

1 Introduction

High accuracy metering is an essential feature of an electronic power meter application 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.

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, Root Mean Square (RMS) voltage, and RMS current. The active and reactive energies are sometimes referred to as billing quantities. Their computations must be compliant with IS13779:1999 for class 0.5 or equivalent standard accuracy for a dynamic range of the meter. For example, 10-60 A.

The remaining quantities are calculated for informative purposes and are referred to 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 also referred to as the Low-Power Real-Time or the LPRT Metering Algorithm. This algorithm calculates all billing and non-billing quantities in the time domain.

The LPRT Metering Algorithm integrates into an electronic power meter application. The algorithm requires only instantaneous voltage and current samples provided at constant sampling intervals. The instantaneous voltage and current samples are 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 the 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 LPRT Metering Algorithm or with the aid of delayed sampling before using them.

The followings section provides a block diagram and a brief description of the LPRT Metering Algorithm in a one-phase power meter configuration.

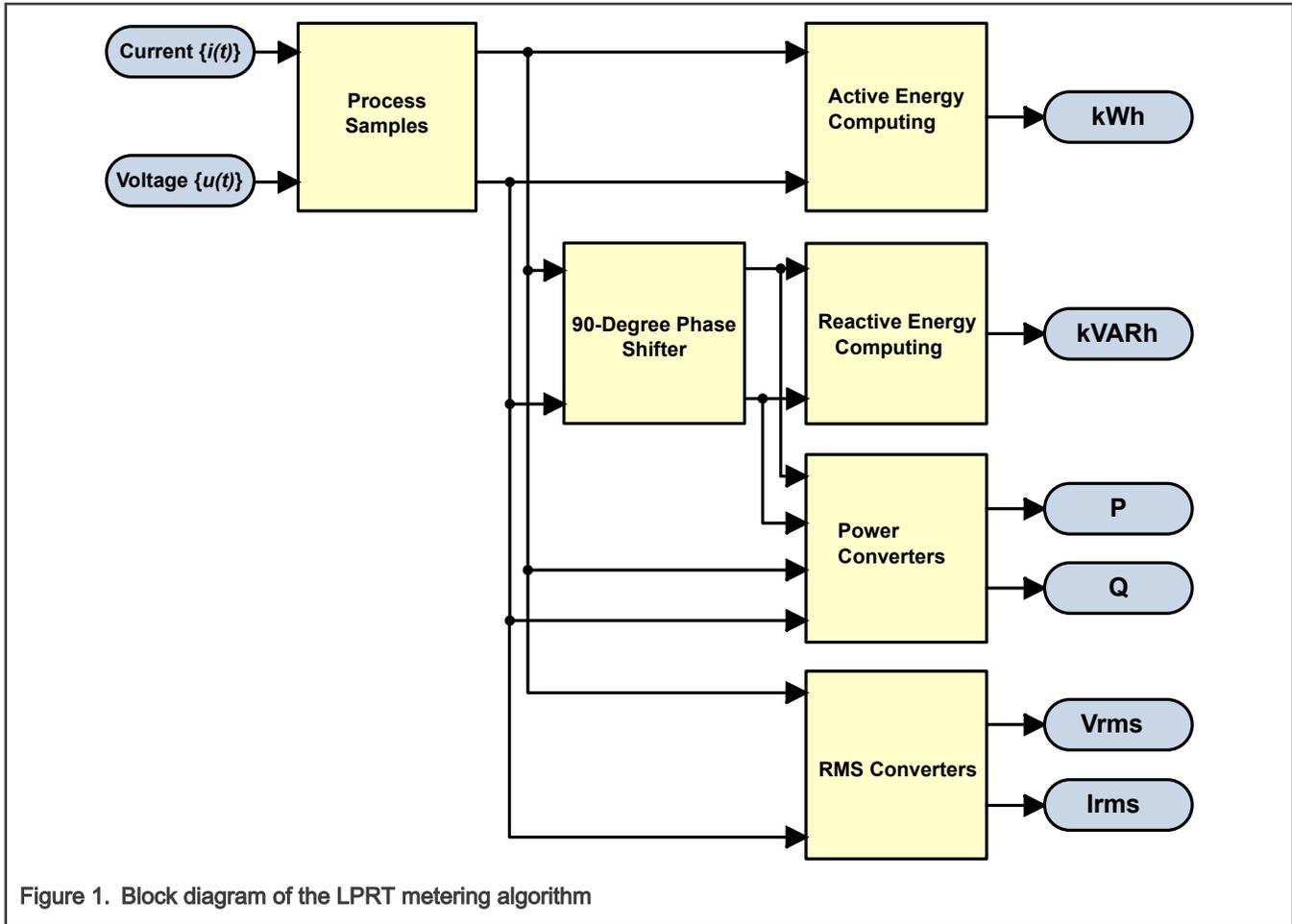
2 Block diagram

Figure 1 shows a block diagram of the LPRT 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 time-aligned instantaneous current and voltage samples at constant sampling intervals. The new voltage and current samples trigger a recalculation using some part of the algorithm blocks. After every one second, consolidated recalculation is done using algorithm blocks and then new billing and non-billing quantities will become available. All calculated quantities are 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.

Contents

1	Introduction	1
2	Block diagram	1
3	Theory	2
3.1	Explicit RMS converter.....	3
3.2	Average power converter.....	3
4	Power meter application development	4
4.1	Metering library.....	6
5	Accuracy and performance testing	25
6	Summary	29
7	References	30
8	Revision history	30
9	C-Header file	30
10	Test application	34





The algorithm consists of several blocks as described below.

The first block in the signal flow is sample processing. This block removes offset from the instantaneous voltage and current samples and returns the order of voltage waveform for single or 3 three-phase systems. If the Direct Current (DC) offset is stable and deterministic, DC is removed 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. The DC offset of the analog circuits is not constant but varies with time and other factors. Due to this fact, the first block calculates the DC offsets periodically and thereafter removes them from the alternating voltage and current measurements.

The second block is essential for reactive energy calculation and is called the 90-degree phase shifter. This block implements a sample buffering method to pick up the phase-shifted samples.

The following blocks in the signal flow diagram are the active and reactive energy computing and energy counter updater. These blocks calculate active and reactive energies.

The RMS converters 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 is called with the new samples of the analyzed signal and at the end of no of periods of the analyzed fundamental frequency. For details, 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 mathematical calculations based on the theory described in this document, For details, see [Average power converter](#).

3 Theory

The LPRT Metering Algorithm comprises several blocks. The calculations performed by the algorithm are based on elementary calculations, such as addition, subtraction, integration, multiplication, and square root.

3.1 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 (I_{RMS}) 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:

$$I_{RMS} = \sqrt{\frac{1}{T} \int_0^T [i(t)]^2 dt}$$

Equation 1. Computation of the RMS current from the signal function

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.

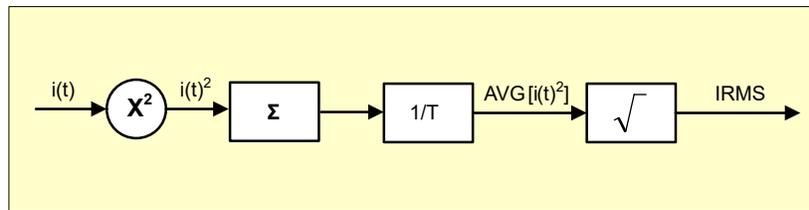


Figure 2. Explicit RMS current converter

This method for computing the RMS value calls the numerical square and summing functions every time a new sample of the analyzed signal is obtained. At the end, the number of periods per second of the analyzed signal, average, and square root functions are utilized to compute RMS values. Figure 6 shows the explicit RMS converter implementation for RMS current computation.

The section [Average power converter](#) describes a similar method for the calculation of active and reactive power.

3.2 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 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. The average power of any whole number of cycles is the same as the average power value of just one cycle. You can find the average power of a very long-duration periodic waveform by calculating the average value of one complete cycle.

$$P = \frac{1}{T} \int_0^T u(t) i(t) dt$$

Equation 2. Calculating the average value

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

Figure 3 shows that the power is calculated by multiplying instantaneous voltage and current samples.

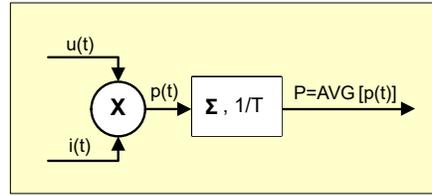


Figure 3. Average active power converter

The LPRT Metering Algorithm uses the average power converter for the calculation of active power (P) and reactive power (Q). The reactive power (Q) is measured in units of volt-amperes-reactive (V_{AR}) 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 for the current input waveform.

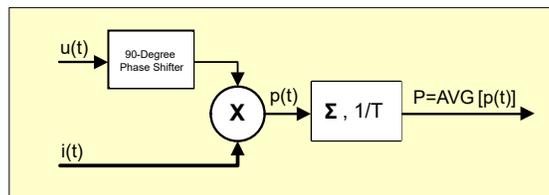


Figure 4. Average reactive power converter

LPRT Metering Algorithm uses buffering and weighted average to phase shift the voltage input waveform by 90 degrees for the current input waveform. The shifted waveforms are then used for calculating the reactive power (Q) and reactive energy ($kVARh$).

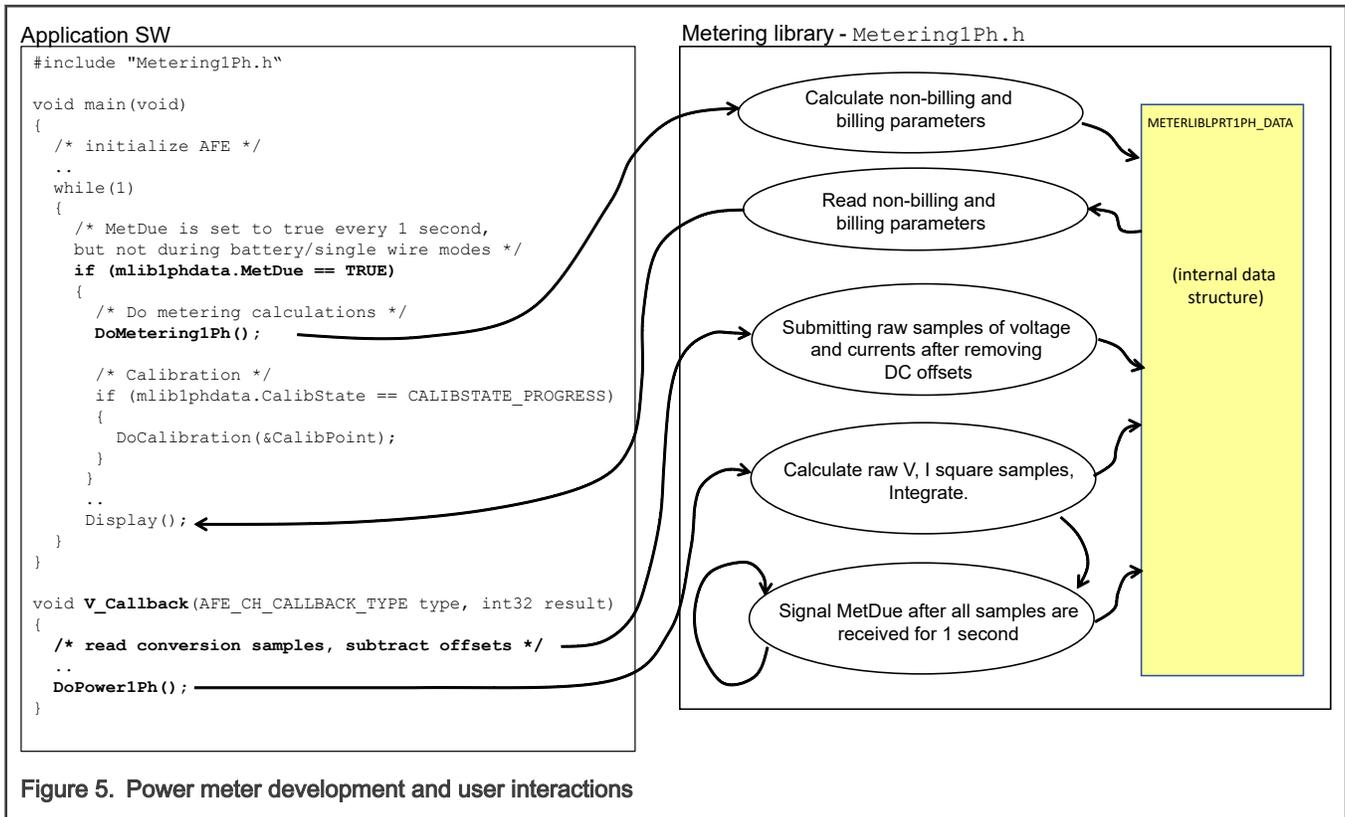
4 Power meter application development

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 is the time to market that drives power meter development milestones. The metrology portion of the power meter must be robust and behave deterministically under all conditions. Therefore, 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 LPRT Metering Algorithm theory, this section 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 are called at the fixed sampling intervals.

NOTE

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



The software required to perform basic metering functionality is 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** – This comprises a set of highly optimized functions for calculating the billing and non-billing quantities from the measured phase voltage and current samples.

Figure 5 depicts the usage of the metering library 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 Callback() 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 DoPower1Ph() function.

DoPower1Ph() internally sets the variable 'MetDue' of the tMETERLIBLPR1PH_DATA data structure once about every second. At this point, the application should call DoMetering1Ph() function to calculate all non-billing and billing parameters. Practically, these functions shall be called at least 1200 times per second to calculate active and reactive energies in the frequency bandwidth up to the 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 sample rate will not bring any advantage.

Finally, the information stored within the metering library's internal data structure can be read by the DoMetering1Ph() function. This function is usually called from the main() function or a low-frequent software task. The calling frequency should be once every second, which is indicated by the variable 'MetDue' of the tMETERLIBLPR1PH_DATA data structure. By calling DoMetering1Ph(), the variable 'MetDue' of the tMETERLIBLPR1PH_DATA is negated internally and will be set again after 1 second, as described above.

Figure 5 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.

The metering library support one-phase and three-phase power meter applications. These deliverables are discussed in the following sections.

4.1 Metering library

This section describes the metering library implementation of the LPRT 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 and three-phase.

NOTE

The IAR Embedded Workbench for Arm® (version 7.50.3) tool is used to obtain performance data for all library functions. The code is compiled with full optimization for execution speed for the MKM35Z512 an Arm® Cortex®-M0+ core-based target [4]. The device is clocked at 12 MHz using the Phase-Locked Loop(PLL) for one-phase power meter and 24 MHz Frequency-Locked Loop (FLL) module operating in FLL Engaged External (FEE) mode for three-phase power meter, 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.

[Figure 6](#) shows a simple block diagram of the computing process, split by the functions realized by the LPRT metering library in a one-phase power meter application.

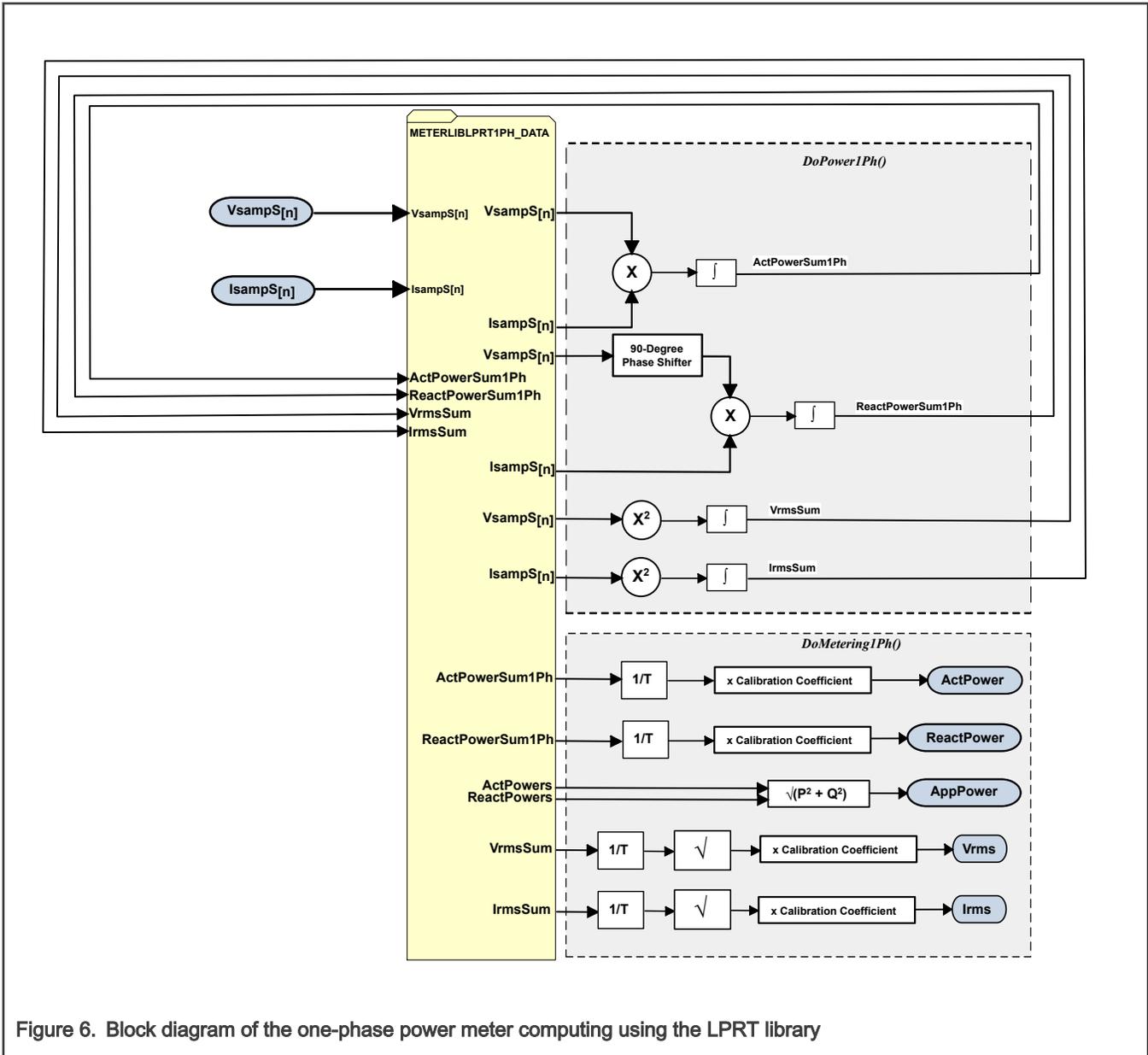


Figure 6. Block diagram of the one-phase power meter computing using the LPRT library

A detailed description of the exported data types of the libraries, data types of the library functions, and APIs, is given in the following subsections.

4.1.1 Core architecture and compiler support

The LPRT metrology library supports Arm® Cortex®-M0+ core.

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

Include files and libraries			METERLIBLPRT
			Cortex-M0+ w/o MMAU
include	files		Metering1Ph.h MeteringRun1Ph.h Calibration1Ph.h Fundamental1Ph.h Metering3Ph.h MeteringRun3Ph.h Calibration1Ph.h Fundamental3Ph.h
	paths		..\lib\inc
library	files	iar	meterlprtlb_cm0p_iar.a
	paths	iar	..\lib
library	files	mcuxpresso	libmeterlprtlb_cm0p.a
	paths	mcuxpresso	..\lib
library	files	keil	meterlprtlb_cm0p_mdk.lib
	paths	keil	..\lib

Figure 7. LPRT library integration

4.1.2 LPRT library function API

This section summarizes the API functions in the LPRT metering library libmeterliblprt_cmp0p.a, meterlprtlb_cm0p_iar.a, or meterlprtlb_cm0p_mdk.lib. Prototypes of all functions and internal data structures are declared in the Metering1Ph.h, Calibration1Ph, Metering3Ph.h, and Calibration3Ph.h header files.

4.1.2.1 One-phase power meter

- `void MeterLibLPRT1Ph_InitParams (tMETERLIBLPRT1PH_DATA *mLib, uint16 nSamples, uint16 samplesForOffset, float *pFreqDependentPhErr, uint8 doFundamental);`

Initializes meter library data structure object tMETERLIBLPRT1PH_DATA with user-defined parameters.

- `void DoPower1Ph (void);`

Process voltage and current samples for billing and non-billing parameter calculations.

- `void DoMetering1Ph (void);`

Calculate billing and non-billing parameters and finalize, once every second.

- `void CorrectAppPhAngle1Ph (uint8 currenttoUse);`

LPRT library provides this callback function to the application to correct the phase angle, if required.

- `void FudgeParameters1Ph (void);`

LPRT library provides this callback function to the application to tune up the non-billing and billing parameters.

- `void EnableGain (void);`

LPRT library provides this callback function to the application to enable gain of the current channel.

- `void DisableGain(void);`

LPRT library provides this callback function to the application to disable gain of the current channel.

- `void DoCalibration1Ph(tCalibPoint1Ph *calibpoint);`

LPRT library provides this callback function to the application to start calibration of the meter.

4.1.2.2 Three-phase power meter

- `void MeterLibLPRT3Ph_InitParams(tMETERLIBLPRT3PH_DATA *mlib, uint16 nSamples, uint16 samplesForOffset, float *pFreqDependentPhErr, uint8 doFundamental);`

Initializes meter library data structure object tMETERLIBLPRT3PH_DATA with user-defined parameters.

- `void ProcSamples3Ph(uint8 channel);`

Process each phase voltage, current samples for a phase in a three-phase power meter.

- `void DoPower3Ph(void);`

Process voltage and current samples for billing and non-billing parameter calculations.

- `void DoMetering3Ph(void);`

Calculate billing and non-billing parameters and finalize, once every second.

- `void CorrectAppPhAngle3Ph(uint8 currenttoUse);`

LPRT library provides this callback function to the application to correct the phase angle.

- `void FudgeParameters3Ph(void);`

LPRT library provides this callback function to the application to tune up the non-billing and billing parameters.

- `void DoCalibration3Ph(tCalibPoint3Ph *calibpoint);`

LPRT library provides this callback function to the application to start calibration of the meter.

4.1.3 Data structures

This section describes the data structures for accessing those state variables calculated by the LPRT metering library.

4.1.3.1 tMETERLIBLPRT1PH_DATA

Structure containing LPRT metering library parameters for a one-phase power meter.

4.1.3.1.1 Reference

```
#include "Metering1Ph.h"
```

4.1.3.1.2 Data fields

Type	Name	Description
float	Vrms	Calculated RMS value of the voltage
float	Irms[nCURRENTS]	Calculated RMS currents - Phase and neutral
float	VrmsNoFudge	RMS value of voltage w/o compensation
float	IrmsNoFudge[nCURRENTS]	RMS value of currents w/o compensation
float	PFmetImax	Calculated Power factor value during forced Imax condition
float	ActPowers[nCURRENTS]	Calculated active power
float	ActPowersNoFudge[nCURRENTS]	Calculated active power value w/o compensation
float	ReactPowers[nCURRENTS]	Calculated reactive power
float	AppPowers[nCURRENTS]	Calculated apparent power
float	AppPowersNoFudge[nCURRENTS]	Calculated apparent power w/o compensation
float	PowerFactors[nCURRENTS]	Calculated power factor
float	Frequency	Calculated frequency
unsigned short int	nSamples	No of voltage/current samples per second - application to load
unsigned char	MetDue	Indicates metrology processing is due for a second
unsigned char	MetOnImax	Indication to do metering with Imax condition - application to load
signed char	ISigns[nCURRENTS]	Forward/reverse sign of currents, 1 = forward, -1 = reverse
unsigned char	ReactSampleIndex	Reactive sample index – internal
unsigned char	CurToUse	Indicates which current has higher value - Phase/neutral
unsigned short int	NSamps	Indicates current sample index ~ [0-(nSamples-1)]
int	VOffset	Calculated Voltage offset
int *	pVQCycleSamps	90-degree phase-shifted voltage samples buffer pointer – internal
int	IOfstSum[nCURRENTS]	Current offset sum – internal
int	IOffsets[nCURRENTS]	Calculated current offsets
int	VSampsS	Offset compensated voltage sample

Table continues on the next page...

Table continued from the previous page...

Type	Name	Description
int	VOfstSum	Voltage offset sum – internal
unsigned long long	VrmsSums[nBUFFERS]	Square sums of the voltage samples – internal
unsigned long long	IrmsSums[nBUFFERS][nCURRENTS]	Square sums of the current samples – internal
unsigned long long	RelaySenseSums[nBUFFERS]	Square sums of the relay sense voltage samples – internal
int	VRelayOfstSum	Relay sense voltage offset sum – internal
int	VRelayOffset	Calculated relay voltage offset
float	IBasic	Basic current of the meter - application to load
float	IMax	Maximum current rating of the meter - application to load
float	VHystHigh	High threshold value of voltage to enable gain for the current measurement - application to load
float	VHystLow	Low threshold value of voltage to disable gain for the current measurement - application to load
float	MaxPower	Maximum value of calculated power to restrict power/energy count calculation - application to load
int	ISamps[nCURRENTS]	Offset compensated current samples
float	PhAngles[2]	Calculated Phase angle
unsigned char	MetRecordingType	Power/energy recording type METREC_FWDED/METREC_NET
unsigned char	WBuffer	Buffer index of voltage/current samples buffers for a second period
unsigned char	FirstTime	internal use
unsigned char	IsGainEnabled	Returns current gain is status. TRUE = enabled, FALSE = disabled
unsigned char	LastGainStatus	Related to current gain status – internal
unsigned char	IncEnerPtrs[3]	Indicated energy pointer types active, reactive and apparent – internal
unsigned long int	MetEnergyCounts[3]	Calculated energy counts for 50 periods

Table continues on the next page...

Table continued from the previous page...

Type	Name	Description
unsigned long long	MetEnergySecCounts[3]	Calculated energy counts for last 1 second
unsigned char	DoFundamental	Indicate whether fundamental frequency calculation to be done - application to load
unsigned char	fSampNo	Used for fundamental frequency-related calculations – internal
unsigned char	CalibState	Holds calibration state CALIBSTATE_IDLE, CALIBSTATE_PROGRESS, CALIBSTATE_COMPLETE

4.1.3.2 tCalibStruct1Ph

Structure containing LPRT metering library calibration parameters for one-phase meter.

4.1.3.2.1 Reference

```
#include "Metering1Ph.h"
#include "Calibration1Ph.h"
```

4.1.3.2.2 Data fields

Type	Name	Description
unsigned long int	CalibSign	Defines calibration status
float	FrequencyCoeff	Frequency coefficient for calibration
float	VCoeff	Calibration Voltage coefficient
float	ICoeff[nCURRENTS]	Calibration Current coefficient with gain enabled
float	ActPowerCoeff[nCURRENTS]	Phase active power coefficient
float	PhAngle[nCURRENTS]	Phase error coefficient
float	VFCoeff	Calibration Voltage coefficient fundamental
float	IFCoeff[nCURRENTS]	Calibration Current coefficient fundamental
float	ICoeff_LP	Calibration Current coefficient with gain disabled
float	ActPowerCoeff_LP	Calibration power coefficient with gain disabled
signed char	RTCCompValue	RTC crystal compensation value
unsigned char	RTCCompInterval	RTC crystal compensation interval

Table continues on the next page...

Table continued from the previous page...

Type	Name	Description
unsigned char	PhCalibDone	Phase calibration done
unsigned char	NuCalibDone	Neutral calibration done
unsigned short int	CRC	CRC

4.1.3.3 tCalibPoint1Ph

Structure containing LPRT metering library calibration point parameters for a one-phase power meter.

4.1.3.3.1 Reference

```
#include "Metering1Ph.h"
#include "Calibration1Ph.h"
```

4.1.3.3.2 Data fields

Type	Name	Description
float	Voltage	Calibration voltage - application to load
float	Current	Calibration current - application to load
float	PhAngle	Calibration Phase angle in radians - application to load
float	PowerFactor	Calibration power factor - application to load
float	Frequency	Calibration frequency - application to load
float	CalibPhase	Calibration CURRENT_PHASE or CURRENT_NEUTRAL - application to load

4.1.3.4 tMETERLIBLPRT3PH_DATA

Structure containing LPRT metering library parameters for a three-phase power meter.

4.1.3.4.1 Reference

```
#include "Metering3Ph.h"
```

4.1.3.4.2 Data fields

Type	Name	Description
float	Vrms[nVPHASES]	Calculated RMS value of the voltages

Table continues on the next page...

Table continued from the previous page...

Type	Name	Description
float	Irms[nIPHASES]	Calculated RMS currents
float	VrmsNoFudge[nIPHASES]	RMS value of voltages w/o compensation
float	IrmsNoFudge[nIPHASES]	RMS value of currents w/o compensation
float	Frequency	Calculated frequency
unsigned short int	nSamples	No of voltage/current samples per second - application to load
unsigned char	ReactSampleIndex	Reactive sample index – internal
short int *	pVQCycleSamps[nVPHASES]	90-degree phase-shifted voltage samples buffer pointer – internal
unsigned long long	VrmsSums[2][nVPHASES]	Square sums of the voltage samples – internal
unsigned long long	IrmsSums[2][nIPHASES]	Square sums of the current samples – internal
unsigned char	MetDue	Indicates metrology processing is due for a second
unsigned char	MetOnImax	Indication to do metering with Imax condition - application to load
short int	VSampsS[nVPHASES]	Offset compensated voltage samples
unsigned long int	VOfstSum[nVPHASES]	Voltage offset sums – internal
int	IOfstSum[nIPHASES]	Current offset sums – internal
int	IOffsets[nIPHASES]	Calculated current offsets
unsigned short int	VOffsets[nVPHASES]	Calculated Voltage offsets
int	ISamps[nIPHASES]	Offset compensated current samples
signed char	ISigns[nVPHASES]	Forward/reverse sign of currents, 1 = forward, -1 = reverse
float	ActPowers[nVPHASES]	Calculated active power values
float	ReactPowers[nVPHASES]	Calculated reactive power values
float	AppPowers[nVPHASES]	Calculated apparent power values
float	PhPowerFactors[nVPHASES]	Calculated power factor values of three-phases
float	PowerFactor	Calculated total power factor
float	Powers[nPOWERS_3PH]	Calculated powers values per second

Table continues on the next page...

Table continued from the previous page...

Type	Name	Description
float	PhAngles[nVPHASES]	Calculated Phase angles
float	AppPowersNoFudge[nVPHASES]	Calculated apparent power values w/o compensation
float	ActPowerNoFudge	Calculated total active power value w/o compensation
float	AppPowerNoFudge	Calculated total apparent power value w/o compensation
float	PhPFNoFudge[nIPHASES]	Calculated power factors w/o compensation
float	IBasic	Basic current of the meter - application to load
float	IMax	Maximum current rating of the meter - application to load
float	MaxPower	Maximum value of calculated power to restrict power/energy count calculation - application to load
unsigned char	MetRecordingType	Power/energy recording type METREC_FWDED/METREC_NET
unsigned char	WBuffer	Buffer index of voltage/current samples buffers for a second period
unsigned char	IncEnerPtrs[3]	Indicated energy pointer types active, reactive and apparent – internal
unsigned long int	MetEnergyCounts[3]	Calculated energy counts for 50 periods
unsigned long long	MetEnergySecCounts[3]	Calculated energy counts for last 1 second
unsigned char	DoFundamental	Indicate whether fundamental frequency calculation to be done - application to load
unsigned char	CalibState	Holds calibration state CALIBSTATE_IDLE, CALIBSTATE_PROGRESS, CALIBSTATE_COMPLETE

4.1.3.5 tCalibStruct3Ph

Structure containing LPRT metering library calibration parameters for a three-phase meter.

4.1.3.5.1 Reference

```
#include "Metering3Ph.h"
#include "Calibration3Ph.h"
```

4.1.3.5.2 Data fields

Type	Name	Description
unsigned long int	CalibStatus	Defines calibration status = 0xFFFFFFFF for phases, 0x0000FFFF for neutral
float	VCoeff[nVPHASES]	Calibration Voltage coefficient for all phases
float	ICoeff[nIPHASES]	Calibration Current coefficient for all phases
float	ActPowerCoeff[nVPHASES]	Phase active power coefficient for all phases
float	FrequencyCoeff	Frequency coefficient for calibration
float	PhAngle[nVPHASES]	Phase error coefficient.
float	VFCoeff[nVPHASES]	Calibration Voltage coefficient fundamental
float	IFCoeff[nIPHASES]	Calibration Current coefficient fundamental
signed char	RTCCompValue	RTC crystal compensation value
unsigned char	RTCCompInterval	RTC crystal compensation interval
unsigned short int	CRC	CRC

4.1.3.6 tCalibPoint3Ph

Structure containing LPRT metering library calibration point parameters for a three-phase power meter.

4.1.3.6.1 Reference

```
#include "Metering3Ph.h"
#include "Calibration3Ph.h"
```

4.1.3.6.2 Data fields

Type	Name	Description
float	Voltage	Calibration voltage - application to load
float	Current	Calibration current - application to load
float	PhAngle	Calibration Phase angle in radians - application to load
float	PowerFactor	Calibration power factor - application to load
float	Frequency	Calibration frequency - application to load

4.1.4 MeterLibLPRT_InitParams

These functions initialize the LPRT meter library along with few application-specific parameters. It also resets most of the metering library parameters of structure object tMETERLIBLPRT1PH_DATA and tMETERLIBLPRT3PH_DATA once before starting metering application functioning.

4.1.4.1 Syntax

```
#include "Metering1Ph.h"
void MeterLibLPRT1Ph_InitParams(tMETERLIBLPRT1PH_DATA *mlib,
                               uint16 nSamples, uint16 samplesForOffset,
                               float *pFreqDependentPhErr, uint8 doFundamental);

#include "Metering3Ph.h"
void MeterLibLPRT3Ph_InitParams(tMETERLIBLPRT3PH_DATA *mlib,
                               uint16 nSamples, uint16 samplesForOffset,
                               float *pFreqDependentPhErr, uint8 doFundamental);
```

Arguments

Type	Name	Direction	Description
tMETERLIBLPRT1PH_DATA *	mlib	in/out	Pointer to LPRT library data structure tMETERLIBLPRT1PH_DATA.
tMETERLIBLPRT3PH_DATA *	Mlib	in/out	Pointer to LPRT library data structure tMETERLIBLPRT3PH_DATA.
unsigned short int	nSamples	In	Number of voltage/current samples per second.
unsigned short int	samplesForOffset	in/out	Number of samples per second to be used for offset calculation.
float *	pFreqDependentPhErr	in/out	Application-defined frequency-dependent phase correction values.
unsigned char	doFundamental	in	Application-defined initialization if the metering library should perform fundamental calculation-related tasks.

4.1.4.2 Return

None.

4.1.4.3 Description

These functions initialize tMETERLIBLPRT1PH_DATA or tMETERLIBLPRT3PH_DATA metering structure object with few application-specific parameters. Typically to be called once in application run cycle.

4.1.5 DoPower

This function does a part of RMS and power calculation per the voltage and current samples. Also, does the offset calculations. Once this function finished all samples processing for 50 periods (50 Hz signal), metering application specific MetDue is set to 'TRUE' value.

4.1.5.1 Syntax

```
#include "Metering1Ph.h"
void DoPower1Ph (void);
#include "Metering3Ph.h"
void DoPower3Ph (void);
```

4.1.5.2 Arguments

These functions do not use any arguments.

4.1.5.3 Return

None.

4.1.5.4 Description

These functions must be called after 1 set of voltage, current samples are received.

Voltage samples are squared and integrated for 1 second and stored into VrmsSum.

Current samples are squared and integrated for 1 second and stored into IrmsSum.

Voltage and in-phase current samples are multiplied, integrated for 1 second, and stored into ActPowerSum.

For reactive power, current samples and voltage samples shifted by 90-degree phase shifter are multiplied, integrated for 1 second, and stored into ReactPowerSum.

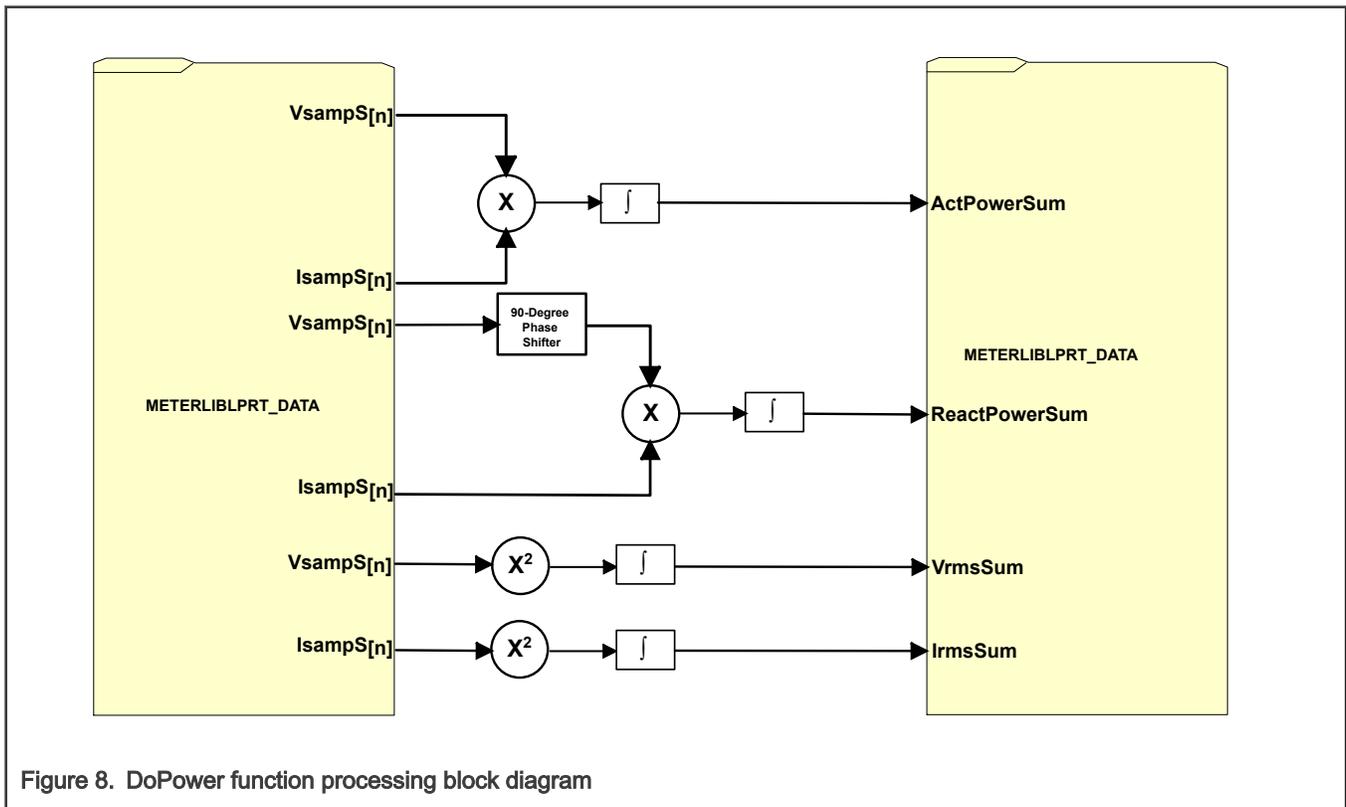
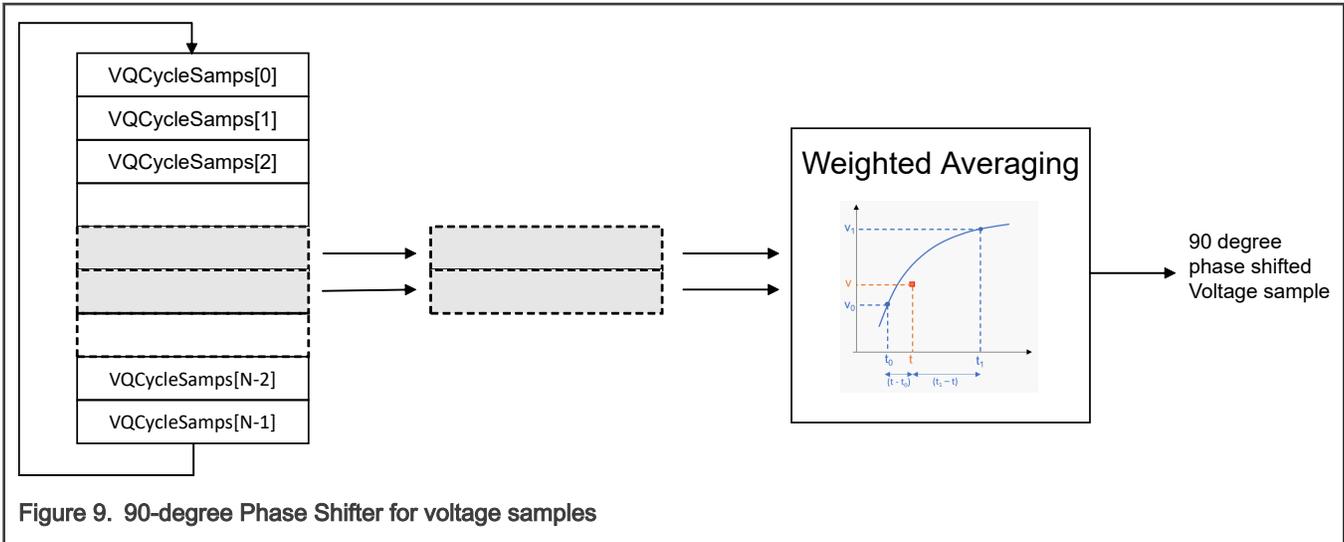
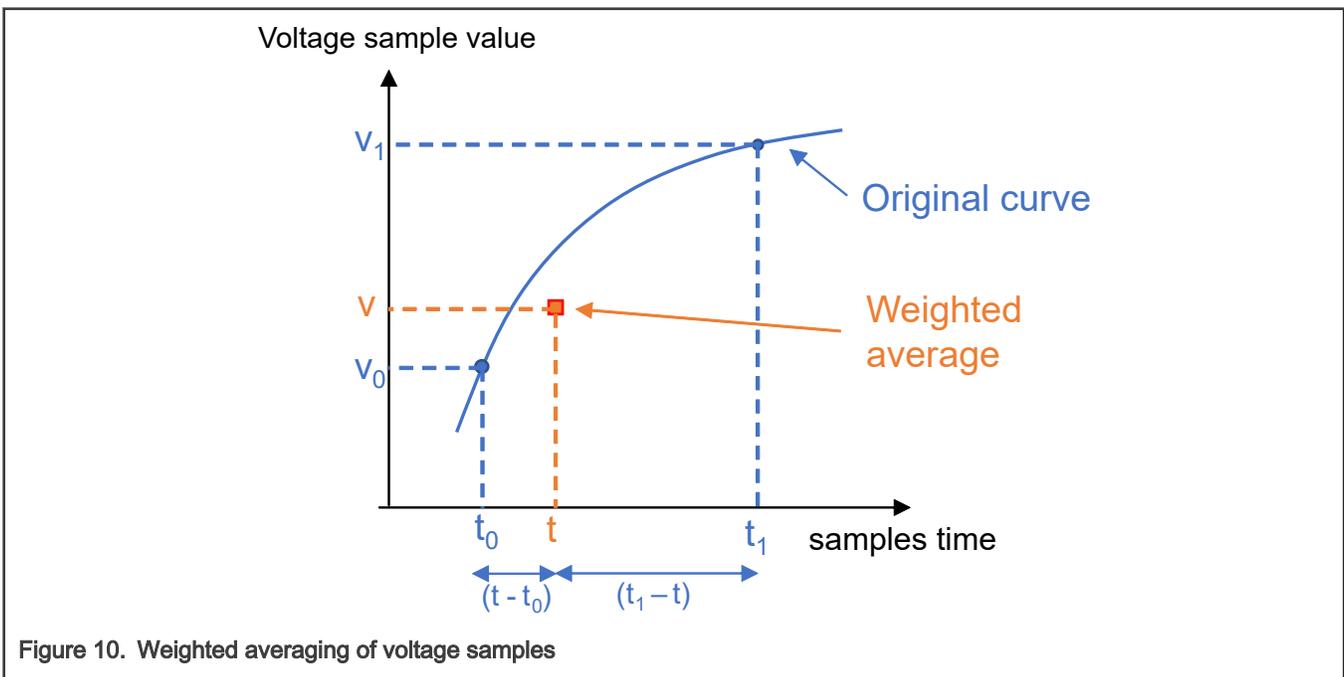


Figure 8. DoPower function processing block diagram

90-degree phase-shifted samples are derived by storing raw voltage samples in a circular buffer for a quarter cycle period of the input fundamental frequency. To derive the quarter cycle phase shifted sample of incoming voltage samples stream, 2 adjacent samples are identified from the circular voltage buffer and are linearly interpolated.



Weighted averaging of samples of the originally non-linear voltage signal will introduce error in power/energy calculation but this process is simple and fast. This is suitable for the low operating speed of meter MCU and found adequate for up to class 0.5 of energy meter applications.



4.1.5.5 Performance

Function	Code size	Stack size	Clock cycles
DoPower1Ph	486	15	2825
DoPower3Ph	1132	10	1919

4.1.6 DoMetering

These functions recalculate all billing and non-billing quantities for the last 1 second period.

4.1.6.1 Syntax

```
#include "Metering1Ph.h"
void DoMetering1Ph (void);
#include "Metering3Ph.h"
void DoMetering3Ph (void);
```

4.1.6.2 Arguments

None.

4.1.6.3 Return

None.

4.1.6.4 Description

The DoMetering() function must be called after tMETERLIBLPRT_DATA -> MetDue is set to 'TRUE' value by DoPower() function. DoMetering() function internally clears tMETERLIBLPRT_DATA -> MetDue to 'FALSE'.

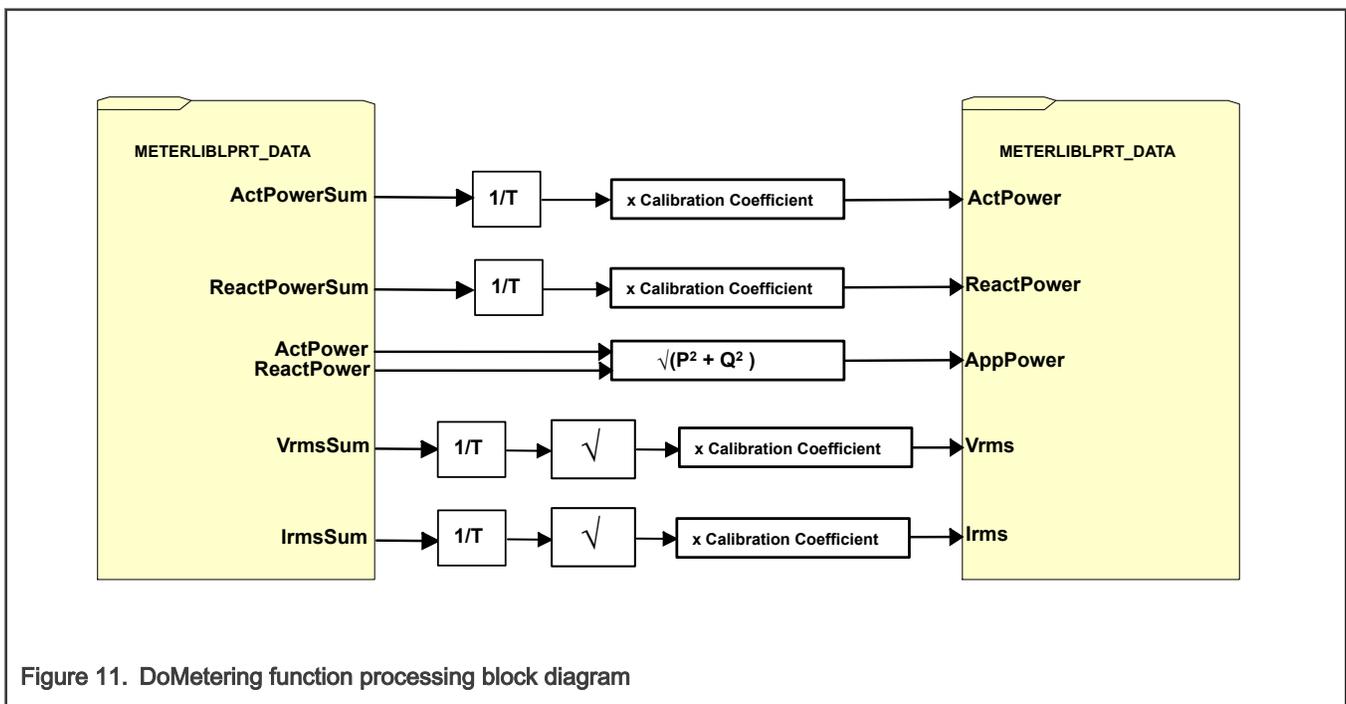


Figure 11. DoMetering function processing block diagram

In DoMetering() functions, processed outputs of DoPower() functions are finalized for re-computation of non-billing and billing parameters, once every second.

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.

$$Wh = \frac{1}{3600} \int_0^{\infty} u(t) i(t) dt$$

Equation 3. Computing active energy

Calculated active power of one second is converted to active energy counters as per the following equations.

$$\mathbf{MetEnergySecCounts}[0] = \frac{2^{(28)}}{3600} \times \mathbf{ActPower}$$

Equation 4. Calculated active power

$$0 \leq \mathbf{MetEnergySecCounts}[0] \leq [2^{(28)} - 1]$$

Equation 5. Calculated active energy counters

Power values are multiplied with $2^{(28)}$ to retain high-resolution decimal values. **MetEnergySecCounts** are recalculated every power integration period of '1' second and must be accumulated by the application only.

The reactive energy in a typical one-phase power meter is computed as an infinite integral of the unbiased instantaneous shifted phase voltage $u(t-90^\circ)$ and phase current $i(t)$ waveforms.

$$\mathbf{VARh} = \frac{1}{3600} \int_0^\infty u(t-90^\circ) i(t) dt$$

Equation 6. Reactive energy

Calculated reactive power of one second is converted to reactive energy counters as per the following equations.

$$\mathbf{MetEnergySecCounts}[1] = \frac{2^{(28)}}{3600} \times \mathbf{ReactPower}$$

Equation 7. Calculated reactive power

$$0 \leq \mathbf{MetEnergySecCounts}[1] \leq [2^{(28)} - 1]$$

Equation 8. Calculated reactive energy counters

These functions retrieve apparent power (s) from the internal data structure. The total power in an AC circuit (both absorbed and dissipated) is referred to as the total apparent power (s). The apparent power is measured in the units of volt-amperes (VA). For any general waveforms with higher harmonics, the apparent power is given by the product of the RMS phase current and RMS phase voltage.

$$\mathbf{S} = \mathbf{IRMS} * \mathbf{URMS}$$

Equation 9. Apparent power given by a product

For sinusoidal waveforms with no higher harmonics, the apparent power can also be calculated using the power triangle method, as a vector sum of the active power (p) and reactive power (q) components.

$$\mathbf{S} = \sqrt{\mathbf{P}^2 + \mathbf{Q}^2}$$

Equation 10. Power triangle method

Calculated apparent power of one second is converted to energy counters as per the following equations.

$$\mathbf{MetEnergySecCounts}[2] = \frac{2^{(28)}}{3600} \times \mathbf{AppPower}$$

Equation 11. Calculated apparent power

$$0 \leq \text{MetEnergySecCounts}[2] \leq [2^{(28)} - 1]$$

Equation 12. Calculated apparent energy counters

The non-billing parameters such as **Vrms**, **Irms**, **ActPower**, **ReactPower** and billing parameters such as **MetEnergySecCounts[]** and are stored in the metering library data structure objects **tMETERLIBLPRT_DATA** and are directly read by the metering application.

4.1.6.5 Performance

Function	Code size	Stack size	Clock cycles
DoMetering1Ph	2582	50	165847
DoMetering3Ph	2636	46	203837

4.1.7 CorrectAppPhAngle

These functions provide an option to further tune-up phase angle correction by the user application.

4.1.7.1 Syntax

```
#include "Metering1Ph.h"
void CorrectAppPhAngle1Ph (uint8 currenttoUse);
#include "Metering3Ph.h"
void CorrectAppPhAngle3Ph (uint8 currenttoUse);
```

4.1.7.2 Arguments

Type	Name	Direction	Description
Unsigned char	currenttoUse	In	Phase number CURRENT_PHASE or CURRENT_NEUTRAL for a one-phase power meter. Phase number V_R_PHASE, V_Y_PHASE, or V_B_PHASE for a three-phase power meter.

4.1.7.3 Return

None.

4.1.7.4 Description

CorrectAppPhAngle() functions are called from DoMetering () library functions.

4.1.8 FudgeParameters

These functions are used to clean up any leakage current etc. calculated as per the threshold values defined in the application etc.

4.1.8.1 Syntax

```
#include "Metering1Ph.h"
void FudgeParameters1Ph (void);
#include "Metering3Ph.h"
void FudgeParameters3Ph (void);
```

4.1.8.2 Arguments

None.

4.1.8.3 Return

None.

4.1.8.4 Description

FudgeParameters() functions are called from DoMetering() library functions.

4.1.9 ChkVolLvl

This function is called from the metering library to allow an application to check current phase voltage and indicate the metering library to switch to low-power mode measurements of non-billing parameters through tMETERLIBLPRT1PH_DATA -> IsGainEnabled parameter.

4.1.9.1 Syntax

```
#include "Metering1Ph.h"
void ChkVolLvl (void);
```

4.1.9.2 Arguments

None.

4.1.9.3 Return

None.

4.1.9.4 Description

ChkVolLvl is called from DoMetering1Ph() metering function.

4.1.10 DisableGain

This function is called from the metering library during the calibration phase only to calculate calibration coefficients during low-power operation and current gain is disabled. The application should also call this function in the context of ChkVolLvl() if the current RMS voltage is higher than tMETERLIBLPRT1PH_DATA -> VHystLow. These functions recalculate active power (P).

4.1.10.1 Syntax

```
#include "Metering1Ph.h"
void DisableGain (void);
```

4.1.10.2 Arguments

None.

4.1.10.3 Return

None.

4.1.10.4 Description

DisableGain() is called from DoMetering1Ph() metering function.

4.1.11 EnableGain

This function is called from the metering library during the calibration phase only to calculate calibration coefficients after a low-power operation and current gain is enabled. The application should also call this function in the context of ChkVolLvl() if the current RMS voltage is higher than tMETERLIBLPRT1PH_DATA -> VHystHigh.

4.1.11.1 Syntax

```
#include "Metering1Ph.h"
void EnableGain (void);
```

4.1.11.2 Arguments

None.

4.1.11.3 Return

None.

4.1.11.4 Description

EnableGain() is called from DoMetering1Ph() metering function.

4.1.12 CalibMemwrite

These functions are called from the metering library to allow an application to save CalibStruct data structure objects - once the calibration is done.

4.1.12.1 Syntax

```
#include "Metering1Ph.h"
void CalibMemwrite1Ph (void);
#include "Metering3Ph.h"
void CalibMemwrite3Ph (void);
```

4.1.12.2 Arguments

None.

4.1.12.3 Return

None.

4.1.12.4 Description

CalibMemwrite() is called by the DoMetering() metering function after calibration operation is performed.

4.1.13 ProcSamples3Ph

This function Process three-phase each phase voltage, current samples for a phase.

4.1.13.1 Syntax

```
#include "Metering3Ph.h"
void ProcSamples3Ph (void);
```

4.1.13.2 Arguments

Type	Name	Direction	Description
unsigned char	channel	in	Phase number of three phases.

4.1.13.3 Return

None.

4.1.13.4 Description

ProcSamples3Ph() function is utilized to process per phase voltage and current samples of three-phase meter application.

4.1.13.5 Performance

Table 1.

Function	Code size	Stack size	Clock cycles
ProcSamples3Ph	170	20	218

5 Accuracy and performance testing

The performance of the metering library was tested on the one-phase Kinetis M power meter reference design [1]. The MKM35Z512 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 LPRT Metering Algorithm.

Table 2. Accuracy and performance testing

Single-phase KM3x power meter specification	
Type of meter	Single-phase residential
Type of measurement	4-quadrant
Metering algorithm	Low-Power Real-Time (LPRT)
Precision (accuracy)	IS13779 class 0.5 %
Nominal Voltage	240 VAC ± 20 %

Table continues on the next page...

Table 2. Accuracy and performance testing (continued)

Single-phase KM3x power meter specification	
Current Range	0 – 60 A (10 A is nominal current; dynamic range is up to 72 A)
Nominal Frequency	50 Hz ± 5 %
Meter constant (imp/kWh, imp/kVArh)	3200
Functionality	V, A, kW, VAr, VA, kWh (import/export), kVArh (import / export), Hz, power factor, time, date
Voltage sensor	Voltage divider
Current sensor	Shunt resistor down to 200 μΩ (350 μΩ used in this design), Current transformer with 2500:1 turns ratio
Energy output pulse interface	Two red LEDs (active and reactive energy)
User interface (HMI)	8 x 15 segment LCD, one pushbutton
Tamper detection	Two hidden buttons (module area and main cover)
IEC62056-21 infrared interface	9600 / 8-N-1 IR interface
Remote communication modules (optional only) • GPRS	GPRS modules with 1 x SIM card slot, IPv6 capable module
aExternal NVMs • EEPROM • Flash (optional only)	M240M2, 256 KB IS25LQ040B, 512 KB
Internal battery	½ AA, 3.6 V Lithium-Thionyl Chloride (Li-SOCI ₂) 1.2 Ah
Power consumption @ 3.3 V and 22 °C: • Normal mode (powered from mains) • Stand-by mode (powered from a battery) • Power-down mode (powered from a battery)	11.0 mA ¹ 2.2 mA 4.0 μA (both covers closed, no tampering)

NOTE

1. Valid for CORECLK = 12.288 MHz and without any plugin communication module software.

The MKM35Z512 one-phase reference designs are fully calibrated using the test equipment ST6300V2. All power meters were tested according to the IS13779 class 1 (1.0 %) Indian standards for electronic meters.

During the calibration and testing process, the power meter measured electrical quantities generated by the test bench ST6300V2, calculated the active energy, and generated pulses on the output LED. Each generated pulse was equal to the active energy amount in kWh/ imp. The deviations between the pulses generated by the power meter and the reference pulses generated by the test equipment defined the measurement accuracy.

The following figures show the accuracy plot of the NXP MKM35Z512 one-phase smart power meter. The figures indicate the results of the power meter accuracy performed at 25 °C. The accuracy of the measurement for various phase currents, various phase voltages, various frequency values, and the angles between phase current and phase voltage, are shown in the graphs.

Figure 12 shows the accuracy of the active energy measurement after calibration. The x-axis shows the variation of the phase current, and the y-axis denotes the average accuracy of the power meter, computed from five successive measurements. The two bold red lines define the Class 1 (IS13779) accuracy margins for active energy measurement for power factor 1 for this test.

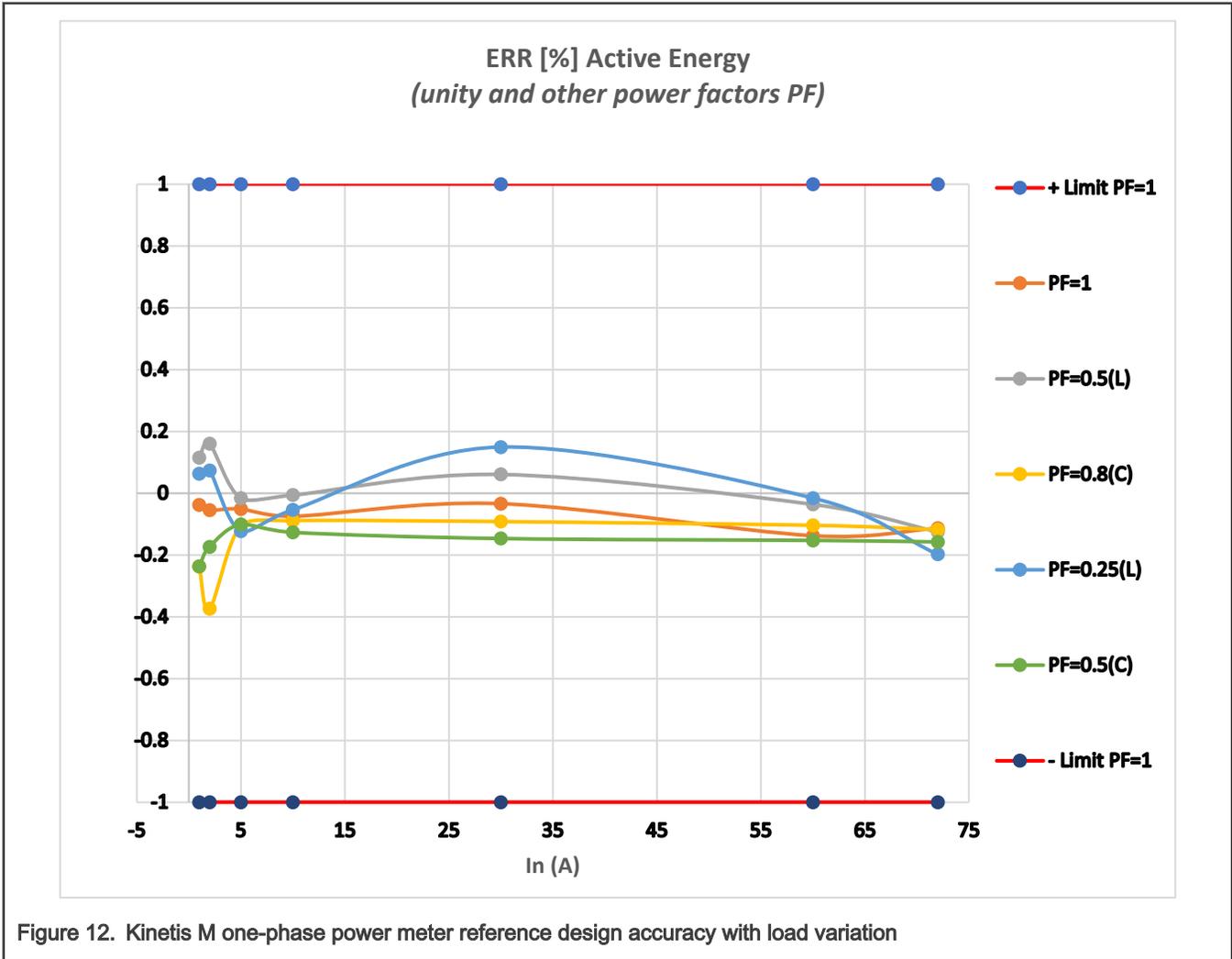


Figure 12. Kinetis M one-phase power meter reference design accuracy with load variation

Figure 13 shows the accuracy of the active energy after calibration. The x-axis shows the variation of the phase voltage, and the y-axis denotes the average accuracy of the power meter, computed from five successive measurements. The two bold red lines define the Class 1 (IS13779) accuracy margins for active energy measurement for power factor 1 for this test.

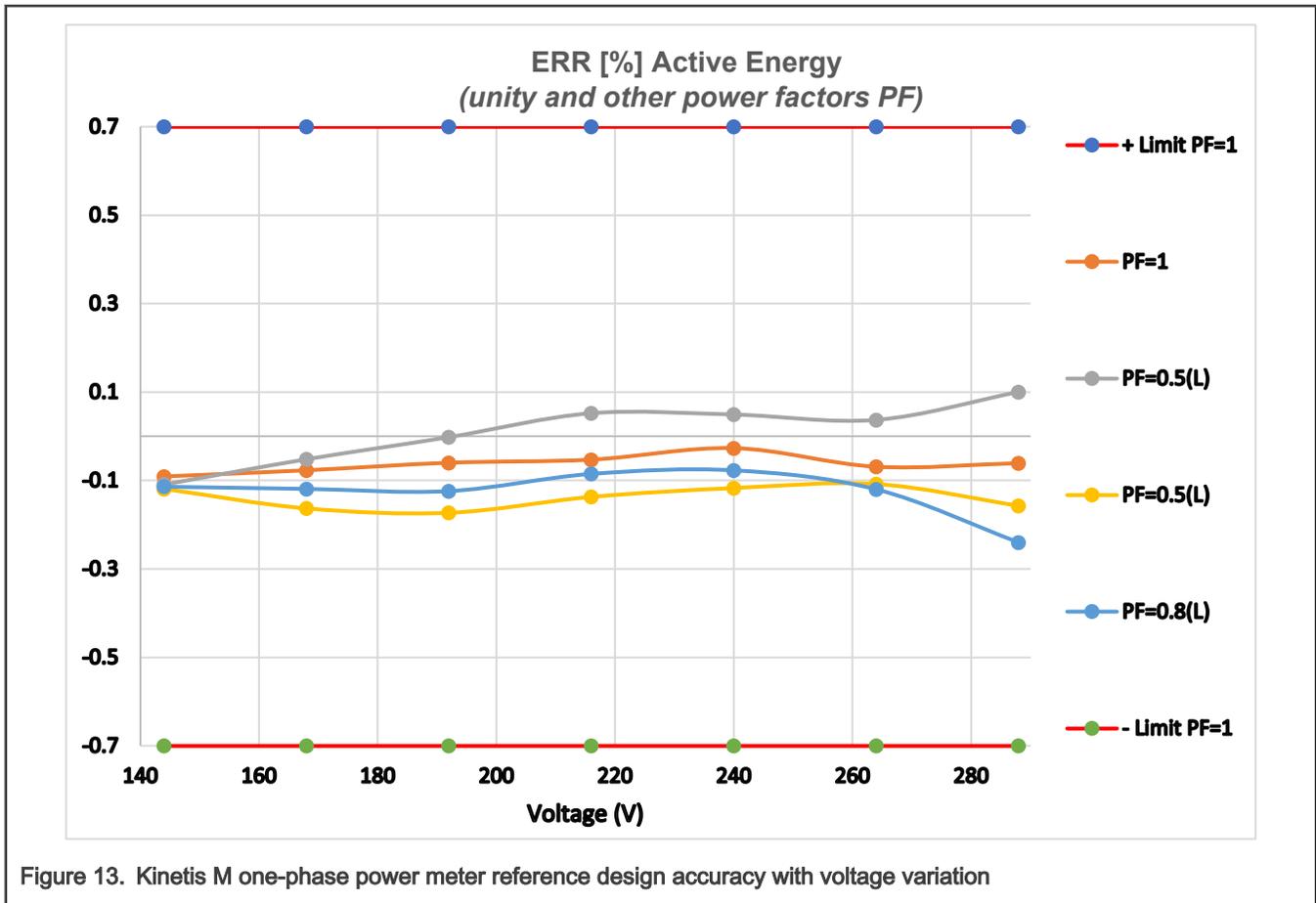


Figure 14 shows the accuracy of the active energy after calibration. The x-axis shows the variation of the frequency, and the y-axis denotes the average accuracy of the power meter, computed from five successive measurements. The two bold red lines define the Class 1 (IS13779) accuracy margins for active energy measurement for power factor 1 for this test.

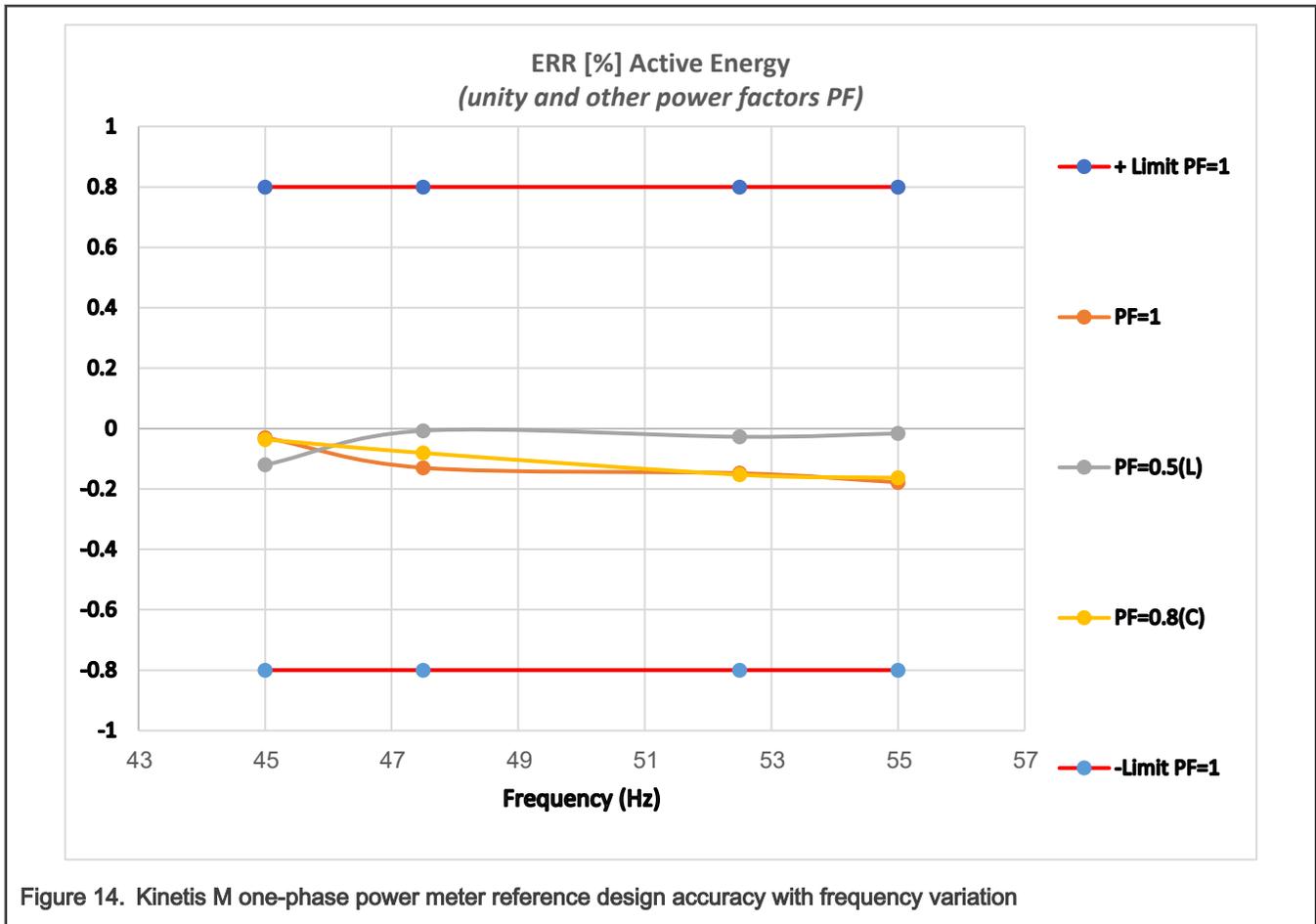


Figure 14. Kinetis M one-phase power meter reference design accuracy with frequency variation

By analyzing the protocols of several MKM35Z512 one-phase power meters, this equipment measures active and reactive energies at all power factors, at 25 °C ambient temperature, and in the current range of 0.1 – 80 A, with the accuracy range of ±0.5 %.

CAUTION

Even though the current range of the power meter is scaled to 72 A, it is not recommended to operate the power meter in the 60 – 72 A range for a longer period, due to heating of the shunt resistor in this current range.

6 Summary

This application note describes a metering library that implements the Low-Power Real-Time (LPRT) 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 LPRT 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 for phase voltage measurement, or a shunt resistor, and a current transformer for phase and neutral current measurement, are supported. The presented algorithms are intended for post-processing instantaneous phase voltage and current samples after phase shift compensation.

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 ±0.5 % in the current range of 0.1 – 72 A.

7 References

The following documents are useful when using the LPRT Algorithm for Metering Applications. These documents can be found in can be found on www.nxp.com.

1. *KM35Z512 based One-Phase Smart Meter Reference Design* (document [AN12837](#))
2. *Filter-Based Algorithm for Metering Application* (document [AN4265](#))
3. *Single Point Meter Calibration process* (document [AN12827](#)).
4. *Kinetis KM35 Sub-Family Reference Manual* (document [KM35P144M75SF0RM](#))

8 Revision history

[Table 3](#) summarizes the changes done to this document since the initial release.

Table 3. Revision history

Revision number	Date	Substantial changes
0	01 June 2021	Initial release

9 C-Header file

```
#include "types.h"
/*****
 * Definitions
 *****/
#define nBUFFERS 2
#define POWERTOENERGY (74.56540444)

typedef enum
{
    CURRENT_PHASE,
    CURRENT_NEUTRAL,
    nCURRENTS
} CURRENTS;

typedef enum
{
    ACT_POWER,
    REACT_POWER,
    APP_POWER,
    nPOWERS
} POWERS;

typedef enum
{
    ACTI_ENERGY,
    REACTIACTI_ENERGY,
    REACTEACTI_ENERGY,
    APPI_ENERGY,
    ACTE_ENERGY,
    APPE_ENERGY,
    nENERGIES
} ENERGIES;
```

```

typedef enum
{
    METREC_FWDED,
    METREC_NET,
} METRECTYPES;

/*! Metering library data structure definition - 1PH METER */
typedef struct
{
    float    Vrms;                /*!< Calculated RMS value of the voltage */
    float    Irms[nCURRENTS];     /*!< Calculated RMS currents - Phase and neutral */
    float    VrmsNoFudge;        /*!< RMS value of voltage w/o compensation */
    float    IrmsNoFudge[nCURRENTS]; /*!< RMS value of currents w/o compensation */
    float    PFMetImax;          /*!< Calculated Power factor value during forced Imax condition */
    float    ActPowers[nCURRENTS]; /*!< Calculated active power */
    float    ActPowersNoFudge[nCURRENTS]; /*!< Calculated active power value w/o compensation */
    float    ReactPowers[nCURRENTS]; /*!< Calculated reactive power */
    float    AppPowers[nCURRENTS]; /*!< Calculated apparent power */
    float    AppPowersNoFudge[nCURRENTS]; /*!< Calculated apparent power w/o compensation */
    float    PowerFactors[nCURRENTS]; /*!< Calculated power factor */
    float    Frequency;          /*!< Calculated frequency */
    uint16   nSamples;           /*!< No of voltage/current samples per second - application to load */
    uint8    MetDue;             /*!< Indicates metrology processing is due for a second */
    uint8    MetOnImax;         /*!< Indication to do metering with Imax condition - application to load */
    int8     ISigns[nCURRENTS];  /*!< Forward/reverse sign of currents, 1 = forward, -1 = reverse */
    uint8    ReactSampleIndex;  /*!< Reactive sample index - internal */
    uint8    CurToUse;          /*!< Indicates which current has higher value - Phase/neutral */
    uint16   nSamps;            /*!< Indicates current sample index ~ [0-(nSamples-1)] */
    int32    VOffset;           /*!< Calculated Voltage offset */
    int32    *pVQCycleSamps;    /*!< 90-degree phase-shifted voltage samples buffer pointer - internal */
    int32    IOfstSum[nCURRENTS]; /*!< Current offset sum - internal */
    int32    IOffsets[nCURRENTS]; /*!< Calculated current offsets */
    int32    VSampsS;          /*!< Offset compensated voltage sample */
    int32    VOfstSum;         /*!< Voltage offset sum - internal */
    uint64   VrmsSums[nBUFFERS]; /*!< Square sums of the voltage samples - internal */
    uint64   IrmsSums[nBUFFERS][nCURRENTS]; /*!< Square sums of the current samples - internal */
    uint64   RelaySenseSums[nBUFFERS]; /*!< Square sums of the relay sense voltage samples - internal */
    int32    VRelayOfstSum;    /*!< Relay sense voltage offset sum - internal */
    int32    VRelayOffset;     /*!< Calculated relay voltage offset */
    float    IBasic;           /*!< Basic current of the meter - application to load */
    float    IMax;             /*!< Maximum current rating of the meter - application to load */
    float    VHystHigh;        /*!< High threshold value of voltage to enable gain for the current measurement - application to load */
    float    VHystLow;         /*!< Low threshold value of voltage to disable gain for the current measurement - application to load */
    float    MaxPower;         /*!< Maximum value of calculated power to restrict power/energy count calculation - application to load */
    int32    ISamps[nCURRENTS]; /*!< Offset compensated current samples */
    float    PhAngles[2];      /*!< Calculated Phase angle */
    uint8    MetRecordingType;  /*!< Power/energy recording type METREC_FWDED/METREC_NET */
    uint8    WBuffer;          /*!< Buffer index of voltage/current samples buffers for a second period */
    uint8    FirstTime;        /*!< internal use */
    uint8    IsGainEnabled;     /*!< Returns current gain is status. TRUE = enabled, FALSE = disabled */
    uint8    LastGainStatus;    /*!< Related to current gain status - internal */
}

```

```

    uint8  IncEnerPtrs[3];          /*!< Indicated energy pointer types active, reactive and apparent -
internal */
    uint32 MetEnergyCounts[3];     /*!< Calculated energy counts for 50 periods */
    uint64 MetEnergySecCounts[3];  /*!< Calculated energy counts 1 second */
    uint8  DoFundamental;         /*!< Indicate whether fundamental frequency-calculation to be done -
application to load */
    uint8  fSampNo;               /*!< Used for fundamental frequency-related calculations - internal
*/
    uint8  CalibState;           /*!< Holds calibration state CALIBSTATE_IDLE, CALIBSTATE_PROGRESS,
CALIBSTATE_COMPLETE */
} tMETERLIBLPRT1PH_DATA;

#define SWB_SIGN      0xA5
#define SWB_CUR_BOTH (SWB_CUR_PH | SWB_CUR_NU)

/*****
* Prototypes
*****/
extern tMETERLIBLPRT1PH_DATA *pmliblphdata;
extern float VFundamental1Ph;
extern float IFundamental1Ph[nCURRENTS];
extern float VTHD1Ph;
extern float ITHD1Ph[nCURRENTS];

extern float Vrelaysense;

/*****//*!
* @brief   Does the RMS and power calculation per the voltage and current samples.
*          Also, does the offset calculations. Once this function finished all
*          samples processing for 50 periods (50 Hz signal),
*          tMETERLIBLPRT3PH_DATA -> MetDue is set to 'TRUE' value.
* @note    The @ref DoPower1Ph function must be called after 1 set of voltage,
*          current samples are received.
*****/
extern void DoPower1Ph(void);

/*****//*!
* @brief   Re-calculates non-billing parameters once every second or 50
*          sinusoidal wave periods.
* @note    The @ref DoMetering1Ph function must be called after
*          tMETERLIBLPRT1PH_DATA -> MetDue is set to 'TRUE' value. This
*          function internally clears tMETERLIBLPRT1PH_DATA -> MetDue to
*          'FALSE'.
*****/
extern void DoMetering1Ph(void);

/*****//*!
* @brief   Initializes tMETERLIBLPRT1PH_DATA metering structure object with
*          few application specific parameters. Typically to be called once
*          application run cycle.
* @param   nSamples      No of voltage/current samples per second
*          - application to load.
* @param   samplesForOffset No of samples per second to be used for offset
*          calculation.
* @param   pFreqDependentPhErr Application defined Frequency dependent phase
*          correction values.
* @param   doFundamental Application defined initialization if metering library
*          should perform fundamental calculation-related tasks.
* @note    The @ref MeterLibLPRT1Ph_InitParams function must be called
*          typically once in application initialization cycle.
*****/

```

```

*****/
extern void MeterLibLPRT1Ph_InitParams(tMETERLIBLPRT1PH_DATA *mLib,
                                       uint16 nSamples, uint16 samplesForOffset,
                                       float *pFreqDependentPhErr, uint8 doFundamental);

/* Callbacks/hooks exported by metrology library */
*****//*!
* @brief This function is called from the metering library to allow the application
*        to further correct the phase angle to fine-tune power calculation.
* @param currenttouse Current phase number CURRENT_PHASE or CURRENT_NEUTRAL.
* @note The @ref CorrectAppPhAngle1Ph is called from DoMetering1Ph().
*****/
extern void CorrectAppPhAngle1Ph(uint8 currenttouse);

*****//*!
* @brief This function is called from the metering library to allow the application
*        to tune up non-billing parameters as per application need - for
*        example to clear leakage current and related parameters.
* @note The @ref FudgeParameters1Ph is called from DoMetering1Ph().
*****/
extern void FudgeParameters1Ph(void);

*****//*!
* @brief This function is called from the metering library to allow the application
*        to check current phase voltage and indicate the metering library
*        to switch to low-power mode measurements of non-billing parameters
*        through tMETERLIBLPRT1PH_DATA -> IsGainEnabled parameter.
* @note The @ref ChkVolLvl is called from DoMetering1Ph().
*****/
extern void ChkVolLvl(void);

*****//*!
* @brief This function is called from the metering library during calibration
*        phase only to calculate calibration coefficients during low-power
*        operation and current gain are disabled. The application should also call
*        this function in the context of ChkVolLvl() if the current RMS
*        voltage is higher than tMETERLIBLPRT1PH_DATA -> VHystLow.
* @note The @ref DisableGain is called from DoMetering1Ph().
*****/
extern void DisableGain(void);

*****//*!
* @brief This function is called from the metering library during calibration
*        phase only to calculate calibration coefficients after low-power
*        operation and current gain are enabled. The application should also call
*        this function in the context of ChkVolLvl() if the current RMS
*        voltage is higher than tMETERLIBLPRT1PH_DATA -> VHystHigh.
* @note The @ref EnableGain is called from DoMetering1Ph().
*****/
extern void EnableGain(void);

*****//*!
* @brief This function is called from the metering library to allow the application
*        to save CalibStruct1Ph data structure object - once after the
*        calibration is done.
* @note The @ref CalibMemwrite1Ph is called whenever calibration is
*        performed.
*****/
extern void CalibMemwrite1Ph(void);

```

```
#endif /* __METERING1PH_H */
```

10 Test application

```
#include <math.h>

#include "fraclib.h"
#include "Metering1Ph.h"
#include "Calibration1Ph.h"
#include "pin_mux.h"
#include "board.h"

#define _PI 3.14159265358979323846 /* pi */
#define NSAMPLES 3000 /*!< No of samples per second */
#define I_MAX 141.421 /*!< Maximal current I-peak in amperes */
#define U_MAX 350.000 /*!< Maximal voltage U-peak in volts */
#define PHASE_PHERR_PERHZ 0.0f
#define NEUTRAL_PHERR_PERHZ 0.0f

/* static data definitions */
static tMETERLIBLPRT1PH_DATA mliblphdata;
tCalibPoint1Ph CalibPoint;
float FreqDependentPhErr[nCURRENTS] = { PHASE_PHERR_PERHZ, NEUTRAL_PHERR_PERHZ};
static volatile frac32 u24_sample, i24_sample;
static double time = 0.0, U_ANGLE = (60.0/180.0)*_PI, I_SHIFT = (-5.5/180.0)*_PI;

void RestoreDefCalib(void);
void AppInterfaceInit(void);

#if defined(__ICARM__)
#pragma diag_suppress=Pa082
#endif

void FudgeParameters1Ph(void)
{
}

void CorrectAppPhAngle1Ph(uint8 currenttoUse)
{
}

void ChkVolLvl(void)
{
}

void EnableGain(void)
{
    mliblphdata.IsGainEnabled = TRUE;
}

void DisableGain(void)
{
    mliblphdata.IsGainEnabled = FALSE;
}

void RestoreDefCalib(void)
{
}
```

```

CalibStruct1Ph.FrequencyCoeff = 1.00023889;

CalibStruct1Ph.VCoeff = 7.517977528089888e-7;
CalibStruct1Ph.ICoeff[0] = 3.0005420511E-7;
CalibStruct1Ph.ICoeff[1] = 3.0005420511E-7;
CalibStruct1Ph.ActPowerCoeff[0] = 3.0870693433E-13;
CalibStruct1Ph.ActPowerCoeff[1] = 3.0870693433E-13;
CalibStruct1Ph.PhAngle[0] = 2.24286317E-2;
CalibStruct1Ph.PhAngle[1] = -3.33993435E-2;
CalibStruct1Ph.VFCoeff = 5.52673412E-13;
CalibStruct1Ph.IFCoeff[0] = 1.12404917E-13;
CalibStruct1Ph.IFCoeff[1] = 6.38489369E-14;
CalibStruct1Ph.ICoeff_LP = 3.0005420511E-7;
CalibStruct1Ph.ActPowerCoeff_LP = 3.0870693433E-13;

CalibStruct1Ph.RTCCompValue = 0;
CalibStruct1Ph.RTCCompInterval = 0;

CalibStruct1Ph.CalibSign = CALIBNOTDONE;
}

void AppInterfaceInit(void)
{
    mliblphdata.IMax = 60.0;
    mliblphdata.IBasic = 10.0;
    mliblphdata.VHystHigh = 160.0;
    mliblphdata.VHystLow = 155.0;
    mliblphdata.MaxPower = 240.0 * mliblphdata.IMax * 1.6;
}

void CalibMemwrite1Ph(void)
{
    CalibStruct1Ph.CalibSign = CALIBDONE;
}

uint32 NoOfMetDue = 0;
void main (void)
{
    BOARD_InitPins();
    BOARD_BootClockRUN();

    MeterLibLPRT1Ph_InitParams(&mliblphdata, NSAMPLES, (NSAMPLES*20)/100, FreqDependentPhErr, FALSE);
    mliblphdata.Frequency = 50.0f;
    AppInterfaceInit();
    RestoreDefCalib();
    mliblphdata.CalibState = CALIBSTATE_PROGRESS;
    CalibPoint.Voltage = 240.0;
    CalibPoint.Current = 10.0;
    CalibPoint.PhAngle = 1.04719;
    CalibPoint.PowerFactor = 0.5;
    CalibPoint.Frequency = 50.0;
    /* calibration can be done for CALIB_PHASE or CALIB_NEUTRAL - one at a time */
    CalibPoint.CalibPhase = CALIB_PHASE;
    CalibStruct1Ph.CalibSign = CALIBNOTDONE;

    while (1)
    {
        /* calculate phase voltage and phase current waveforms */
        time = time+(1.0/NSAMPLES);
        u24_sample = FRAC24(((sin(2*_PI*50.0*time+U_ANGLE)*240.0*sqrt(2)+0.0)/U_MAX));
    }
}

```

```
mliblphdata.VOfstSum += u24_sample;
mliblphdata.VSampsS = u24_sample - mliblphdata.VOffset; /* Previous offset */
i24_sample = FRAC24(((sin(2*_PI*50.0*time+I_SHIFT)*10.0*sqrt(2)+0.0)/I_MAX));
mliblphdata.IOfstSum[0] += i24_sample;
mliblphdata.IOfstSum[1] += i24_sample;
mliblphdata.ISamps[0] = i24_sample - mliblphdata.IOffsets[0]; /* Previous offset */
mliblphdata.ISamps[1] = i24_sample - mliblphdata.IOffsets[1]; /* Previous offset */

DoPower1Ph();

if (mliblphdata.MetDue == TRUE)
{
    NoOfMetDue++;
    /* Do metering calculations */
    DoMetering1Ph();

    if (mliblphdata.CalibState == CALIBSTATE_PROGRESS)
    {
        /* comes to this place until CALIBSTATE_COMPLETE */
        DoCalibration1Ph(&CalibPoint);
    }
}
}
```

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

Limited warranty and liability— Information in this document is provided solely to enable system and software implementers to use NXP 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 reserves the right to make changes without further notice to any products herein.

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.

Right to make changes - NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Security— Customer understands that all NXP products may be subject to unidentified or documented vulnerabilities. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer's applications and products. Customer's responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer's applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately. Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP. NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

NXP, the NXP logo, NXP SECURE CONNECTIONS FOR A SMARTER WORLD, COOLFLUX, EMBRACE, GREENCHIP, HITAG, ICODE, JCOP, LIFE, VIBES, MIFARE, MIFARE CLASSIC, MIFARE DESFire, MIFARE PLUS, MIFARE FLEX, MANTIS, MIFARE ULTRALIGHT, MIFARE4MOBILE, MIGLO, NTAG, ROADLINK, SMARTLX, SMARTMX, STARPLUG, TOPFET, TRENCHMOS, UCODE, Freescale, the Freescale logo, AltiVec, CodeWarrior, ColdFire, ColdFire+, the Energy Efficient Solutions logo, Kinetis, Layerscape, MagniV, mobileGT, PEG, PowerQUICC, Processor Expert, QorIQ, QorIQ Qonverge, SafeAssure, the SafeAssure logo, StarCore, Symphony, VortiQa, Vybrid, Airfast, BeeKit, BeeStack, CoreNet, Flexis, MXC, Platform in a Package, QUICC Engine, Tower, TurboLink, EdgeScale, EdgeLock, eIQ, and Immersive3D are trademarks of NXP B.V. All other product or service names are the property of their respective owners. AMBA, Arm, Arm7, Arm7TDMI, Arm9, Arm11, Artisan, big.LITTLE, Cordio, CoreLink, CoreSight, Cortex, DesignStart, DynamIQ, Jazelle, Keil, Mali, Mbed, Mbed Enabled, NEON, POP, RealView, SecurCore, Socrates, Thumb, TrustZone, ULINK, ULINK2, ULINK-ME, ULINK-PLUS, ULINKpro, μ Vision, Versatile are trademarks or registered trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. The related technology may be protected by any or all of patents, copyrights, designs and trade secrets. All rights reserved. Oracle and Java are registered trademarks of Oracle and/or its affiliates. The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org. M, M Mobileye and other Mobileye trademarks or logos appearing herein are trademarks of Mobileye Vision Technologies Ltd. in the United States, the EU and/or other jurisdictions.

© NXP B.V. 2021.

All rights reserved.

For more information, please visit: <http://www.nxp.com>

For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 01 June 2021

Document identifier: AN13259

