User's Guide

User Guide for E-Metrology Library for Digital Power Applications With F28P55x and F28P65x

ABSTRACT

Modern electrical systems demand high-precision metrology and comprehensive power quality (PQ) analyzes; both tasks that challenge conventional microcontrollers with real-time processing of complex waveforms and computationally intensive algorithms. This white paper introduces the advanced metrology software library architected for Texas Instruments' C2000™ real-time microcontrollers, specifically leveraging the integrated, 32-bit floating point unit (FPU32), trigonometric math unit (TMU), and inherent DSP capabilities of these microcontrollers to overcome these challenges. The library delivers a rich set of high-accuracy measurements—including true RMS voltage and current, multiphase active, reactive, and apparent power and energy, power factor, line frequency, sag and swell detection, and total harmonic distortion (THD) based on precise FFT —with all measurements performed using high-precision 32-bit floating-point arithmetic. TI's C2000 MCUs offer excellent performance with the efficient split-architecture of these devices, which separates time-critical background data accumulation (conceptually driven by ADC interrupts) from complex foreground calculations. This work highlights the exceptional performance and accuracy of C2000 MCUs for demanding metrology and PQ applications when compared to general-purpose MCUs (for example, native floating-point FFT execution and dedicated DSP instructions), and the distinct advantages of choosing TI's C2000 MCUs over general-purpose MCUs.

Table of Contents

1 Introduction 2
2 Library Notice 3
3 Library Architecture 4
3.1 Background Processing 4
3.2 Foreground Processing 4
3.3 Data Transfer Mechanism 4
4 Metrology Calculations 5
5 Configuration and Data Structures 5
6 Calibration Module 5
7 User Specific Configurations 5
7.1 Debug Mode 5
7.2 ADC Mode 6
7.3 Switching Between Modes 8
7.4 THD Algorithm Selection 9
7.5 Topology Selection 9
7.6 Enable Features 10
7.7 System Parameters 10
7.8 Scaling Factors Per Phase 10
8 Running the Example 11
8.1 Accessing the Parameters Per Phase 11
8.2 Accessing the Total System Measurements 12
8.3 Accessing the Phase Status Flags 12

Trademarks

1 Introduction

The evolution of electrical infrastructure, including smart grids, renewable energy integration, and sophisticated industrial automation, places increasingly stringent demands on the accuracy and comprehensiveness of electrical parameter measurement. Precise metrology—the science of measurement—is no longer just about basic energy consumption; metrology now encompasses detailed power quality (PQ) analysis, crucial for maintaining grid stability, equipment longevity, and operational efficiency. Parameters, such as RMS voltage and current, active and reactive power, power factor, frequency, and harmonic distortion, provide vital insights into the health and performance of electrical systems. However, implementing a robust metrology and PQ analysis process presents significant technical hurdles. Modern electrical loads are often non-linear, leading to distorted current and voltage waveforms rich in harmonics. Accurately processing these complex signals in real-time, while performing computationally intensive algorithms like fast fourier transforms (FFTs) for harmonic analysis or precise dot-product accumulations for true RMS and power calculations, strains the capabilities of traditional microcontrollers. Furthermore, achieving the high precision required by metrology standards (often necessitating floating-point arithmetic) adds another layer of complexity, particularly on resource-constrained embedded platforms.

2 Library Notice

This library is provided as reference code to demonstrate the metrology algorithm concept on Texas Instruments' C2000 microcontrollers. The library is not a production-ready solution and is not to be used for shipping a product without substantial modification and validation.

The code ships in DEBUG mode by default, providing the metrology engine with a mathematically perfect sine wave generated in software. This default is intentional, as DEBUG mode allows the user to verify the algorithms and explore the measurement outputs without requiring hardware. However, DEBUG mode does not inform the user about how the system behaves with real analog-to-digital converter (ADC) inputs, real current transformers, real voltage dividers, or a real mains signal.

The following lists the areas that require engineering work specific to the hardware of the user before the library can be used with a power measurement system:

- **ADC Configuration** – The on-chip ADC must be configured in the system configuration (**SysConfig**) menu to match the channel assignments, sampling rate, trigger source, and acquisition window of the board used. The default configuration targets a specific LaunchPad™ Development Kit pinout, and this configuration is not correct for a custom board unless changes are made to both the *.syscfg* file and the *adc_config.h* file.
- **Analog Front-End Scaling** – The values of the **PHASE_x_VOLTAGE_SCALE_FACTOR** and **PHASE_x_CURRENT_SCALE_FACTOR** fields in the *template.h* template are placeholders. The following list of values must be derived from the actual hardware used:
  - Voltage divider ratio
  - Current transformer
  - Shunt resistor
  - Operational amplifier (op-amp) gain
  - ADC reference voltage

  Using incorrect scale factors produce readings that are numerically plausible but incorrect.
- **ADC Calibration** – Two-point calibration (DC offset and gain) must be performed on the actual hardware as described in Section 7.2.2.
- **Phase Correction** – Current transformers and analog filter stages introduce a phase shift between the voltage and current measurement paths. This phase shift directly affects the accuracy of the active power and power factor.
- **External ADC Support** – If the design uses an external sigma-delta or successive approximation register (SAR) ADC, rather than the on-chip ADC, the **ADC_readMetrologyData()** function must be completely rewritten to interface with the ADC driver. The library architecture supports the rewriting of this function, however, nothing in the *adc_config.c* file works out of the box with a different ADC.
- **ISR Timing** – The metrology background task must be executed at precisely SAMPLE_RATE samples per second. The default uses the **CPUTimer0** task. If the system uses a different timer, direct memory access (DMA) trigger, or ADC end-of-conversion interrupt, the interrupt service routine (ISR) wiring in the *metrology_main_file.c* file must be changed accordingly.

None of the default values are validated against a real hardware reference. These values exist to allow the DEBUG mode demo to produce outputs that look numerically reasonable. Treat every number in the *template.h* template and the *adc_config.h* file as a starting point, not a specification.

3 Library Architecture

To meet the stringent requirements of real-time, continuous, power monitoring and prevent the loss of critical data, the metrology library employs an efficient split-processing architecture. This design decouples time-sensitive tasks from less frequent, more computationally intensive calculations. The design also optimizes resource utilization and maintains system responsiveness. The architecture is broadly divided into two main processing layers:

- Background processing
- Foreground processing

3.1 Background Processing

Background processing is executed at a high frequency, typically synchronized with the analog-to-digital converter (ADC) sampling rate. The background processing tasks are handled within an interrupt service routine (ISR). The primary responsibilities of this layer include:

- **Raw Sample Acquisition**: Reading the latest voltage and current samples from the ADC.
  - Essential pre-processing: Performing minimal, time-critical operations on the raw samples, such as DC estimation and filtering to remove any DC bias.
- **Core Accumulations**: Calculating and accumulating fundamental dot products (for example, the sum of squared voltage samples, the sum of squared current samples, and the sum of voltage-current products) over a defined measurement window (typically one or more main cycles). These accumulations form the basis for most subsequent metrology calculations.
- **Data Buffering and Synchronization**: Managing data buffers and preparing accumulated data for safe transfer to the foreground tasks.

The key design principle for background processing is to keep the ISR execution time as short as possible for high sampling fidelity and to allow the system to respond to other critical interrupts.

3.2 Foreground Processing

Foreground processing operates at a lower frequency within the main application loop. The foreground tasks are triggered once the background process has accumulated a complete block of data. This layer handles the more computationally intensive calculations, including:

- **Retrieving Accumulated Data**: Safely reading the consistent set of dot products and other relevant data logged by the background task.
- **Applying Calibration**: Correcting the raw accumulated values using predetermined calibration coefficients to compensate for sensor inaccuracies and analog front-end variations.
- **Calculating Final Parameters**: Computing the full suite of metrology parameters, such as RMS voltage and current, active, reactive, and apparent power, power factor, frequency, THD, and sag and swell events, using the calibrated data.
- **Energy Accumulation**: Integrating power values over time to calculate energy consumption.
  - Updating results: Making the final, calibrated measurements available to the application for display, logging, or communication.

3.3 Data Transfer Mechanism

A crucial element of this split architecture is the robust data transfer mechanism between the background ISR and the foreground tasks. The library employs a synchronized pass-off technique, often referred to as double-buffering, for data integrity. The process is as follows:

1. The background ISR continuously updates a primary set of accumulators with per-sample data.
2. Once a complete measurement block (for example, a defined number of samples or main cycles) is accumulated, the background task copies the current values from these primary accumulators to a secondary, *logged* set of variables.
3. A status flag is then set by the background task, signaling to the foreground task that a new, consistent block of data is ready in the logged variables.
4. The foreground task, running in the main application loop, periodically checks this status flag. Upon detecting that the flag is set, the foreground task knows that a complete and consistent set of data is

available in the *logged* accumulators. The foreground task then proceeds to use this data to complete calculations.

4 Metrology Calculations

The actual metrology calculations take place in the *meterology_caluclations.c* file.

- This module contains the library of mathematical functions that implement the metrology algorithms.
- The module takes inputs such as accumulated dot products, sample counts, and calibration data.
- The module outputs the calculated electrical parameters (for example, **calculateRMSVoltage, calculateActivePower**, and **calculateFrequency_PLL**).

5 Configuration and Data Structures

These files define constants (for example, nominal voltage or frequency, or sample rates), preprocessor macros for feature enablement (for example, **HARMONICS_SUPPORT** and **NEUTRAL_MONITOR_SUPPORT**), and the critical data structures that hold all information related to metrology.

- **metrologyData**: The top-level global structure, an instance of which holds all data for all phases, neutral, totals, and system status.
- **phaseMetrology** or **neutralMetrology**: Structures within **metrologyData** containing parameters (params) and readings (readings) for each phase and the neutral line.
  - **params**: Stores configuration, intermediate accumulations from background (like dotProd arrays), calibration-derived values, and state variables (for example, for PLL, and sag and swell).
  - **readings:** Stores the final calculated metrology values (RMS, power, PF, and so forth).
- **calibrationData** (*metrology_nv_structs.h*): Holds the calibration coefficients.

6 Calibration Module

The calibration module manages the loading, application, and storage (conceptual) of calibration data.

- **Metrology_setup_init** (or similar in *metrology_setup.c*): Initializes data structures and applies the initial calibration settings like phase corrections.

7 User Specific Configurations

The user can configure the metrology library to calculate the metrology parameters so that the parameters meet the requirements of the application in use. All user-facing configurations are contained in two files:

- *template.h*
- *adc_config.h*

All other library files must be left unchanged.

The library supports two operating modes that are selected at compile time:

- DEBUG mode
- ADC mode

The mode is controlled by the DEBUG preprocessor symbol in the *template.h.* file.

7.1 Debug Mode

DEBUG mode replaces real ADC hardware with a software signal generator. DEBUG mode is intended for initial bring-up, algorithm validation, and verifying that the metrology engine produces the correct results before any hardware is connected. No ADC hardware or analog front-end is required.

7.1.1 The Function of Debug Mode

When DEBUG mode is defined, the **cpuTimer0ISR** function calls **get_debugData()** function instead of the **ADC_readMetrologyData()** function. The **get_debugData()** function serves pre-computed samples from two look-up tables—one for voltage and one for current—that are filled at start-up by the **debugInit()** function.

The **debugInit()** function populates the voltage[] and current[] arrays (each of the length: **SAMPLES_PER_CYCLE** = 160 samples for a 50Hz / 8000Hz system) using the C standard **sinf()** function evaluated at uniformly spaced phase angles over one full cycle [0, 2*PI]:

- voltage[i] = (V_RMS × SQRT2) / (V_SCALING) × sinf(t)
- current[i] = ((I_RMS × SQRT2) / I_SCALING) × sinf(t - phaseDiff)

Where t = (i / SAMPLES_PER_CYCLE) × 2*PI and phaseDiff = acosf(POWER_FACTOR).

The samples are normalized by the **V_SCALING** and **I_SCALING** macro variables defined in the template.h file before storage, so that the values fed into the metrology engine are in the same per-unit format as the real ADC samples. The metrology engine then applies the calibration scale factors (**VscaleFactor** and **IscaleFactor**) to reconstruct physical units. In DEBUG mode, these scale factors are overridden in the **main()** function in the *metrology_main_file.c* file to match the **V_SCALING** and **I_SCALING** parameters so that the displayed readings equal the **V_RMS** and **I_RMS** values exactly.

Each call to get the **debugData()** function advances a set of index pointers (**voltageIndex[]** and **currentIndex[]**) through the circular look-up tables, so the ISR sees a continuous periodic waveform at the correct sample rate.

For multiphase topologies, each phase is offset from Phase A by **PHASE_INCREMENT** = 53 samples (equivalent to 53/160 × 360 = 119.25 degrees inter-phase spacing) by initializing **voltageIndex[1]** and **voltageIndex[2]** at different starting positions.

7.1.2 How to Configure DEBUG Mode Parameters

All DEBUG mode parameters are defined at the top of the *metrology_main_file.c* file inside the **#ifdef DEBUG** block:

- **V_RMS** – Root mean square (RMS) voltage of the generated sine wave in volts. Default: 120.0.
- **I_RMS** – RMS current of the generated sine wave in amps. Default: 10.0.
- **POWER_FACTOR** – Power factor of the simulated load (0.0 to 1.0). Sets the phase angle between voltage and current as phaseDiff = acosf(POWER_FACTOR). Default: 0.5 (lagging, 60° shift).
- **V_SCALING** – Voltage scale factor used to normalize the generated samples. Must match **PHASE_A_VOLTAGE_SCALE_FACTOR** in the *template.h* template. Default: 1585.2.
- **I_SCALING** – Current scale factor used to normalize the generated samples. Must match **PHASE_A_CURRENT_SCALE_FACTOR** in the *template.h* template. Default: 125.0.
- **SAMPLES_PER_CYCLE** – Number of samples per mains cycle in the look-up table. Must equal **SAMPLE_RATE | MAINS_NOMINAL_FREQUENCY**. Default: 160 (8000Hz / 50Hz).
- **PHASE_INCREMENT** – Starting index offset between phases in the look-up table. Default: 53.

Injecting Harmonics into the Generated Signal

To test the total harmonic distortion (THD) calculations, harmonic content can be injected into the generated waveform by defining **INJECT_HARMONIC** inside the **#ifdef DEBUG** block in the *metrology_main_file.c* file. When **INJECT_HARMONIC** is defined, an additional sinusoidal component is added to both the voltage and current look-up tables during the **debugInit()** function:

- voltage[i] ± (V_RMS × SQRT2) / (V_SCALING) × HARMONIC_PERCENT × sinf(HARMONIC_ORDER × t)
- current[i] ± (I_RMS × SQRT2) / (I_SCALING) × HARMONIC_PERCENT × sinf(HARMONIC_ORDER × t - phaseDiff)

- **HARMONIC_ORDER** – Integer order of the harmonic to inject (for example, 5 injects the 5th harmonic at 250Hz). Default: 5.
- **HARMONIC_PERCENT** – Amplitude of the injected harmonic as a fraction of the fundamental amplitude (for example, 0.50 = 50%). The expected THD result is approximately equal to this value. Default: 0.50.

Multiple harmonics can be injected by adding further terms manually inside the loop in the **debugInit()** function.

7.2 ADC Mode

In ADC mode the library reads real voltage and current samples from the on-chip ADC peripherals configured using the **SysConfig** menu. Two sub-cases exist:

- Using the on-chip ADC directly

- Replacing the on-chip ADC with an external ADC

7.2.1 Using the On-Chip ADC

The on-chip ADC is configured entirely through the system configuration (*.syscfg*) file included in the project. Open the *.syscfg* file in the **SysConfig** menu to configure:

- ADC module selection (ADCA for voltage, ADCB for current by default)
- SOC (start of conversion) trigger source – Set to the same **CPUTimer0** period used by the metrology ISR
- Sampling window duration – Set to meet the ADC input circuit settling time
- Acquisition window and EOC (end of conversion) interrupt if required

After saving the *.syscfg* file, the **SysConfig** menu regenerates the ***board.h*** header file and ***board.c*** source file with the peripheral names (**myADC0_RESULT_BASE**, **myADC0_SOC0**, and so forth) that the library uses.

Open the ***adc_config.h*** header file and assign the generated names to the library macros for each phase:

Phase A (always required):

- **ADC_VA_RESULT | ADC_VA_SOC** – Result base and SOC number for the Phase A voltage channel.
- **ADC_IA_BASE | ADC_IA_RESULT | ADC_IA_SOC** – Base address, result base, and SOC number for the phase A current channel.

Phase B (required for **TWO_PHASE_SUPPORT**, **THREE_PHASE_SUPPORT**, or **SPLIT_PHASE_SUPPORT**):

- **ADC_VB_RESULT | ADC_VB_SOC** – Phase B voltage channel.
- **ADC_IB_RESULT | ADC_IB_SOC** – Phase B current channel.

Phase C (required for **THREE_PHASE_SUPPORT** only, currently commented out, remove comment **THREE_PHASE_SUPPORT** when a third ADC is added in the *SysConfig* GUI):

- **ADC_VC_RESULT | ADC_VC_SOC** – Phase C voltage channel. A third ADC SOC must be added in the **SysConfig** menu first.
- **ADC_IC_RESULT | ADC_IC_SOC** – Phase C current channel.

7.2.2 Two Point Calibration

The on-chip ADC requires two calibration steps to produce accurate measurements:

- Offset calibration
- Gain calibration

Together these form a two-point calibration that removes the DC bias of the analog front-end and corrects for gain error introduced by the resistor divider, current transformer, or operational amplifier chain.

7.2.2.1 Step 1 – Offset Calibration (Zero-Input Point)

With no AC signal applied to the ADC inputs:

1. Call the **ADC_calibrateOffsets()** function once at start-up before the **App_startDataCollection()** function.
2. This function averages 1000 consecutive ADC readings on the voltage and current channels to find the DC midpoint:
   - **ADC_calibrateOffsets();** // call with no AC input connected
3. The function stores the measured averages in the internal variables **g_voltageOffset** and **g_currentOffset**. These averages are then subtracted from every subsequent ADC reading in the **ADC_readNormalized()** function. The theoretical midpoint for a 12-bit ADC with a bipolar signal centered at Vref/2 (1.65V) is 2048, but component tolerances shift this midpoint in practice.
4. If the **ADC_calibrateOffsets()** function is not called, the default values of **ADC_VOLTAGE_DC_OFFSET** and **ADC_CURRENT_DC_OFFSET** are used, as defined in the *adc_config.h* file (both default to 2048.0).

7.2.2.2 Step 2 – Gain Calibration (Full-Scale Point)

1. Apply a known AC reference signal and compare the library output to the known value.
2. Adjust **ADC_VOLTAGE_GAIN** and **ADC_CURRENT_GAIN** in the *adc_config.h* file using:
   - gain_new = gain_old × (measured_reading / known_reference)

  - For example, if the library reports 118.2V RMS against a known 120.0V reference: gain_new = 2056.8 × (120.0 / 118.2) = 2087.1
3. Repeat for current using **ADC_CURRENT_GAIN**. The ***ADC_readNormalized()*** function applies these gains as:
   - normalized = (raw_count - dc_offset) / gain
   - The resulting normalized value is a per-unit float in the same range as the debug signal generator output, which the metrology engine then scales by VscaleFactor / IscaleFactor (from template.h) to produce physical units.

Phase Correction (for current transformer phase shift):

Current transformers and analog filter stages introduce a small but measurable phase shift between the voltage and current channels that directly affect power factor and power accuracy. This is corrected by the **phaseOffset** field in **currentSensorCalibrationData** (*metrology_nv_structs.h*).

The default phase correction values are defined in ***metrology_calibration_defaults.h*** :

- **DEFAULT_BASE_PHASE_A_CORRECTION** – Phase correction for phase A in units of: Microseconds × **SAMPLE_RATE** × 256. Default: 0.0 (appropriate for DEBUG mode and appropriate sensors). For a real CT with a 2.4 microsecond lag, this is: –2.4e-6 × 8000 × 256.

Acceptable range is approximately –125 microseconds to +125 microseconds. Measure the actual phase error using a precision power analyzer at unity power factor and set the correction accordingly.

7.2.3 Using an External ADC

If the on-chip ADC is replaced with an external ADC (for example a sigma-delta device connected over SPI), the following changes are required:

1. In the *adc_config.c* file – Replace the **ADC_readNormalized()** function calls inside the **ADC_readMetrologyData()** function with calls to the external ADC driver. The function must write a normalized float (centered at zero, full scale approximately ±1.0) into **workingData→rawVoltageData[PHASE_x]** and **workingData→rawCurrentData[PHASE_x]**. The normalization convention is:

   normalized = (raw_sample - dc_offset) / gain

   Where dc_offset and gain are the same two-point calibration constants described above, applied to the raw output counts of the external ADC.
2. In the *adc_config.h* file – Remove or replace the **ADC_VA_RESULT | ADC_VA_SOC** macro definitions since the names generated in the **SysConfig** menu no longer apply. Define whatever constants the driver requires instead. **ADC_VOLTAGE_DC_OFFSET**, **ADC_CURRENT_DC_OFFSET**, **ADC_VOLTAGE_GAIN**, and **ADC_CURRENT_GAIN** remain valid and must be set to match the resolution and front-end gain of the external ADC.
3. In the *.syscfg* file – Remove the ADCA and ADCB peripheral instances if the on-chip ADC is not used, to avoid consuming memory and clock resources.
4. In the *metrology_main_file.c* file – If the external ADC uses a data-ready interrupt rather than a timer trigger, move the **ADC_readMetrologyData()** and **Metrology_perSampleProcessing()** function calls from the **cpuTimer0ISR** functions into the external ADC interrupt handler instead. Verify that the ISR executes at the same rate as SAMPLE_RATE.
5. In the *template.h* file – The **PHASE_x_VOLTAGE_SCALE_FACTOR** and **PHASE_x_CURRENT_SCALE_FACTOR** macro must still reflect the physical scaling of the external ADC analog front-end, as the metrology engine uses these macros to convert normalized samples to volts and amps.

7.3 Switching Between Modes

To switch from DEBUG mode to the real ADC input, comment out or remove the *#define* DEBUG line near the top of *metrology_main_file.c* file:

//#define DEBUG

The ISR then calls the **ADC_readMetrologyData()** function instead of the **get_debugData()** function, and the calibration scale factor overrides the **main()** function under the **#ifdef DEBUG** macro (which serves as a hook to shift between modes) no longer applies. The scale factors from the *template.h* file are used directly.

7.4 THD Algorithm Selection

The library offers three methods for THD computation, selected at compile time by the **HARMONICS_SUPPORT** and **USE_GOERTZEL_THD** macros in the *template.h* template:

- Method 1 – Based on PLL (default when the HARMONICS_SUPPORT macro is not defined)
- Method 2 – Goertzel DFT (when the HARMONICS_SUPPORT and USE_GOERTZEL_THD macros are both defined)
- Method 3 – 2048-Point Real FFT (when the HARMONICS_SUPPORT macro is defined but the USE_GOERTZEL_THD macro is not)

7.4.1 Method 1 – Based on PLL (Default When the HARMONICS_SUPPORT Macro is Not Defined)

Method one computes THD using the relationship:

$$THD = \sqrt{RMS\_total^2 - RMS\_fundamental^2} / RMS\_fundamental$$

Where the RMS_total is computed from the raw waveform dot product and the RMS_fundamental is extracted using a PLL that tracks the 50Hz or 60Hz component. This method is computationally lightweight but has an inherent error floor of ≅0.1% for clean sine inputs due to a window mismatch between the zero-crossing-triggered integration window and the PLL cycle boundary. Results are returned in the **voltageTHD** and **currentTHD** fields of the **phaseReadings** structure.

Use method one when memory is constrained or THD is only a qualitative indicator.

7.4.2 Method 2 – Goertzel DFT (When the HARMONICS_SUPPORT and USE_GOERTZEL_THD Macros are Both Defined)

Method two uses the Goertzel algorithm to compute the DFT of individual harmonic bins (2nd through 50th harmonic) over a 1600-sample window (10 cycles at 50Hz / 8000Hz sample rate, or 12 cycles at 60Hz). The Goertzel algorithm is a recursive filter that produces the same result as an FFT for a single frequency bin, but with lower memory overhead and no spectral leakage from window functions. THD is then computed as:

$$THD = \sqrt{sum(H2^2 + H3^2 + ... + H50^2)} / H1$$

Where H1 is the fundamental amplitude and H2...H50 are the harmonic amplitudes. Results are returned in the **vTHDfft** and **iTHDfft** fields of the **phaseReadings** structure.

This method is the recommended method for the production metrology application as the method settles in ten to twelve cycles and meets the Class A accuracy requirements of the IEC 61000-4-30 standard.

7.4.3 Method 3 – 2048-Point Real FFT (When the HARMONICS_SUPPORT Macro is Defined But the USE_GOERTZEL_THD Macro is Not)

Method three uses a full 2048-point real FFT with a Hann window applied before the transform. The FFT window spans approximately 12.8 cycles at 50Hz / 8000Hz (2048 / 8000 × 50 = 12.8). After the FFT, harmonics are grouped according to the IEC 61000-4-7 standard (5Hz bins centered on each harmonic) and THD is computed from the grouped harmonic bins. The results are returned in the **vTHDfft** and **iTHDfft** fields.

This method provides full spectral information but requires significantly more RAM (two 2048-element float arrays for the real and imaginary FFT buffers) and longer settling time than the Goertzel DFT method. Use method three only if full spectrum analysis beyond THD is required.

7.5 Topology Selection

Define exactly one topology macro to match the system that is being measured:

- **SINGLE_PHASE_SUPPORT** – Single phase, Phase A only (default).
- **TWO_PHASE_SUPPORT** – Two phase, Phase A and Phase B.
- **THREE_PHASE_SUPPORT** – Three phase, Phase A, B, and C.
- **SPLIT_PHASE_SUPPORT** – Split phase, Phase A and B with shared neutral.

7.6 Enable Features

Individual measurements can be enabled or disabled by defining or undefining the corresponding macro. The following features are available and must be defined in the *template.h* file:

- **HARMONICS_SUPPORT** – Harmonic analysis based on FFT. Required for THD calculations.
- **NEUTRAL_MONITOR_SUPPORT** – Neutral current monitoring.
- **VRMS_SUPPORT** – RMS voltage.
- **IRMS_SUPPORT** – RMS current.
- **ACTIVE_POWER_SUPPORT** – Active power.
- **REACTIVE_POWER_SUPPORT** – Reactive power.
- **APPARENT_POWER_SUPPORT** – Apparent power.
- **ACTIVE_ENERGY_SUPPORT** – Active energy accumulation.
- **REACTIVE_ENERGY_SUPPORT** – Reactive energy accumulation.
- **POWER_FACTOR_SUPPORT** – Power factor.
- **POWER_FACTOR_ANGLE_SUPPORT** – Power factor angle.
- **FREQUENCY_SUPPORT** – Line frequency.
- **SAG_SWELL_SUPPORT** – Voltage sag and swell detection.
- **FUNDAMENTAL_PARAMETERS** – Enables all fundamental (50Hz or 60Hz component only) measurements.
- **VOLTAGE_THD_SUPPORT** – Voltage THD. Requires the HARMONICS_SUPPORT macro.
- **CURRENT_THD_SUPPORT** – Current THD. Requires the HARMONICS_SUPPORT macro.

When the **THREE_PHASE_SUPPORT** macro is defined, the following are additionally available:

- **LINETOLINE_VOLTAGE_SUPPORT** – Line-to-line voltages.
- **AGGREGATE_POWER_FACTOR** – Aggregate power factor across all phases.
- **CURRENT_VECTOR_SUM** – Current vector sum.
- **PHASE_TO_PHASE_ANGLE_SUPPORT** – Phase-to-phase angle.

7.7 System Parameters

- **SAMPLE_RATE** – ADC sample rate in Hz. Accepted values: 2000, 2930, 3906, 4000, 5859, 6000, 6400, 7812, and 8000. Default: 8000.
- **MAINS_NOMINAL_FREQUENCY** – Nominal mains frequency in Hz (50 or 60). Default: 50.
- **MAINS_MIN_FREQUENCY | MAINS_MAX_FREQUENCY** – Valid frequency range for zero-crossing detection. Default: 40.0Hz to 70.0Hz.
- **MAINS_NOMINAL_VOLTAGE** – Nominal RMS voltage in V. Used as the reference for sag or swell detection. Default: 230.0V.
- **ENERGY_PULSE_CONSTANT** – Energy pulse rate in pulses per kWh. Default: 6400.

7.8 Scaling Factors Per Phase

Scale factors convert the normalized ADC value to physical units and must match the analog front-end hardware (voltage divider ratio, current transformer or shunt ratio, or op-amp gain).

Find these macros in the ***template.h*** file:

- **PHASE_A_VOLTAGE_SCALE_FACTOR** – Voltage in V per normalized ADC unit for Phase A.
- **PHASE_A_CURRENT_SCALE_FACTOR** – Current in A per normalized ADC unit for Phase A.
- **PHASE_A_POWER_SCALE_FACTOR** – Set to the product of voltage and current scale factors.

Phase B and Phase C have equivalent macros (**PHASE_B_*** and **PHASE_C_***) that are active when the corresponding topology is enabled.

The DC estimates (**PHASE_A_VOLTAGE_DC_ESTIMATE** and **PHASE_A_CURRENT_DC_ESTIMATE**) can be set to an approximate DC midpoint to reduce the filter settling time at start-up. Set the DC midpoint to 0.0 if unknown.

AC offsets (**PHASE_A_VOLTAGE_AC_OFFSET**, **PHASE_A_CURRENT_AC_OFFSET**, **PHASE_A_ACTIVE_POWER_OFFSET**, and **PHASE_A_REACTIVE_POWER_OFFSET**) are subtracted from the computed values and are used to trim the residual measurement error after calibration.

8 Running the Example

First, create and run the project in the Code Composer Studio™ (CCS) software:

1. Use the CPU1_FLASH build configuration to build the **energy_metrology_f28p55** or **energy_metrology_f28p65** project in the CCS.
2. Right-select the project after the project is built and select the **DEBUG** button.
3. Start the execution by using the **RUN** button from the CCS debug pop-up window.

Next, add a new expression in CCS with the target halted at a breakpoint or running in real-time mode:

- Open the *Expressions* window – *Window* → *Show View* → *Expressions*
- Select the green plus icon in the *Expressions* pane and select *Add New Expression*.

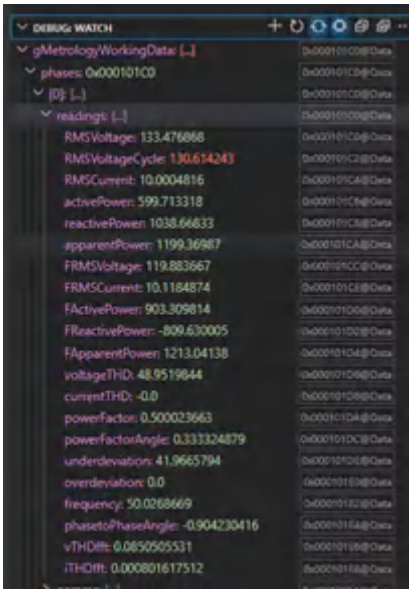

Figure 8-1. Expressions Window in Code Composer Studio (CCS) Software

8.1 Accessing the Parameters Per Phase

The structure of the **gMetrologyWorkingData** structure contains a phases[] array indexed by the **PHASES** enumeration (**PHASE_ONE** = 0, **PHASE_TWO** = 1, **PHASE_THREE** = 2). Each phase has a readings member of type **phaseReadings** containing the final computed values.

Table 8-1. To View the Computed Value Per Phase for the phaseReadings

phaseReadings Name[1]	Phase	Computed Value
gMetrologyWorkingData.phases[0].readings.RMSVoltage	Phase A	RMS voltage[2]
gMetrologyWorkingData.phases[0].readings.activePower	Phase A	Active power
gMetrologyWorkingData.phases[0].readings.powerFactor	Phase A	Power factor
gMetrologyWorkingData.phases[0].readings.frequency	Phase A	Frequency
Based on PLL (When the HARMONICS_SUPPORT macro is not defined)		
gMetrologyWorkingData.phases[0].readings.voltageTHD	Phase A	Voltage THD
Goertzel or FFT (When the HARMONICS_SUPPORT macro is defined)		
gMetrologyWorkingData.phases[0].readings.vTHDfft	Phase A	Voltage THD

(1) Replace [0] with [1] for Phase B or [2] for Phase C.

(2) To view the Phase A RMS voltage, the phaseReadings structure must be added to the *Expressions Watch* window.

8.2 Accessing the Total System Measurements

System totals are stored in the **gMetrologyWorkingData.totals** structure as:

- gMetrologyWorkingData.totals.readings.activePower
- gMetrologyWorkingData.totals.readings.reactivePower
- gMetrologyWorkingData.totals.readings.apparentPower
- gMetrologyWorkingData.totals.activeEnergy.energy
- gMetrologyWorkingData.totals.reactiveEnergy.energy

8.3 Accessing the Phase Status Flags

Each phase has a status field containing a bitmask of the **PHASE_STATUS** flags: **gMetrologyWorkingData.phases[0].status**.

The following lists useful flag values defined in the *metrology_defines.h* file:

- **PHASE_STATUS_NEW_LOG** (0x0001) – The background has completed a measurement window; however, the foreground has not yet processed the measurement.
- **PHASE_STATUS_V_OVERRANGE** (0x0010) – The voltage ADC input hits the end stops (clipping detected).
- **PHASE_STATUS_I_OVERRANGE** (0x0020) – The current ADC input hits the end stops.
- **PHASE_STATUS_ZERO_CROSSING_MISSED** (0x4000) – The expected zero crossing did not occur within the valid window (frequency out of range or signal lost).

To check if a flag is set, display the status field in hexadecimal format or use a bitwise **AND** expression: **gMetrologyWorkingData.phases[0].status** and 0x4000 are non-zero if a zero crossing was missed.

Note

For true real-time monitoring without halting, enable CCS continuous refresh mode (requires an XDS debug probe with real-time memory access capability). To enable continuous refresh mode:

1. Right-select in the *Expressions* window → *Continuous Refresh* → *Enable*.
   - This polls the target memory periodically while the CPU continues running.
   - This enablement adds to bus traffic and can potentially affect real-time performance. Therefore, use this mode sparingly.

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you fully indemnify TI and its representatives against any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale, TI’s General Quality Guidelines, or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products. Unless TI explicitly designates a product as custom or customer-specified, TI products are standard, catalog, general purpose devices.

TI objects to and rejects any additional or different terms you may propose.

Last updated 10/2025