1PH_CalcPAVG	400	96	633
METERLIB2PH_CalcPAVG	778	96	1249
METERLIB3PH_CalcPAVG	1150	96	1845
METERLIBLP1PH_CalcPAVG	186	20	129
METERLIBLP2PH_CalcPAVG	388	28	283
METERLIBLP3PH_CalcPAVG	542	24	379

4.1.12. METERLIB_ReadResults

There are three "reading" functions defined in the high-precision and low-power metering libraries. Each function is intended to read non-billing quantities from the internal data structure of the respective metering library and power meter type. The variables returned by the functions are already scaled to the physical representation.

Syntax

```
#include "meterlib.h"
void METERLIB1PH_ReadResults     (tMETERLIB1PH_DATA *p, double *urms, double *irms, double
*pavg, double *qavg, double *s);
void METERLIB2PH_ReadResultsPh1 (tMETERLIB2PH_DATA *p, double *urms, double *irms, double
*pavg, double *qavg, double *s);
void METERLIB2PH_ReadResultsPh2 (tMETERLIB2PH_DATA *p, double *urms, double *irms, double
*pavg, double *qavg, double *s);
void METERLIB3PH_ReadResultsPh1 (tMETERLIB3PH_DATA *p, double *urms, double *irms, double
*pavg, double *qavg, double *s);
void METERLIB3PH_ReadResultsPh2 (tMETERLIB3PH_DATA *p, double *urms, double *irms, double
*pavg, double *qavg, double *s);
void METERLIB3PH_ReadResultsPh3 (tMETERLIB3PH_DATA *p, double *urms, double *irms, double
*pavg, double *qavg, double *s);

#include "meterliblp.h"
void METERLIBLP1PH_ReadResults     (tMETERLIBLP1PH_DATA *p, double *urms, double *irms, double
*pavg, double *qavg, double *s);
void METERLIBLP2PH_ReadResultsPh1 (tMETERLIBLP2PH_DATA *p, double *urms, double *irms, double
*pavg, double *qavg, double *s);
void METERLIBLP2PH_ReadResultsPh2 (tMETERLIBLP2PH_DATA *p, double *urms, double *irms, double
*pavg, double *qavg, double *s);
```

```
void METERLIBLP3PH_ReadResultsPh1 (tMETERLIBLP3PH_DATA *p, double *urms, double *irms, double
*pavg, double *qavg, double *s);
void METERLIBLP3PH_ReadResultsPh2 (tMETERLIBLP3PH_DATA *p, double *urms, double *irms, double
*pavg, double *qavg, double *s);
void METERLIBLP3PH_ReadResultsPh3 (tMETERLIBLP3PH_DATA *p, double *urms, double *irms, double
*pavg, double *qavg, double *s);
```

Arguments

Type	Name	Direction	Description
tMETERLIB1PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB1PH_DATA.
tMETERLIB2PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB2PH_DATA.
tMETERLIB3PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB3PH_DATA.
tMETERLIBLP1PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP1PH_DATA.
tMETERLIBLP2PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP2PH_DATA.
tMETERLIBLP3PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP2PH_DATA.
double	urms	out	Pointer to RMS voltage in volts.
double	irms	out	Pointer to RMS current in amperes.
double	pavg	out	Pointer to active power in watts.
double	qavg	out	Pointer to reactive power in volt-amperes-reactive.
double	s	out	Pointer to apparent power in volt-amperes.

Return

These functions do not return any arguments.

Description

These functions retrieve active power (P), reactive power (Q), RMS voltage (URMS), and RMS current (IRMS) from the internal data structure of the respective metering library. All quantities are scaled to the physical representation and returned in double floating point precision. The powers and RMS values are scaled by U_MAX and I_MAX.

$$P = P_Q * U\_MAX * I\_MAX \quad \textit{Eq. 46}$$

$$Q = Q_Q * U\_MAX * I\_MAX \quad \textit{Eq. 47}$$

$$I_{RMS} = IRMS_Q * I\_MAX \quad \textit{Eq. 48}$$

$$U_{RMS} = URMS_Q * U\_MAX \quad \textit{Eq. 49}$$

This figure shows block diagram of the METERLIB1PH_ReadResults() functions for reading non-billing quantities from the METERLIB1PH_DATA high-precision library internal data structure:

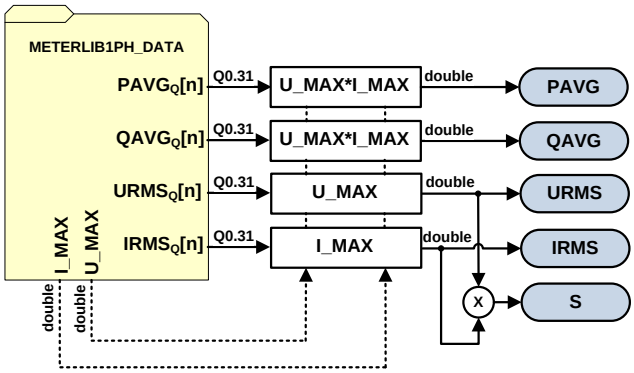


Figure 35. **METERLIB1PH_ReadResults function block diagram**

This figure shows block diagram of the METERLIBLP1PH_ReadResults() functions for reading non-billing quantities from the METERLIBLP1PH_DATA low-power library internal data structure:

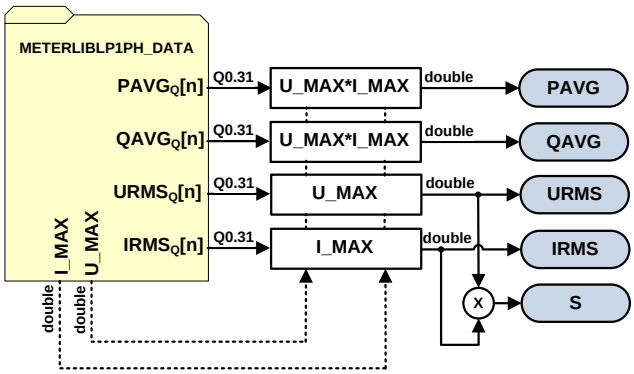


Figure 36. **METERLIBLP1PH_ReadResults function block diagram**

NOTE

All reading functions perform several calculations in double precision format. Due to this fact, and also for practical reasons, the reading functions shall be called in an interrupt which is intended to update non-billing information on the LCD. It is sufficient to update the LCD and call reading functions every 250 milliseconds or so. With such a low update rate the processor load caused by the reading functions will be almost negligible.

Performance

Function	Code size	Stack size	Clock cycles
METERLIB1PH_ReadResults	240	32	2557
METERLIB2PH_ReadResultsPh1 METERLIB2PH_ReadResultsPh2	236	32	5083
METERLIB3PH_ReadResultsPh1 METERLIB3PH_ReadResultsPh2 METERLIB3PH_ReadResultsPh3	240	32	7605
METERLIBLP1PH_ReadResults	232	32	2526
METERLIBLP2PH_ReadResultsPh1 METERLIBLP2PH_ReadResultsPh2	232	32	5008
METERLIBLP3PH_ReadResultsPh1 METERLIBLP3PH_ReadResultsPh2 METERLIBLP3PH_ReadResultsPh3	232	32	7473

4.1.13. METERLIB_ReadURMS

These functions read RMS voltages from the internal data structure.

Syntax

```
#include "meterlib.h"
void METERLIB1PH_ReadURMS (tMETERLIB1PH_DATA *p, double *urms1);
void METERLIB2PH_ReadURMS (tMETERLIB2PH_DATA *p, double *urms1, double *urms2);
void METERLIB3PH_ReadURMS (tMETERLIB3PH_DATA *p, double *urms1, double *urms2, double
*urms3);

#include "meterliblp.h"
void METERLIBLP1PH_ReadURMS (tMETERLIBLP1PH_DATA *p, double *urms1);
void METERLIBLP2PH_ReadURMS (tMETERLIBLP2PH_DATA *p, double *urms1, double *urms2);
void METERLIBLP3PH_ReadURMS (tMETERLIBLP3PH_DATA *p, double *urms1, double *urms2, double
*urms3);
```

Arguments

Type	Name	Direction	Description
tMETERLIB1PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB1PH_DATA.
tMETERLIB2PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB2PH_DATA.
tMETERLIB3PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB3PH_DATA.
tMETERLIBLP1PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP1PH_DATA.
tMETERLIBLP2PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP2PH_DATA.
tMETERLIBLP3PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP2PH_DATA.
double	urms1	out	Pointer to phase 1 RMS voltage value in volts.
double	urms2	out	Pointer to phase 2 RMS voltage value in volts.
double	urms3	out	Pointer to phase 3 RMS voltage value in volts.

Return

These functions do not return any arguments.

Description

These functions retrieve RMS voltage (URMS) from the internal data structure of the respective metering library. The RMS voltages are scaled by U_MAX to the physical representation and returned in double floating point precision.

$$U_{RMS} = URMS_Q * U\_MAX \quad \textit{Eq. 50}$$

A block diagram of the METERLIB1PH_ReadURMS() function defined in the high-precision metering library is shown in this figure:

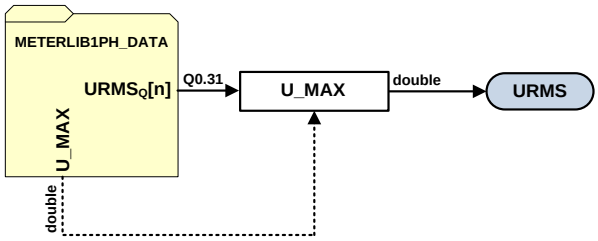


Figure 37. **METERLIB1PH_ReadURMS function block diagram**

A block diagram of the METERLIBLP1PH_ReadURMS() function defined in the low-power metering library is shown in this figure:

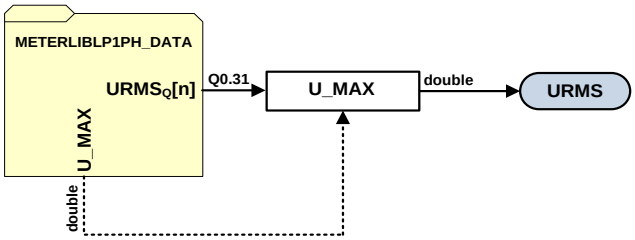


Figure 38. **METERLIBLP1PH_ReadURMS function block diagram**

Performance

Function	Code size	Stack size	Clock cycles
METERLIB1PH_ReadURMS	42	16	502
METERLIB2PH_ReadURMS	72	16	990
METERLIB3PH_ReadURMS	118	24	1464
METERLIBLP1PH_ReadURMS	40	16	489
METERLIBLP2PH_ReadURMS	72	16	965
METERLIBLP3PH_ReadURMS	112	24	1427

4.1.14. METERLIB_ReadIRMS

These functions read RMS currents from the internal data structure.

Syntax

```
#include "meterlib.h"
void METERLIB1PH_ReadIRMS (tMETERLIB1PH_DATA *p, double *irms1);
void METERLIB2PH_ReadIRMS (tMETERLIB2PH_DATA *p, double *irms1, double *irms2);
void METERLIB3PH_ReadIRMS (tMETERLIB3PH_DATA *p, double *irms1, double *irms2, double
*irms3);

#include "meterliblp.h"
void METERLIBLP1PH_ReadIRMS (tMETERLIBLP1PH_DATA *p, double *irms1);
void METERLIBLP2PH_ReadIRMS (tMETERLIBLP2PH_DATA *p, double *irms1, double *irms2);
void METERLIBLP3PH_ReadIRMS (tMETERLIBLP3PH_DATA *p, double *irms1, double *irms2, double
*irms3);
```

Arguments

Type	Name	Direction	Description
tMETERLIB1PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB1PH_DATA.
tMETERLIB2PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB2PH_DATA.
tMETERLIB3PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB3PH_DATA.
tMETERLIBLP1PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP1PH_DATA.
tMETERLIBLP2PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP2PH_DATA.
tMETERLIBLP3PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP2PH_DATA.
double	irms1	out	Pointer to phase 1 RMS current value in amperes.
double	irms2	out	Pointer to phase 2 RMS current value in amperes.
double	irms3	out	Pointer to phase 3 RMS current value in amperes.

Return

These functions do not return any arguments.

Description

These functions retrieve RMS current (IRMS) from the internal data structure. The RMS currents are scaled by I_MAX to the physical representation and returned in double floating point precision.

$$I_{RMS} = IRMS_Q * I\_MAX \qquad \textit{Eq. 51}$$

A block diagram of the METERLIB1PH_ReadIRMS() function defined in the high-precision metering library is shown in this figure:

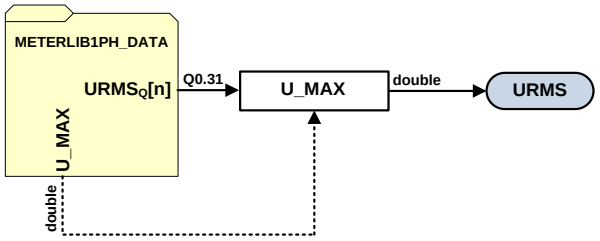


Figure 39. **METERLIB1PH_ReadIRMS function block diagram**

A block diagram of the METERLIBLP1PH_ReadIRMS() function defined in the low-power metering library is shown in this figure:

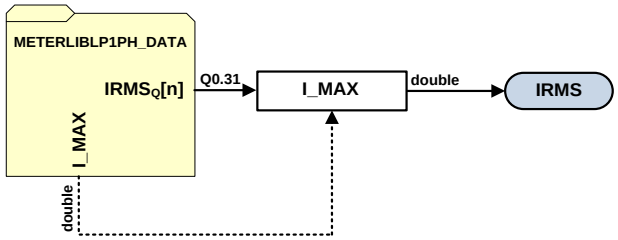


Figure 40. **METERLIBLP1PH_ReadIRMS function block diagram**

Performance

Function	Code size	Stack size	Clock cycles
METERLIB1PH_ReadIRMS	44	16	556
METERLIB2PH_ReadIRMS	76	16	1082
METERLIB3PH_ReadIRMS	124	24	1608
METERLIBLP1PH_ReadIRMS	42	16	505
METERLIBLP2PH_ReadIRMS	78	24	985
METERLIBLP3PH_ReadIRMS	110	24	1461

4.1.15. METERLIB_ReadPAVG

These functions read active powers from the internal data structure.

Syntax

```
#include "meterlib.h"
void METERLIB1PH_ReadPAVG (tMETERLIB1PH_DATA *p, double *pavg1);
void METERLIB2PH_ReadPAVG (tMETERLIB2PH_DATA *p, double *pavg1, double *pavg2);
void METERLIB3PH_ReadIAVG (tMETERLIB3PH_DATA *p, double *pavg1, double *pavg2, double
*pavg3);

#include "meterliblp.h"
void METERLIBLP1PH_ReadPAVG (tMETERLIBLP1PH_DATA *p, double *pavg1);
void METERLIBLP2PH_ReadPAVG (tMETERLIBLP2PH_DATA *p, double *pavg1, double *pavg2);
void METERLIBLP3PH_ReadIAVG (tMETERLIBLP3PH_DATA *p, double *pavg1, double *pavg2, double
*pavg3);
```

Arguments

Type	Name	Direction	Description
tMETERLIB1PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB1PH_DATA.
tMETERLIB2PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB2PH_DATA.
tMETERLIB3PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB3PH_DATA.
tMETERLIBLP1PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP1PH_DATA.
tMETERLIBLP2PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP2PH_DATA.
tMETERLIBLP3PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP2PH_DATA.
double	pavg1	out	Pointer to phase 1 active power in watts.
double	pavg2	out	Pointer to phase 2 active power in watts.
double	pavg3	out	Pointer to phase 3 active power in watts.

Return

These functions do not return any arguments.

Description

These functions retrieve active power (PAVG) from the internal data structure. The active powers are scaled by U_MAX and I_MAX to the physical representation and returned in double floating point precision.

$$P = P_Q * U\_MAX * I\_MAX \quad \textit{Eq. 52}$$

A block diagram of the METERLIB1PH_ReadPAVG() function defined in the high-precision metering library is shown in this figure:

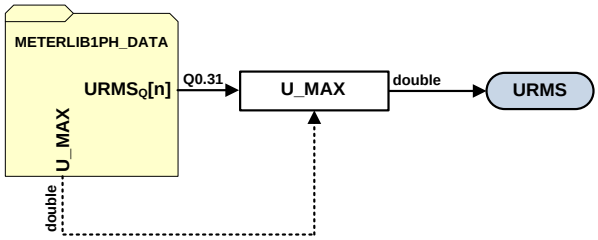


Figure 41. **METERLIB1PH_ReadPAVG function block diagram**

A block diagram of the METERLIBLP1PH_ReadPAVG() function defined in the low-power metering library is shown in this figure:

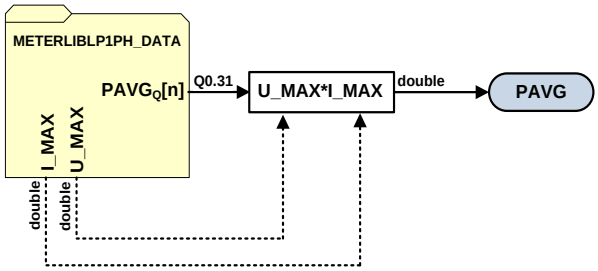


Figure 42. **METERLIBLP1PH_ReadPAVG function block diagram**

Performance

Function	Code size	Stack size	Clock cycles
METERLIB1PH_ReadPAVG	52	16	734
METERLIB2PH_ReadPAVG	96	16	1436
METERLIB3PH_ReadPAVG	156	24	2163
METERLIBLP1PH_ReadPAVG	50	16	722
METERLIBLP2PH_ReadPAVG	96	16	1414
METERLIBLP3PH_ReadPAVG	150	24	2127

4.1.16. METERLIB_ReadS

These functions read apparent powers from the internal data structure.

Syntax

```
#include "meterlib.h"
void METERLIB1PH_ReadS (tMETERLIB1PH_DATA *p, double *s1);
void METERLIB2PH_ReadS (tMETERLIB2PH_DATA *p, double *s1, double *s2);
void METERLIB3PH_ReadS (tMETERLIB3PH_DATA *p, double *s1, double *s2, double *s3);

#include "meterliblp.h"
void METERLIBLP1PH_ReadS (tMETERLIBLP1PH_DATA *p, double *s1);
void METERLIBLP2PH_ReadS (tMETERLIBLP2PH_DATA *p, double *s1, double *s2);
void METERLIBLP3PH_ReadS (tMETERLIBLP3PH_DATA *p, double *s1, double *s2, double *s3);
```

Arguments

Type	Name	Direction	Description
tMETERLIB1PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB1PH_DATA.
tMETERLIB2PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB2PH_DATA.
tMETERLIB3PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB3PH_DATA.
tMETERLIBLP1PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP1PH_DATA.
tMETERLIBLP2PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP2PH_DATA.
tMETERLIBLP3PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP2PH_DATA.
double	s1	out	Pointer to phase 1 apparent power in volt-amperes.
double	s2	out	Pointer to phase 2 apparent power in volt-amperes.
double	s3	out	Pointer to phase 3 apparent power in volt-amperes.

Return

These functions do not return any arguments.

Description

These functions retrieve apparent power (S) from the internal data structure. The apparent powers are scaled by U_MAX and I_MAX to the physical representation and returned in double floating point precision.

$$S = IRMS_Q * URMS_Q * U\_MAX * I\_MAX \qquad \textit{Eq. 53}$$

A block diagram of the METERLIB1PH_ReadS() function defined in the high-precision metering library is shown in this figure:

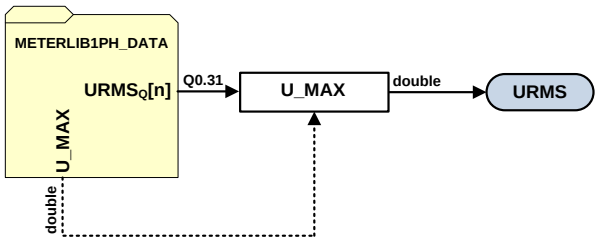


Figure 43. **METERLIB1PH_ReadS function block diagram**

A block diagram of the METERLIBLP1PH_ReadS() function defined in the low-power metering library is shown in this figure:

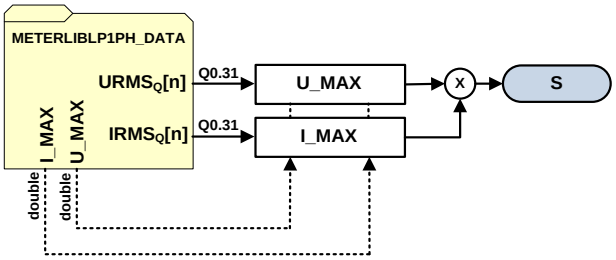


Figure 44. **METERLIBLP1PH_ReadS function block diagram**

Performance

Function	Code size	Stack size	Clock cycles
METERLIB1PH_ReadS	64	16	787
METERLIB2PH_ReadS	120	16	1552
METERLIB3PH_ReadS	192	24	2320
METERLIBLP1PH_ReadS	60	16	777
METERLIBLP2PH_ReadS	116	16	1530
METERLIBLP3PH_ReadS	180	24	2304

4.1.17. METERLIB_KWH_PD

This macro returns a fine delay of the active energy pulse output transition.

Syntax

```
#include "meterlib.h"
#define METERLIB_KWH_PD(p)        (frac16)(p)->wh.puDly

#include "meterliblp.h"
#define METERLIBLP_KWH_PD(p)      (frac16)(p)->wh.puDly
```

Arguments

Type	Name	Direction	Description
tMETERLIB1PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB1PH_DATA.
tMETERLIB2PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB2PH_DATA.
tMETERLIB3PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB3PH_DATA.
tMETERLIBLP1PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP1PH_DATA.
tMETERLIBLP2PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP2PH_DATA.
tMETERLIBLP3PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP2PH_DATA.

Return

This macro returns a fine delay in range from 0x0000 to 0x7fff in case of the active energy pulse output transition. The fine delay is scaled to the calculation step (1/KWH_CALC_FREQ). If active energy pulse output doesn't change then macro returns -1 (0x8000).

4.1.18. METERLIB_KVARH_PD

This macro returns a fine delay of the reactive energy pulse output transition.

Syntax

```
#include "meterlib.h"
#define METERLIB_KVARH_PD(p)      (frac16)(p)->varh.puDly

#include "meterliblp.h"
#define METERLIBLP_KVARH_PD(p)    (frac16)(p)->varh.puDly
```

Arguments

Type	Name	Direction	Description
tMETERLIB1PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB1PH_DATA.
tMETERLIB2PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB2PH_DATA.
tMETERLIB3PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB3PH_DATA.
tMETERLIBLP1PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP1PH_DATA.
tMETERLIBLP2PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP2PH_DATA.
tMETERLIBLP3PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP2PH_DATA.

Return

This macro returns a fine delay in range from 0x0000 to 0x7fff in case of the reactive energy pulse output transition. The fine delay is scaled to the calculation step (1/KVARH_CALC_FREQ). If reactive energy pulse output doesn't change then macro returns -1 (0x8000).

4.1.19. METERLIB_KWH_PS

This macro returns a raw state of the active energy pulse output.

Syntax

```
#include "meterlib.h"
#define METERLIB_KWH_PS(p)        (frac16)(p)->wh.puOut

#include "meterliblp.h"
#define METERLIBLP_KWH_PS(p)      (frac16)(p)->wh.puOut
```

Arguments

Type	Name	Direction	Description
tMETERLIB1PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB1PH_DATA.
tMETERLIB2PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB2PH_DATA.
tMETERLIB3PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB3PH_DATA.
tMETERLIBLP1PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP1PH_DATA.
tMETERLIBLP2PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP2PH_DATA.
tMETERLIBLP3PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP2PH_DATA.

Return

This macro returns a raw state of the active energy pulse output.

4.1.20. METERLIB_KVARH_PS

This macro returns a raw state of the reactive energy pulse output.

Syntax

```
#include "meterlib.h"
#define METERLIB_KVARH_PS(p)      (frac16)(p)->varh.puOut

#include "meterliblp.h"
#define METERLIBLP_KVARH_PS(p)    (frac16)(p)->varh.puOut
```

Arguments

Type	Name	Direction	Description
tMETERLIB1PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB1PH_DATA.
tMETERLIB2PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB2PH_DATA.
tMETERLIB3PH_DATA	p	in/out	Pointer to high-precision library data structure tMETERLIB3PH_DATA.
tMETERLIBLP1PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP1PH_DATA.
tMETERLIBLP2PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP2PH_DATA.
tMETERLIBLP3PH_DATA	p	in/out	Pointer to low-power library data structure tMETERLIBLP2PH_DATA.

Return

This macro returns a raw state of the reactive energy pulse output.

4.1.21. METERLIB_KWH_PR

This macro converts imp/kWh number to pulse output resolution required by metering libraries.

Syntax

```
#include "meterlib.h"
#define METERLIB_KWH_PR(x)        FRAC48(((5e2/(x))/(U_MAX*I_MAX/3600/KWH_CALC_FREQ)))

#include "meterliblp.h"
#define METERLIBLP_KWH_PR(x)      FRAC32(((5e2/(x))/(U_MAX*I_MAX/3600/KWH_CALC_FREQ)))
```

Arguments

Type	Name	Direction	Description
int	x	in	User defined imp/kWh number.

Return

This macro returns an active energy pulse output resolution.

4.1.22. METERLIB_KVARH_PR

This macro converts imp/kVARh number to pulse output resolution required by metering libraries.

Syntax

```
#include "meterlib.h"
#define METERLIB_KVARH_PR(x)      FRAC48(((5e2/(x))/(U_MAX*I_MAX/3600/KVARH_CALC_FREQ)))
```

```
#include "meterliblp.h"
#define METERLIBLP_KVARH_PR(x)    FRAC32(((5e2/(x))/(U_MAX*I_MAX/3600/KVARH_CALC_FREQ)))
```

Arguments

Type	Name	Direction	Description
int	x	in	User defined imp/kVARh number.

Return

This macro returns a reactive energy pulse output resolution.

4.1.23. METERLIB_DEG2SH

This macro converts U-I phase shift in degrees to a 16-bit fractional number with resolution of (fn*360/(32768*KWH_CALC_FREQ)) degrees.

Syntax

```
#include "meterlib.h"
#define METERLIB_DEG2SH(x,fn)     FRAC16((float)(x)*KWH_CALC_FREQ/((float)fn*360.0))

#include "meterliblp.h"
#define METERLIBLP_DEG2SH(x,fn)   FRAC16((float)(x)*KWH_CALC_FREQ/((float)fn*360.0))
```

Arguments

Type	Name	Direction	Description
double	x	in	U-I phase shift in degrees.
double	fn	in	Nominal frequency in Hz.

Return

This macro returns converted U-I phase shift in a 16-bit fractional representation.

4.1.24. METERLIB_RAD2SH

This macro converts U-I phase shift in radians to a 16-bit fractional number with resolution of (fn*2*Pi/(32768*KWH_CALC_FREQ)) radians.

Syntax

```
#include "meterlib.h"
#define METERLIB_RAD2SH(x,fn)
       FRAC16((float)(x)*KWH_CALC_FREQ/((float)fn*2.0*3.14159265358979323846))

#include "meterliblp.h"
#define METERLIBLP_RAD2SH(x,fn)
       FRAC16((float)(x)*KWH_CALC_FREQ/((float)fn*2.0*3.14159265358979323846))
```

Arguments

Type	Name	Direction	Description
double	x	in	U-I phase shift in radians.
double	fn	in	Nominal frequency in Hz.

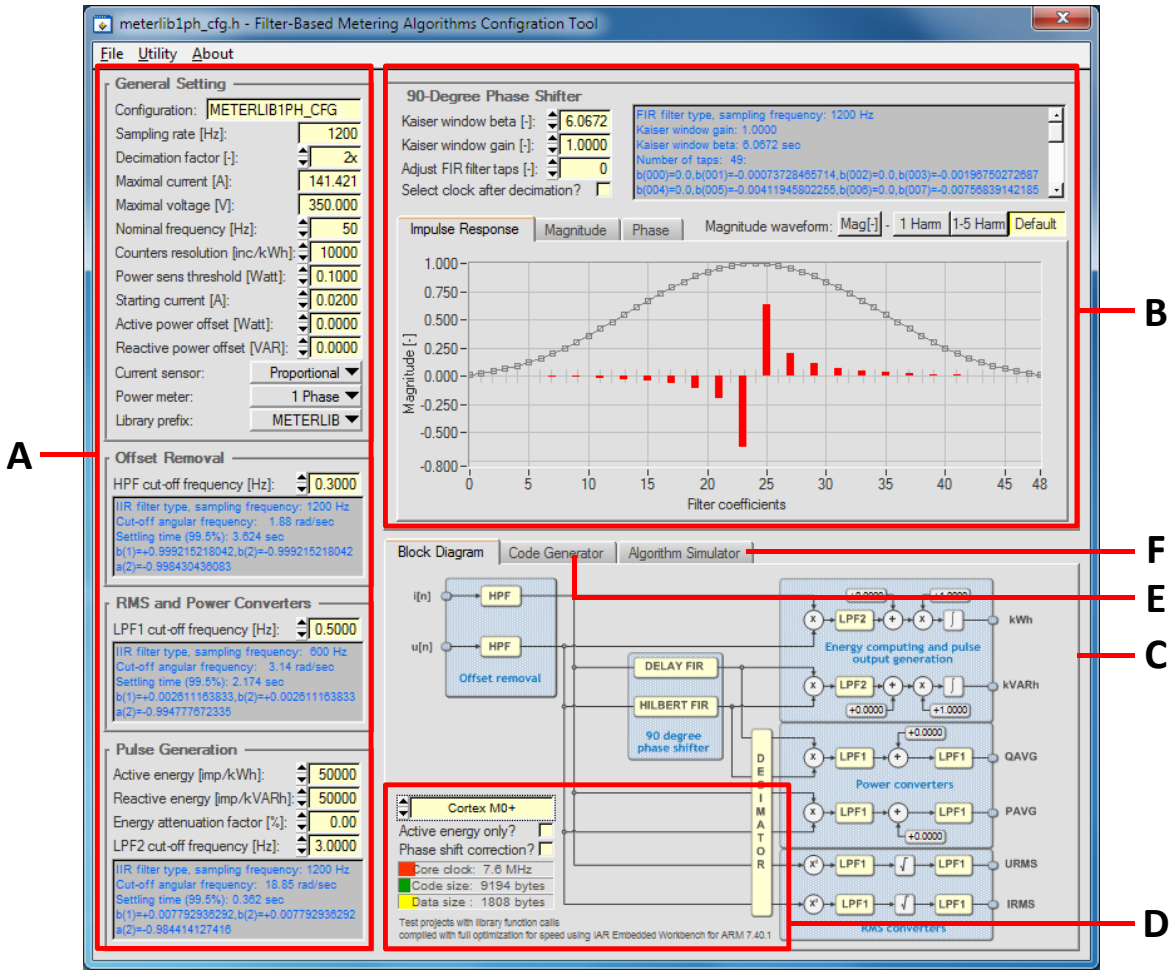
Return

This macro returns converted U-I phase shift in 16-bit fractional representation.

4.2. Configuration tool

This section describes the configuration tool workspace and all its features. The configuration tool is primarily intended to tune the filters to match the required performance and to generate the C-header file that contains the source code with the parameters describing the behavior of the Filter-Based Metering Algorithm. It automatizes the procedure of the algorithm setup and optimization, while providing a rough estimate of the required computational load. The tool generates the C-code for configuring of 32-bit metering algorithms and supports one-phase, two-phase (Form-12S), and three-phase power meter topologies.

It is recommended to be familiar with the workspace, because this is where your time is spent when using the configuration tool to design a file with the parameters for the Filter-Based Metering Algorithm (see Figure 45).



A – General and configuration panel, B – 90-degree phase shifter configuration panel, C – Block diagram panel, D – Performance estimator panel, E – Code generator panel, F – Algorithm simulator panel

Figure 45. **Configuration tool**

The configuration tool comprises six configuration and visualization panels: the general configuration panel, the 90-degree phase shifter configuration panel, the block diagram panel, the performance estimator panel, the code generator panel, and the algorithm simulator panel.

4.2.1. General configuration panel

The general configuration panel is where you configure the basic parameters of your power meter application.

Table 9. **Parameters of the general configuration panel**

Parameter	Unit	Default	Min.	Max.	Description
Configuration	—	—	—	—	The name of the structure containing the configuration parameters.
Sampling rate	Hz	1200	1000	12000	The filter sampling rate.
Decimation factor	—	1	1	10	The decimator factor for the computation of non-billing quantities.
Maximal current	A	141.421	—	—	The peak current scaled to the full ADC input range.
Maximal voltage	V	395.980	—	—	The peak voltage scaled to the full ADC input range.
Nominal frequency	Hz	50	50	60	The nominal frequency of the power meter.
Counters resolution	Inc/kWh	10 000	100	100 000	The resolution of the active and reactive energy counters.
Power sense threshold	W	0.5	0.0	2.0	The power threshold for clearing the RMS current and power values. This threshold does not influence the accumulation of the active and reactive energies.
Starting current	A	0.02	0.0	1.0	The RMS current threshold for clearing the RMS current and power values. If the RMS current in the phase is below this threshold, then the active and reactive energies also do not accumulate.
Active power offset[4]	W	0.0	0.0	9.9999	Used for zero-load active power residue cancellation.
Reactive power offset4	VAR	0.0	0.0	9.9999	Used for zero-load reactive power residue cancellation.
Current sensor	—	Proportional	—	—	The sensor output characteristic: proportional (current transformer or shunt resistor), derivative (Rogowski coil).
Power meter	—	three-phase	—	—	Number of phases: one-phase, two-phase, or three-phase.
Library prefix	—	METERLIB	—	—	METERLIB: high-precision library, METERLIBLP: low-power library.

Check and modify the default settings of the offset-removal configuration panel. The offset-removal block uses the high-pass IIR filters to remove the DC offset from the measured waveforms. By default,

[4] Both the “active” and “reactive” power offsets are intended to compensate for sensors and PCB cross-talks. Use them whenever a library outputs a non-zero power measurement at a no-load condition. Use the following steps to set up the power offsets in the configuration tool: 1) calibrate the power meter with the “power sense threshold”, “starting current threshold” and “power offsets” parameters set to zero, 2) put the calibrated power meter under a no-load test condition and monitor the measured “active” and “reactive” powers until a steady state, 3) write the steady state “active” and “reactive” powers with negative signs into the respective “power offset” controls, 4) update the library configuration header file, recompile the project, and download the application into your power meter. After the recalibration, the accuracy of the power meter in lower currents improves.

the cut-off frequency of these high-pass IIR filters is set to 0.3 Hz. This default setting works for most applications.

Table 10. **Parameters of the offset-removal configuration panel**

Parameter	Unit	Default	Min.	Max.	Description
HPF cut-off frequency	Hz	0.3	0.1	29.9	High-pass filter cut-off frequency.

The RMS and the power converter blocks are represented by a cascade of two low-pass IIR filters. The characteristics of these filters are defined by the cut-off frequency setting in the RMS and power converters configuration panel. By default, the cut-off frequency of the low-pass IIR filter is set to 0.5 Hz.

Table 11. **Parameters of the RMS and power converters configuration panel**

Parameter	Unit	Default	Min.	Max.	Description
LPF1 cut-off frequency	Hz	0.5	0.2	29.9	Low-pass filter cut-off frequency.

Finally, check and adjust the setting of the pulse output generation. The configuration tool and the metering libraries support the generation of two pulse outputs. The pulse outputs are used to calibrate the measurement accuracy of the active and reactive energies using a reference meter. This very popular method of calibration uses a power source, a meter pulse output, and an external reference meter to determine the required compensation. Set the parameters of the meter pulse output (such as the number of pulses per energy quanta and the smoothing factor) in the pulse generation configuration panel.

Table 12. **Parameters of the pulse generation configuration panel**

Parameter	Unit	Default	Min.	Max.	Description
Active energy	Imp/kWh	50 000	100	5e6	Number of pulses generated by the power meter for one kWh.
Reactive energy	Imp/kVARh	50 000	100	5e6	Number of pulses generated by the power meter for one kVARh.
Energy attenuation factor	%	0.0	0	10.0	Attenuation of the energy calculation paths. Used mainly with the low-power libraries to improve the balance between the energy and non-billing computation paths.
LPF2 cut-off frequency	Hz	3.0	2.0	5.0	Pulse output ripple cancellation low-pass filter cut-off frequency.

4.2.2. 90-degree phase shifter panel

The 90-degree phase shifter block is represented by the Hilbert FIR filter. Modify the Hilbert FIR filter characteristics using the 90-degree phase shifter configuration panel. The default setting of this complex FIR filter is computed with the aim to achieve a unity gain of the 90-degree shifted output waveforms in the frequency bandwidth above the nominal frequency of 50 Hz (60 Hz), and below the half of the sampling frequency.

Table 13. **Parameters of the 90-degree phase shifter configuration panel**

Parameter	Unit	Default	Min.	Max.	Description
Kaiser window beta	—	6.0672	0.0	9.9999	Kaiser window coefficient.
Kaiser window gain	—	1.0	0.5	2.0	Kaiser window gain.
Adjust FIR filter taps	—	0	100	100	Increase or lower the number of filter taps manually.
Select clock after decimation	—	—	—	—	Hilbert FIR filter computation frequency.

To make the setting of the Hilbert FIR filter easier, leverage the “Knob Control” object (a graphical panel for a precise adjustment of parameters). This panel is hidden by default, and you can activate it using the CTRL+K keyboard shortcut. For more information about the “Knob Control” usage, see Using the configuration tool.

4.2.3. Block diagram panel

The procedure for computing of metering quantities depends on the library configurations.
The configuration tool is designed to track your changes within the library configurations, analyze their impact on the computation algorithm, and to display the most up-to-date computation block diagram, together with the estimated computational load of the selected MCU core architecture.

4.2.4. Performance estimator panel

This panel shows the estimated core clock frequency (in MHz) that is required to compute the metering algorithms. The core clock frequency is computed for the selected MCU core architecture and for specific library configurations.

4.2.5. Code generator panel

This panel shows a real-time preview of the data structure containing the most up-to-date library configuration (see the following figure). For more information about integrating the library configuration structure into your project, see Configuration tool.

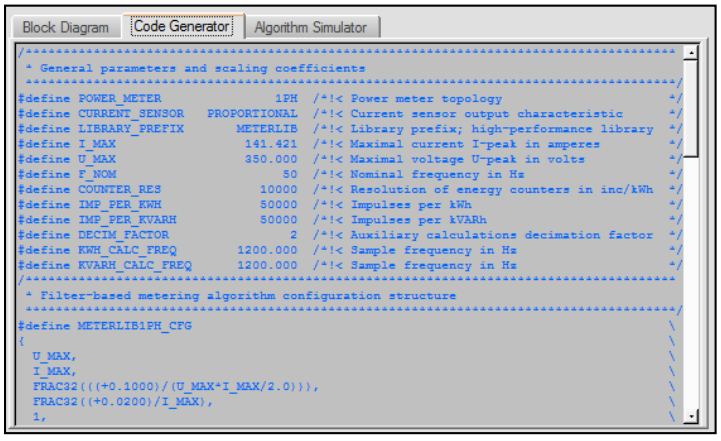


Figure 46. **Code generator visualization panel**

Start working with the configuration tool either by creating a new library configuration data structure or by opening an existing one. Store the library configuration data structure in a C header file if you want to make a reference to it from within your project. This figure shows the menu of all commands for handling the library configuration files.

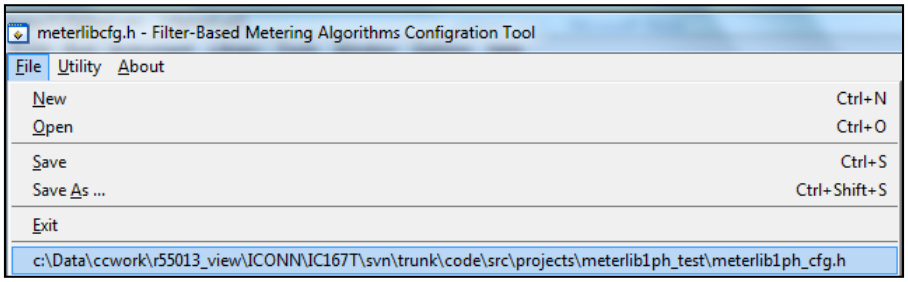
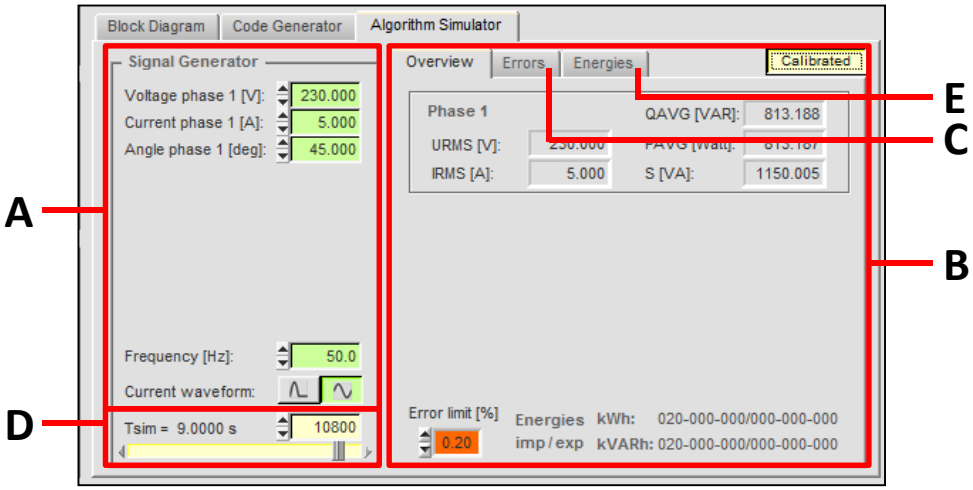


Figure 47. **Handling library configuration structure**

4.2.6. Algorithm simulator panel

The algorithm simulator panel is by default hidden by the configuration tool. Show/hide it using the CTRL+A keyboard shortcut. The algorithm simulator panel is capable of analyzing the performance of the metering library preset using your specific configuration. The algorithm simulator panel comprises four blocks: A—signal generator block, B—overview visualization block, C—errors visualization block, and D—simulation time slider:



A – Signal generator block, B – Overview visualization block, C – Errors visualization block, D – Simulation time slider, E – Energies visualization block

Figure 48. **Algorithm simulator panel**

The algorithm simulator panel is updated after each change in the library configuration. It simulates the dynamic response of the metering library based on the user configuration during the first 10 seconds of operation. Use the simulation time slider (D) to select the computation step of interest. For the selected computation step, the overview (B), errors (C), and energies visualization blocks are updated with the actual simulated values.

NOTE

Use the “Knob Control” pop-up panel to set up all parameters of the signal generator block (A), and to control the simulation time slider (D).
Bring up the “Knob Control” pop-up panel using the CTRL+K keyboard shortcut.

On the contrary to the physical quantities displayed in the overview visualization panel, the errors visualization panel displays the deviations of quantities from the steady-state values in percentages:

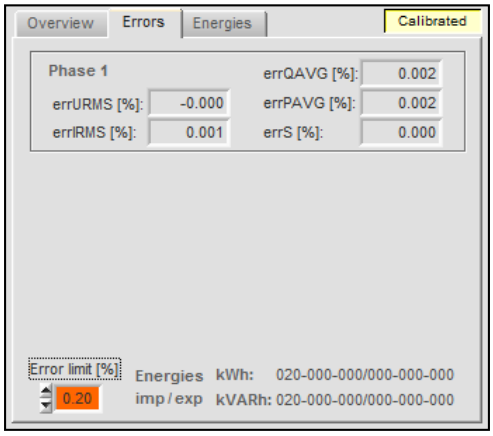


Figure 49. **Errors visualization panel**

The error limit [%] control is visible in both the overview (B) and error (C) visualization blocks. It enables you to preset the error margin as a percentage. When the simulated quantity fits into the specified error margin, the background color of the respective control is grayed out. When the deviation of the simulated quantity for a given computation step doesn’t fit into the specified error margin, the background color of the respective control turns red.

You can quickly check whether your selected and configured library (high-precision or low-power) is accurate enough by simply predefining the error limit [%] of your interest and looking at the background colors of the simulated quantities, while changing the computation step using the simulation time slider (D). You can also find a computation step where all non-billing quantities settle and where their accuracies start to fit into the predefined error margin [%].

The simulation engine (integrated within the configuration tool) uses metering libraries (including data representation) that are compiled for the target MCU platforms. The simulations performed by the configuration tool directly on the PC provide bit-accurate results and the responses to the input waveforms generated by the signal generator block (A).

Toying with the simulator gives you a good insight into the algorithm performance and accuracy. Test the behavior of metering algorithms using the algorithm simulator panel (e.g., when the input current waveform is below the starting current threshold). Investigate the difference in accuracies of the high-precision and low-power metering libraries for the given input voltage and current waveforms within several clicks.

The following figure shows the panel for displaying the results of energy counting. This panel shows the values of all energy counters supported by the library.

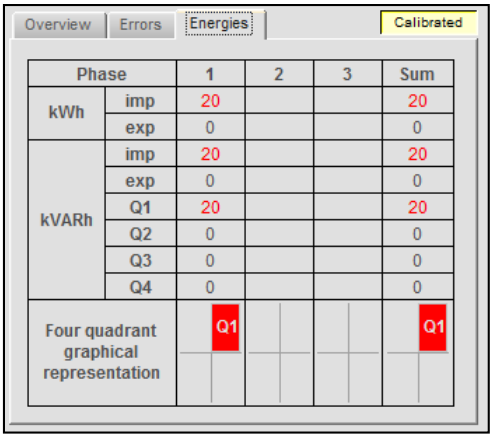


Figure 50. **Energies visualization panel**

The filter-based metering libraries track the changes of the active and reactive energies independently for each phase. The metering libraries compute the imported and exported active and reactive energies, as well as the reactive energies in four quadrants. The energy counters are accumulated at each calculation step with the resolution defined by the value in the "Counters resolution" dialog box.
The only energy counters updated by the metering library are those based on the location of the apparent power phase phasor (S) within the distribution diagram:

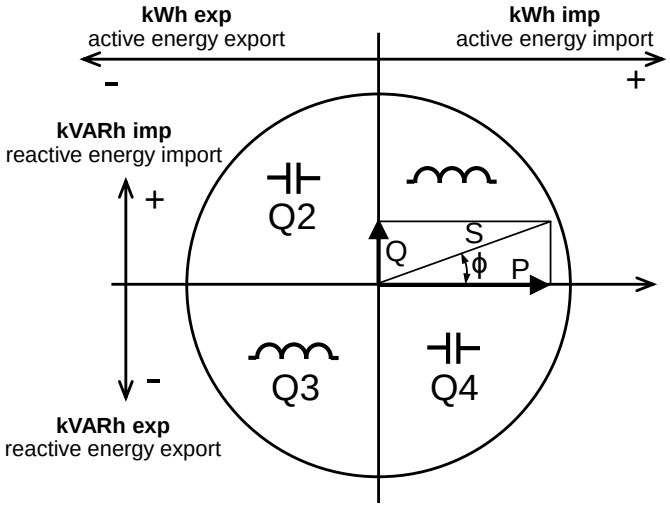


Figure 51. **Energies distribution diagram**

This table summarizes the energy counters supported by the metering library and their updating based on the apparent power phase phasor (S) location within the distribution diagram:

Table 14. **Dependencies of updated energy counters update**

Counter name	Counter description	Apparent power phase phasor sector
kWh imp	Active energy import	Q1 or Q4
kWh exp	Active energy export	Q2 or Q3
kVARh imp	Reactive energy import	Q1 or Q2
kVARh exp	Reactive energy export	Q3 or Q4
kVARh Qn	Reactive energy in Qn quadrant, n=1,2,3, and 4	Qn

The following figure demonstrates the performance of the phase sequence detection. The simulator returns the phase sequence for a three-phase system. The phase sequence is indicated by the color of the decoration panel in the “Overview” and “Errors” visualization panels. The phase sequence 1-2-3 (clockwise rotation) is indicated by the green color and the phase sequence 3-2-1 (counter-clockwise rotation) by the dark-blue color, respectively. The hidden decoration panel in the three-phase system means that the phase sequence cannot be decoded properly.

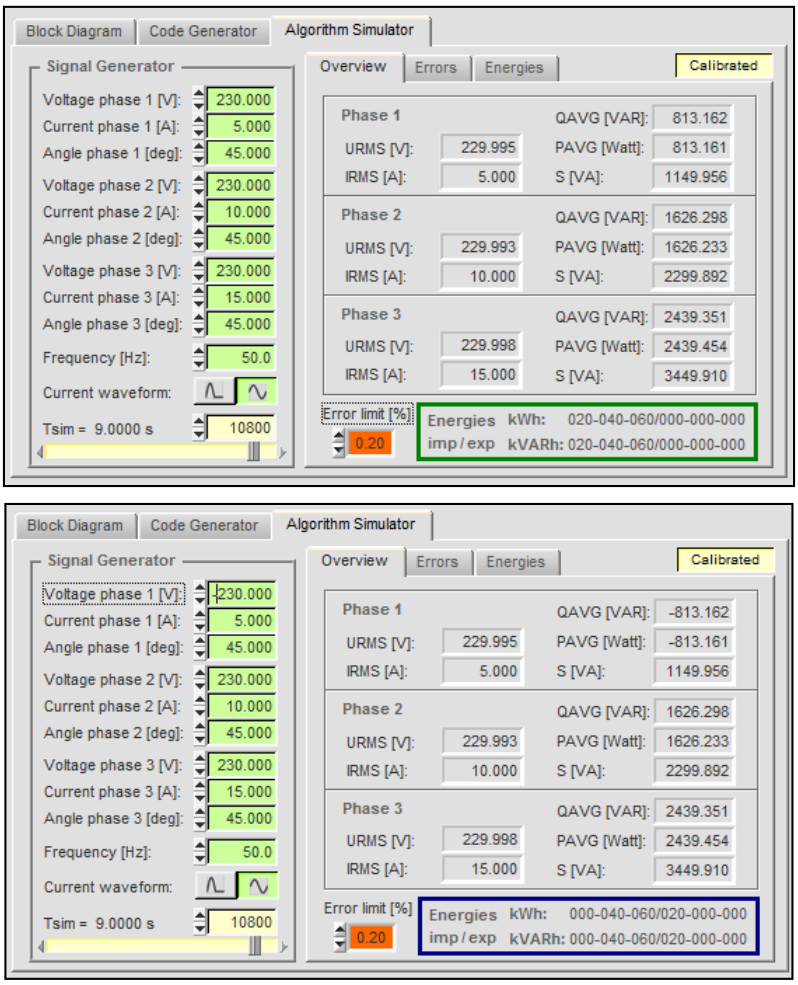


Figure 52. **Phase sequence detection**

NOTE

Select either the calibrated or the un-calibrated simulator output to evaluate the accuracy of Filter-Based Metering Algorithms after or before the calibration. The graphical configuration tool selects the calibrated output of the simulator by default. In the calibrated mode, the simulator shows the values computed by the algorithms with corrections for the high-pass filter amplitude characteristics at the nominal frequency. As opposed to the calibrated mode, you must select the un-calibrated mode manually, and the simulator does not compensate for the filter characteristics in this mode. The un-calibrated simulator output is less accurate than the calibrated simulator output.

4.2.7. Using the configuration tool

The example described in this section shows the procedure of configuring the Filter-Based Metering Algorithm for a typical one-phase power meter application. The application runs on the MKM34Z128 32-bit Kinetis-M MCU. This device is based on the ARM Cortex-M0+ core and integrated with a powerful 24-bit AFE. The 32-bit core, powerful 4-channel 24-bit AFE, and the additional 16-bit SAR ADC make this family of MCUs ideal for one-phase, two-phase (Form-12S), and three-phase power-meter applications.

These information about the target hardware platform, application firmware, and power meter features and capabilities are needed for the configuration:

- Hardware platform:
  - Power meter type: one-phase
  - Mains frequency: 50 Hz
  - Resolution of the active energy counter: 0.1 Wh
  - Resolution of the reactive energy counter: 0.1 VARh
  - Current scaling (I_MAX): 141.42 A
  - Voltage scaling (U_MAX): 325.27 V
- Application firmware:
  - Update rate for billing quantities (decimated from the 6-kHz AFE output rate): 1200 Hz
  - Decimation ratio (update rate) for non-billing quantities: ↓2 (600 Hz)
  - Power sensitivity threshold (for zeroing of non-billing quantities): 0.5 W
- Power meter features:
  - Active energy pulse output rate: 50 000 imp/kWh
  - Reactive energy pulse output rate: 50 000 imp/kVARh
  - Active and reactive energy accuracies: from 49 Hz to 250 Hz (±0.5 %)
  - Starting current threshold (according to IEC50470-3): 20 mA

Initially, the parameters describing the hardware platform and the application firmware are entered into the respective dialogue boxes of the general setting panel (see Figure 53). The name of the configuration structure containing all the algorithm configurations is stored in the *meterlib1ph_cfg.h* file, and referred to in the application code as “METERLIB1PH_CFG”.

NOTE

The only limitation in selecting the update rate for non-billing quantities is the signal bandwidth required for their calculation, and it must be at least 1200 Hz (or higher). The configuration tool enables you to enter an integer number as the decimation ratio to calculate the non-billing quantities. Implement this lower update rate into the software by skipping the non-billing calculations for a given number of times.

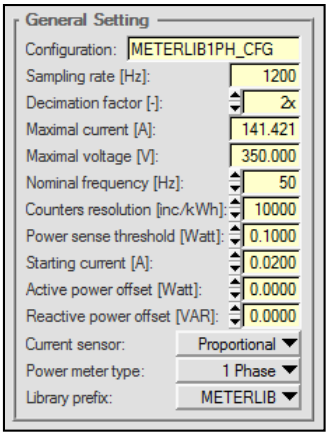


Figure 53. **General Setting**

Set the parameter of the offset removal block, as shown in the following figure. This block is represented by the high-pass first-order Butterworth filter, whose cut-off frequency must be chosen appropriately. It is recommended to use the default setting of 0.3 Hz, which guarantees an effective offset removal. Set the high-pass filter cut-off frequency in the range from 0.1 Hz to 5.0 Hz.

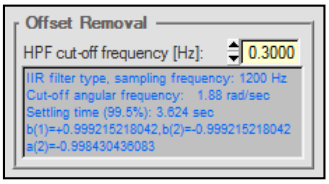


Figure 54. **Offset Removal**

The RMS and power converters comprising the low-pass first-order Butterworth filters are configured. This configuration is very straightforward, and requires you to select the filter cut-off frequency in the range from 0.3 Hz to 5.0 Hz. Using the default cut-off frequency of 0.5 Hz is recommended, unless you need a faster or smoother dynamic response.

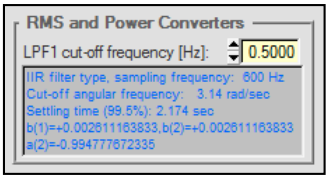


Figure 55. **RMS and Power Converters**

You must understand the default setting of the 90-degree phase shifter. Setting this block, which represents the Hilbert FIR filter with numerous coefficients to set up, is almost impossible without the configuration tool and/or high-level simulation tools (such as Matlab/Simulink). The configuration tool provides an intuitive way to parameterize this block (similar to operating an oscilloscope) when using the “Knob Control” panel. According to the technical requirements, the accuracy of the reactive energy must be ±0.5 % (in the frequency range from 49 Hz to 250 Hz).

The following figure shows the default setting of the block and the magnitude response of the FIR Hilbert filter. It is evident that the accuracy of the reactive energy resulting from the default setting is in the range of ±0.1 % (in the given frequency range).

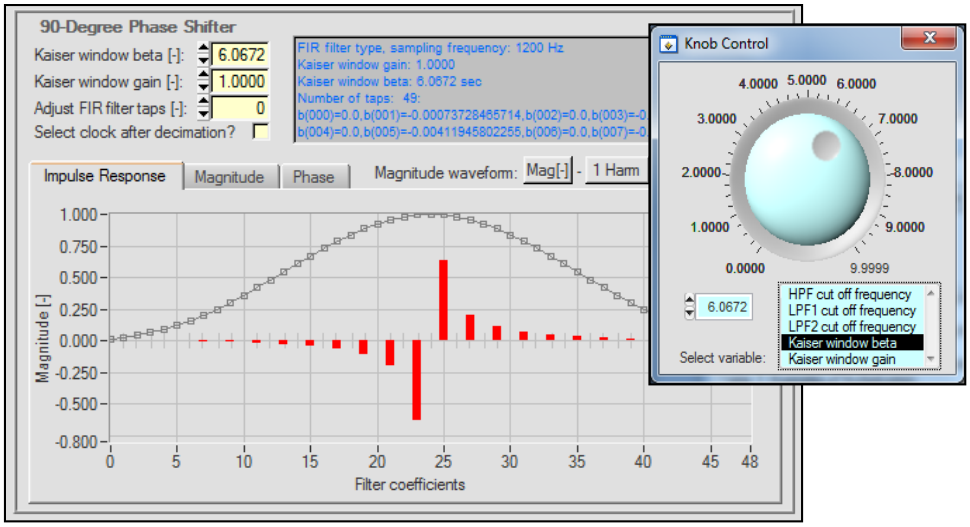


Figure 56. **Default setting of the 90-degree phase shifter**

The default Hilbert FIR filter has a length of 49 taps for the given sampling rate of 1200 Hz, and the mains frequency of 50 Hz. The group delay of the filter is 20 ms. Optimize the filter length, beta, and gain parameters of the Kaiser Window manually (using the "Knob Control" pop-up panel) to further lower the computational requirements.

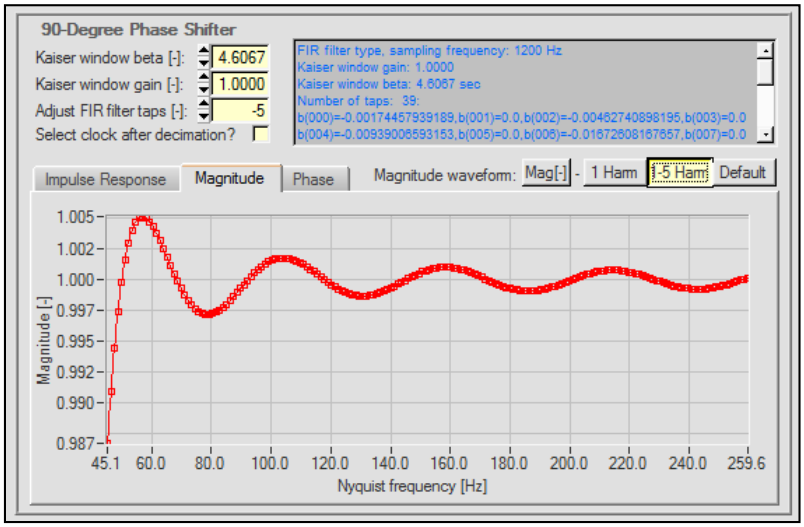


Figure 57. **Optimal setting of the 90-degree phase shifter**

Figure 57 shows the magnitude response of the optimized FIR Hilbert filter. The length of the filter is 39 taps, and its group delay is 15.83 ms. After the optimization, the computational requirements of the FIR Hilbert filter drops by approximately 20 %. The magnitude response of the filter fits into the required accuracy class of ±0.5 %.

NOTE

Even the ±0.5 % accuracy target for the reactive energy is somewhat high. The most demanding requirement for the reactive energy is given by the IEC 62053-23 international standard, which defines the accuracy of reactive energy measurement of ±2.0 %. Such accuracy can be achieved

with a 29-tap FIR filter with just 60 % of the computational performance, when compared to the default setting.

Configure the active and reactive pulse outputs and their low-pass filters. This is performed in several dialogue boxes, as shown in the following figure. To improve the jitter of the pulse output, the low-pass first-order Butterworth filter is used to filter out the energy ripples. It is recommended to use the default filter cut-off frequency of 3 Hz, which achieves the attenuation of the ripple energy by 33.3 dB in a relatively short time (0.281 s). The active and reactive energy pulse output numbers are set to 50 000 imp/kWh and 50 000 imp/kVARh, respectively.

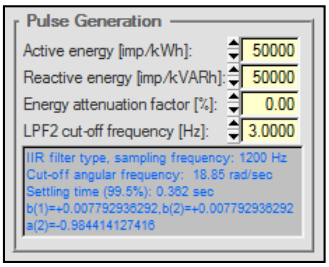


Figure 58. **Pulse generation**

After the configuration is complete, save it to the hard drive. An example of the configuration file generated by the configuration tool for the one-phase power meter application is shown in Appendix A.

NOTE

The configuration is stored in a C header file in the format of initialization data for the configuration structure of the particular power meter type. The configuration structures for all supported power meter topologies are defined in the *meterlib.h* header file.

The configuration file is an essential part of the power meter firmware application. A simple test application that includes a configuration file is shown in Appendix B. For the sake of simplicity, this application does not measure the phase voltage and phase current samples, but emulates these signals by software.

NOTE

Open the configuration file using the configuration tool for printing and parameter adjustment. For example, the configuration file specific to the one-phase power meter can be easily modified and used as the base for generating two-phase (Form-12S) and/or three-phase power meter configurations (and vice-versa). After making changes in the configuration file, recompile the firmware application and rebuild the whole project. Upload the new firmware code to the MCU and test the new algorithm configuration. Repeat the above steps until the algorithm matches the required performance.

The performance of the metering library was thoroughly tested. Real tests were carried out on the one-phase power meter reference designs. The results of the performance testing are described in the following section.

5. Accuracy and performance testing

The performance of the metering library was tested on the one-phase Kinetis M power meter reference design [12]. The MKM34Z128 device (32-bit Kinetis M MCU) at the heart of the reference design is based on the ARM Cortex-M0+ core. This efficient processor core, with support for 32-bit mathematics, enables fast execution of the Filter-Based Metering Algorithm.

Table 15. **Single phase KM3x power meter specification**

Type of meter	Single phase residential
Type of measurement	4-quadrant
Metering algorithm	Filter-based
Precision (accuracy)	IEC50470-3 class C, 0.5% (for active and reactive energy)
Voltage range	90…265 V_{RMS}
Current Range	0…up to 120 A_{RMS} (5 A is nominal current, peak current is up to 152 A)
Frequency range	47…53 Hz
Meter constant (imp/kWh, imp/kVArh)	500, 1000, 2000, 5000, 10000, 20000 (default), 50000, 100000, 200000, 500000, 1000000, 2000000, 4000000 and 6000000. Note that pulse numbers above 50000 are aplicable only for low current measurement.
Functionality	V, A, kW, kVAr, kVA, kWh (import/export), kVARh (lead/lag), Hz, time, date
Voltage sensor	Voltage divider
Current sensor	Shunt down to 120 μΩ
Energy output pulse interface	two red LEDs (active and reactive energy)
Energy output pulse parameters: • Maximum frequency • On-Time • Jitter	 600 Hz 20 ms (50% duty cycle for frequencies above 25 Hz) ±10 μs at constant power
Optoisolated pulse output (optional)	optocoupler (active or reactive energy)
User interface (HMI)	LCD, one push-button, one user LED (red)
Tamper detection	two hidden buttons (terminal cover and main cover)
Infrared interface	4800/8-N-1 FreeMASTER interface
Isolated RS232 serial interface (optional only)	19200/8-N-1
RF interface (optional only)	2.4 GHz RF 1322x-LPN internal daughter card

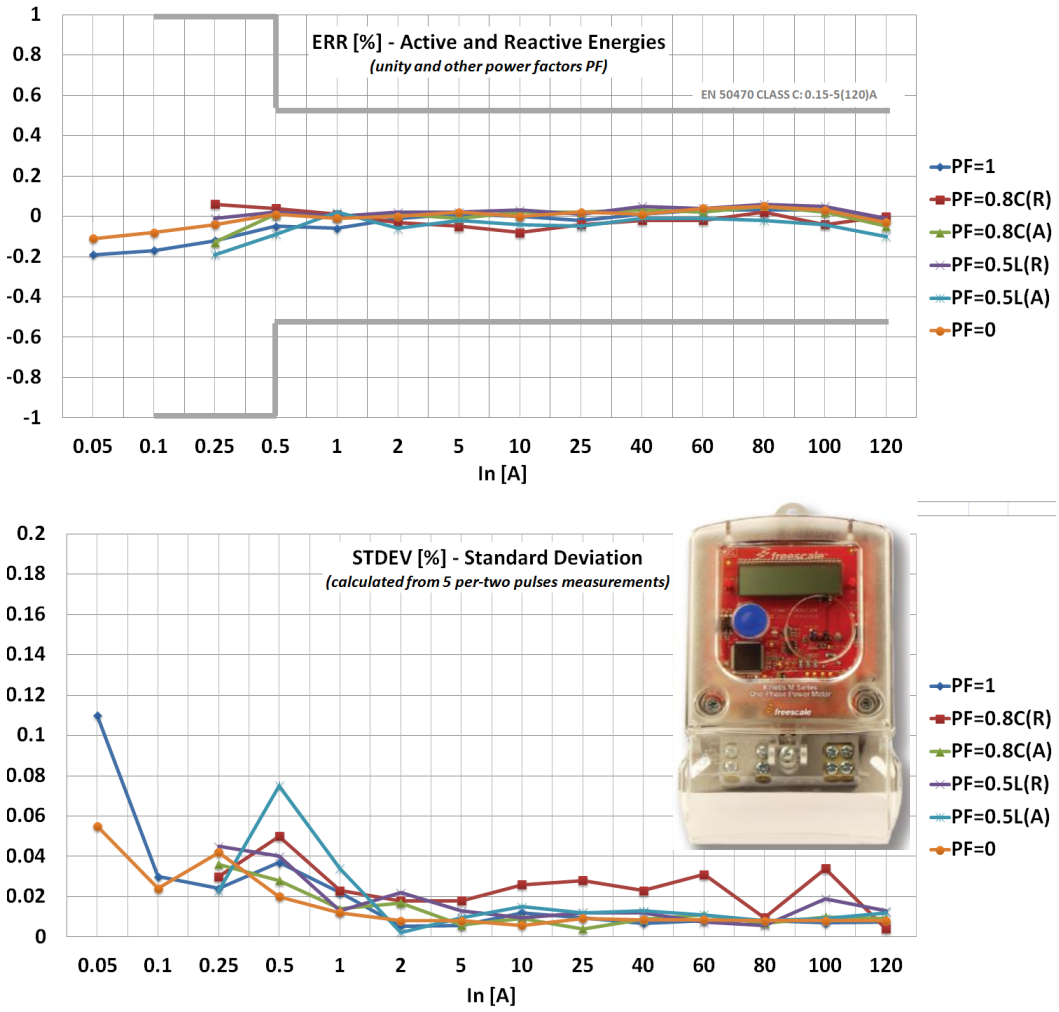
Table 15. **Single phase KM3x power meter specification**

Internal battery (for RTC)	3.6 V
Power consumption @ 3.3V and 22 °C: Measurement mode (powered from mains) Run/Menu list mode (mains disconnected) Standby (4kB system RAM back-up)	 10.88 mA 245 µA 5.6 µA (both cover closed), 4.4 µA (covers opened)
Internal battery (for RTC)	3.6 V

NOTE

The one-phase Kinetis M power meter used for performance testing was populated with a 140 µΩ shunt resistor for current measurement. Additional power meter settings and capabilities are summarized in Table 15.

Figure 59 shows the accuracy errors obtained during performance testing and validation. It is evident that both active and reactive energies at all power factors fit within the accuracy limit ±0.2% in the current dynamic range 2400:1[5].



[5] MKM34Z128 system and bus clocked by 12.288 MHz, AFE clocked by 6.144 MHz, Current scaling: I_MAX= 152A @ PGA gain=16, Voltage scaling: U_MAX= 286.0 V.

Figure 59. **Performance of the Kinetis M one-phase power meter reference design**

The measured accuracy margin, measurement repeatability, and current dynamic range, make both the MKM34Z128 Kinetis MCU and the Filter-Based Metering Algorithm suitable for modern high-performance power meters. The achieved accuracy is compliant with the EN50470-1 and EN50470-3 European standards for electronic meters of active energy classes B and C, IEC 62053-21 and IEC 62052-11 international standards for electronic meters of active energy classes 2 and 1, and the IEC 62053-23 international standard for static meters of reactive energy classes 2 and 3.

6. Summary

This application note describes a metering library that implements the Filter-Based Metering Algorithm. The presented algorithm is simple and highly accurate. It has been designed specifically for devices featuring sigma-delta converters, which have a fixed measurement sample rate.

The presented Filter-Based Metering Algorithm can be easily integrated into electronic meters and requires only instantaneous phase voltage and current samples to be provided to their inputs. All available sensing circuitries, such as a voltage divider, in the case of phase voltage measurement, or a shunt resistor, current transformer, and Rogowski coil for phase current measurement, are supported. The presented algorithms are intended for post-processing instantaneous phase voltage and current samples after phase shift compensation.

The theoretical section explains fixed-point arithmetic and the theory of digital filters and applications on a level necessary to understand the metering algorithm. Setting up of the Filter-Based Metering Algorithm can be realized by the configuration tool. This tool is designed to update configuration data directly in the configuration C-header file. The changes in the C-header file are reflected after code re-compilation. The configuration tool automates the procedure of the algorithm setup and optimization, and it supports one-phase, two-phase (Form-12S), and three-phase power meter applications.

The performance of the metering library has been tested in the one-phase Kinetis M power meter reference design – the accuracy of the measurement was in the range ±0.2 % in the current dynamic range 2400:1.

Sometimes, the lower power consumption is preferred over high accuracy. The existing metering library is characterized by high accuracy and exploiting fractional calculations in Q0.31 (32-bit) and even Q0.63 (64-bit) fractional data format. Such high accuracy requires adequate core performance and a system clock in the range 10 MHz and above. The future development and expansion of the metering library will focus on lowering computational resources while performing calculations of billing quantities at the accuracy level mandated by IEC, MID, and ANSI-C12.20 standards.

7. References

The following documents are useful when using the Filter Based Algorithm for Metering Applications.

1. *Handbook for Digital Signal Processing*, Sanjit K. Mitra, James F. Kaiser (John Wiley & Sons, 1993, USA)
2. *Discrete-Time Signal Processing*, Alan V. Oppenheim, Ronald W. Schafer (Prentice Hall, 1989, USA)
3. *Fractional and Integer Arithmetic - DSP56000 Family of General-Purpose Digital Signal Processors*, (Motorola 1993, USA)
4. Q (number format), en.wikipedia.org/wiki/Q_(number_format)
5. *Digital Computer Design Fundamentals*, Yaohan Chu (1962 by McGraw-Hill, New York, USA)
6. *Otázky a odpovedě z mikroprocesorov a mikropočítačov*, Zdeněk Sobotka (Alfa, 1986, Slovakia)
7. *RMS to DC Conversion Application Guide*, Charles Kitchin, Lew Counts (2nd Edition, Analog Devices, 1986, USA)
8. *Linear Circuit Design Handbook*, Hank Zumbahlen (editor) (Elsevier-Newnes, 1st Edition, 2008, USA)
9. *Analytic Signal Generation-Tips and Traps*, Andrew Reilly, Gordon Frazer, and Boualem Boashash (IEEE Transactions on Signal Processing, vol. 42. No. 11, November 1994)
10. *Current Sensing for Energy Metering*, William Koon, Analog Devices, USA

The following documents can be found on www.nxp.com. Additional documents not listed here can be found on the Kinetis M Series product page.

11. *DSP56800E and DSP56800EX Digital Signal Controller Cores Reference Manual* (document DSP56800ERM)
12. *Kinetis-M One-Phase Power Meter Reference Design* (document DRM143)

8. Revision History

Revision number	Date	Substantial changes
0	08/2013	Initial release
1	12/2013	Changed Section 7 - References
2	11/2014	Added Section 3.6 - Rogowski coil sensor signal processing Changed Section 4 - Power meter application development Changed Section 4.2 - Configuration tool
3	06/2015	Changed Section 4.1 - Metering libraries Added Section 4.1.1 - Core architecture and compiler support Changed Section 4.2 - Configuration tool
4	04/2016	Changed Section 4.2 - Configuration tool

Appendix A. C-Header file

```
/*******************************************************************************
 * Filter-Based Metering Library Configuration File, Created: Sun May 31 09:38:33 2015
 *******************************************************************************
 * @TAGNAME       = METERLIB1PH_CFG
 * @LOCKED        =         0
 * @FSAMPLE       =      1200
 * @DFACTOR       =         2
 * @IMAX          =  141.4214
 * @UMAX          =  350.0000
 * @FREQ          =        50
 * @COUNTERS_RES  =     10000
 * @PWR_THRESHOLD =    0.1000
 * @I_STARTING    =    0.0200
 * @APWR_OFS      =    0.0000
 * @RPWR_OFS      =    0.0000
 * @ENERGY_ATT    =    0.0000
 * @IMP_PER_KWH   =     50000
 * @IMP_PER_KVARH =     50000
 * @HPF_FCUT      =    0.3000
 * @LPF1_FCUT     =    0.5000
 * @LPF2_FCUT     =    3.0000
 * @KWIN_BETA     =    6.0672
 * @KWIN_GAIN     =    1.0000
 * @FIR_TAPS_CHG  =         0
 * @FIR_FREQ_MOD  =         0
 * @CUR_SENSOR    =         1
 * @LIB_TYPE      =         1
 * @MATH_TYPE     =         1
 * @KWH_ONLY      =         0
 * @SW_PH_CORR    =         0
 * @MCU_CORE      =         1
 *******************************************************************************/
#ifndef __METERLIB1PH_CFG_H
#define __METERLIB1PH_CFG_H

/*******************************************************************************
 * General parameters and scaling coefficients
 *******************************************************************************/
#define POWER_METER              1PH  /*!< Power meter topology                        */
#define CURRENT_SENSOR   PROPORTIONAL  /*!< Current sensor output characteristic       */
#define LIBRARY_PREFIX       METERLIB  /*!< Library prefix; high-performance library   */
#define I_MAX                 141.421  /*!< Maximal current I-peak in amperes          */
#define U_MAX                 350.000  /*!< Maximal voltage U-peak in volts            */
#define F_NOM                      50  /*!< Nominal frequency in Hz                    */
#define COUNTER_RES             10000  /*!< Resolution of energy counters inc/kWh  */
#define IMP_PER_KWH             50000  /*!< Impulses per kWh                           */
#define IMP_PER_KVARH           50000  /*!< Impulses per kVARh                         */
#define DECIM_FACTOR                2  /*!< Auxiliary calculations decimation factor   */
#define KWH_CALC_FREQ        1200.000  /*!< Sample frequency in Hz                     */
#define KVARH_CALC_FREQ      1200.000  /*!< Sample frequency in Hz                     */
/*******************************************************************************
 * Filter-based metering algorithm configuration structure
 *******************************************************************************/
#define METERLIB1PH_CFG                                                             \
{                                                                                   \
  U_MAX,                                                                            \
  I_MAX,                                                                            \
  FRAC32(((+0.1000)/(U_MAX*I_MAX/2.0))),                                            \
  FRAC32((+0.0200)/I_MAX),                                                          \
  1,                                                                                \
  {{0l,0l,0l},{0l,0l,0l}},                                                          \
  {{FRAC32(+0.99921521804155),FRAC32(-0.99921521804155),FRAC32(-0.99843043608309)}},  \
  {{FRAC32(+0.13165249758740),FRAC32(+0.13165249758740),FRAC32(-1.0)}},             \
  {{0l,0ll},{0l,0ll}},                                                              \
  {0l,0ll},                                                                         \
  {{0l,0ll},{0l,0ll}},                                                              \
  {  49,                                                                            \
    {                                                                               \
      FRAC32(0.0),FRAC32(-0.00073728465714),FRAC32(0.0),FRAC32(-0.00196750272687),    \
      FRAC32(0.0),FRAC32(-0.00411945802255),FRAC32(0.0),FRAC32(-0.00756839142185),    \
      FRAC32(0.0),FRAC32(-0.01278720365088),FRAC32(0.0),FRAC32(-0.02040684105768),    \
      FRAC32(0.0),FRAC32(-0.03136483560542),FRAC32(0.0),FRAC32(-0.04728105184137),    \
      FRAC32(0.0),FRAC32(-0.07151114503989),FRAC32(0.0),FRAC32(-0.11276139617420),    \
      FRAC32(0.0),FRAC32(-0.20318408017719),FRAC32(0.0),FRAC32(-0.63356345988777),    \
      FRAC32(0.0),FRAC32(+0.63356345988777),FRAC32(0.0),FRAC32(+0.20318408017719),    \
      FRAC32(0.0),FRAC32(+0.11276139617420),FRAC32(0.0),FRAC32(+0.07151114503989),    \
      FRAC32(0.0),FRAC32(+0.04728105184137),FRAC32(0.0),FRAC32(+0.03136483560542),    \
```

```
      FRAC32(0.0),FRAC32(+0.02040684105768),FRAC32(0.0),FRAC32(+0.01278720365088),     \
      FRAC32(0.0),FRAC32(+0.00756839142185),FRAC32(0.0),FRAC32(+0.00411945802255),     \
      FRAC32(0.0),FRAC32(+0.00196750272687),FRAC32(0.0),FRAC32(+0.00073728465714),     \
      FRAC32(0.0)                                                                      \
    },                                                                                 \
     25,                                                                               \
    {                                                                                  \
      FRAC16(0.0),FRAC16(0.0),FRAC16(0.0),FRAC16(0.0),FRAC16(0.0),FRAC16(0.0),         \
      FRAC16(0.0),FRAC16(0.0),FRAC16(0.0),FRAC16(0.0),FRAC16(0.0),FRAC16(0.0),         \
      FRAC16(0.0),FRAC16(0.0),FRAC16(0.0),FRAC16(0.0),FRAC16(0.0),FRAC16(0.0),         \
      FRAC16(0.0),FRAC16(0.0),FRAC16(0.0),FRAC16(0.0),FRAC16(0.0),FRAC16(0.0),         \
      FRAC16(-1.0)                                                                     \
    }                                                                                  \
  },                                                                                   \
  {                                                                                    \
    {                                                                                  \
      0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,  \
      0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l       \
    },                                                                                 \
    0ll,                                                                               \
    {                                                                                  \
      0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l,0l   \
    },                                                                                 \
    0l                                                                                 \
  },                                                                                   \
  {                                                                                    \
    {FRAC32(+0.00261116383261),FRAC32(+0.00261116383261),FRAC32(-0.99477767233478)},   \
    {FRAC32(+0.00261116383261),FRAC32(+0.00261116383261),FRAC32(-0.99477767233478)},   \
  },                                                                                   \
  {0ll,0ll,0l,0ll},                                                                    \
  {0ll,0ll,0l,0ll},                                                                    \
  {0ll,0ll,0l,0ll},                                                                    \
  {0ll,0ll,0l,0ll},                                                                    \
  {                                                                                    \
    FRAC48((+0.0000/(U_MAX*I_MAX))),FRAC32(+1.0000),  METERLIB_KWH_DR(   10000),       \
    {0ll,0ll,0ll},0ll,0l,FRAC16(-1.0),                                                 \
    {FRAC32(+0.00779293629195),FRAC32(+0.00779293629195),FRAC32(-0.98441412741610)},   \
    {0ll,0ll,0ll},{0ll,0ll,0ll}                                                        \
  },                                                                                   \
  {                                                                                    \
    FRAC48((+0.0000/(U_MAX*I_MAX))),FRAC32(+1.0000),METERLIB_KVARH_DR(   10000),       \
    {0ll,0ll,0ll},0ll,0l,FRAC16(-1.0),                                                 \
    {FRAC32(+0.00779293629195),FRAC32(+0.00779293629195),FRAC32(-0.98441412741610)},   \
    {0ll,0ll,0ll},{0ll,0ll,0ll}                                                        \
  }                                                                                    \
}
#endif  /* __METERLIB1PH_CFG_H */
```

Appendix B. Test application

```
#include <math.h>
#include "drivers.h"

#include "fraclib.h"
#include "meterlib.h"
#include "meterlib1ph_cfg.h"

#include "appconfig.h"

#define _PI   3.14159265358979323846    /* pi */

/* static data definitions                                                  */
static tMETERLIB1PH_DATA mlib = METERLIB1PH_CFG;
static volatile frac32  u24_sample, i24_sample;
static tENERGY_CNT wh_cnt, varh_cnt;
static double  time = 0.0, U_RMS, I_RMS, P, Q, S, U_ANGLE = (45.0/180.0)*_PI,
                                                  I_SHIFT = (-5.5/180.0)*_PI;

static int     cycle = 0;

static frac16  shift = METERLIB_DEG2SH(-5.5, 50.0);

#if defined(__ICCARM__)
  #pragma diag_suppress=Pa082
#endif
void main (void)
{
  while (1)
  {
    /* calculate phase voltage and phase current waveforms                  */
    time = time+(1.0/KWH_CALC_FREQ);
    u24_sample = FRAC24((sin(2*_PI*50.0*time+U_ANGLE)*230.0*sqrt(2)+0.0)/U_MAX));
    i24_sample = FRAC24(((sin(2*_PI*50.0*time+I_SHIFT)*5.0*sqrt(2)+0.0)/I_MAX));

    METERLIB1PH_ProcSamples(&mlib,u24_sample,i24_sample,&shift);
    METERLIB1PH_CalcWattHours(&mlib,&wh_cnt,METERLIB_KWH_PR(IMP_PER_KWH));

    /* functions below might be called less frequently - please refer to    */
    /* KWH_CALC_FREQ, KVARH_CALC_FREQ and DECIM_FACTOR constants            */
    if (!(cycle % (int)(KWH_CALC_FREQ/KVARH_CALC_FREQ)))
    {
      METERLIB1PH_CalcVarHours (&mlib,&varh_cnt,METERLIB_KVARH_PR(IMP_PER_KVARH));
    }

    if (!(cycle % DECIM_FACTOR))
    {
      METERLIB1PH_CalcAuxiliary(&mlib);
    }

    METERLIB1PH_ReadResults (&mlib,&U_RMS,&I_RMS,&P,&Q,&S);
    cycle++;
  }
}
```

How to Reach Us:

Home Page:
nxp.com

Web Support:
nxp.com/support

Information in this document is provided solely to enable system and software implementers to use Freescale products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document.

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP 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 NXP 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. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: nxp.com/SalesTermsandConditions.

NXP, the NXP logo, and Kinetis are trademarks of NXP B.V. All other product or service names are the property of their respective owners. ARM, the ARM Powered logo, and Cortex are registered trademarks of ARM Limited (or its subsidiaries) in the EU and/or elsewhere. All rights reserved.

© 2016 NXP B.V.

Document Number: AN4265
Rev. 4
04/2016