

ADMT4000 Calibration Routine Guide

OVERVIEW

This document provides a step-by-step guide to perform the calculations necessary to compensate for mechanical errors in a system using the harmonic calibration engine in [ADMT4000](#).

The ADMT4000 contains a total of eight registers to enable the user to compensate for the 1st, 2nd, 3rd, and 8th harmonics. Each harmonic can be corrected by writing to the appropriate harmonic magnitude (HxMAG) and phase (HxPH) registers.

TABLE OF CONTENTS

Overview.....	1	Calibration Steps.....	5
Data Set Requirements.....	3	Main Function.....	8
Function Signature.....	4	C++ Calibration Code Snippet	9
Parameters.....	4	Notes.....	16

REVISION HISTORY**2/2026—Rev. 0 to Rev. A**

Changes to C++ Code Snippet Section.....	9
--	---

/—Revision : Initial Version

Changes to C++ Code Snippet Section.....	9
--	---

3/2025—Revision 0: Initial Version

DATA SET REQUIREMENTS

This guide does not specifically provide the function to capture a set of data to perform the calibration. The following steps should be followed to capture a suitable data set:

1. Set the contents of the harmonic calibration registers (HxMAG and HxPH) to zero.
2. Set Bit 10 of the GENERAL register to one to disable the factory set values of H8MAG and H8PH.
3. The harmonic calibration coefficients are calculated using a the Fast Fourier Transform (FFT). To void aliasing in the FFT, the captured data set should be coherent.
 - a. The number of measurements should be a power of two, (2^n), where n is an integer; 512 samples is typical sufficient.
 - b. The number of rotations over which the data is captured should be a prime number, typically 11.
4. If the angle filter is enabled (Bit 12 of the GENERAL register), data should only be captured within the 0 to 46 turn range.
5. The angle step between measurement points = (number of turns * 360°)/number of measurements.
6. Data can be captured by
 - a. Stepping the system to a position, allowing the system to settle, and then capture the data.
 - b. Running a system at a constant speed and capturing the data at fixed intervals.

The harmonic coefficients are calculated from the angular error, in this guide the error is determined by a straight-line fit, `linear_fit()`.

FUNCTION SIGNATURE

```
void calibrate(vector<double> PANG, int cycleCount, bool CCW);
```

PARAMETERS

PANG	A vector of angle measurements.
cycleCount	The number of cycles in the data.
CCW	A boolean flag indicating if the data should be processed in a counterclockwise direction.

CALIBRATION STEPS

This section describes the step-by-step process of the calibration routine. To see the full calibration code, refer to [C++ Calibration Code Snippet](#).

1. Initialization

The function starts by initializing some variables:

```
string result = "";
```

2. Check CCW Flag

If the CCW flag is set to `true`, the function reverses the PANG vector:

```
if (CCW) reverse(PANG.begin(),
                PANG.end());
```

3. Precalibration FFT

The function declares vectors to store the precalibration FFT results and calls `getPreCalibrationFFT`:

```
vector<double> angle_errors_fft_pre = vector<double>(PANG.size() / 2);
vector<double> angle_errors_fft_phase_pre = vector<double>(PANG.size() / 2);

getPreCalibrationFFT(PANG, angle_errors_fft_pre, angle_errors_fft_phase_pre, cycleCount, samplesPerCycle);
```

4. Extract Harmonic Magnitudes

The function extracts the harmonic magnitudes from the FFT results:

```
double H1Mag = angle_errors_fft_pre[cycleCount];
double H2Mag = angle_errors_fft_pre[2 * cycleCount];
double H3Mag = angle_errors_fft_pre[3 * cycleCount];
double H8Mag = angle_errors_fft_pre[8 * cycleCount];
```

5. Display Harmonic Magnitudes

The function displays the extracted harmonic magnitudes:

```
cout << "H1Mag = " << H1Mag << "\n";
cout << "H2Mag = " << H2Mag << "\n";
cout << "H3Mag = " << H3Mag << "\n";
cout << "H8Mag = " << H8Mag << "\n";
```

CALIBRATION STEPS

6. Extract Harmonic Phases

The function extracts the harmonic phases from the FFT results and converts them to degrees:

```
double H1Phase = (180 / M_PI) * (angle_errors_fft_phase_pre[cycleCount]);
double H2Phase = (180 / M_PI) * (angle_errors_fft_phase_pre[2 * cycleCount]);
double H3Phase = (180 / M_PI) * (angle_errors_fft_phase_pre[3 * cycleCount]);
double H8Phase = (180 / M_PI) * (angle_errors_fft_phase_pre[8 * cycleCount]);
```

7. Display Harmonic Phases

The function displays the extracted harmonic phases:

```
cout << "H1Phase = " << H1Phase << "\n";
cout << "H2Phase = " << H2Phase << "\n";
cout << "H3Phase = " << H3Phase << "\n";
cout << "H8Phase = " << H8Phase << "\n";
```

8. Calculate Initial Error and Angle

The function calculates the initial error and angle:

```
double H1 = H1Mag * cos(M_PI / 180 * (H1Phase));
double H2 = H2Mag * cos(M_PI / 180 * (H2Phase));
double H3 = H3Mag * cos(M_PI / 180 * (H3Phase));
double H8 = H8Mag * cos(M_PI / 180 * (H8Phase));

double init_err = H1 + H2 + H3 + H8;
double init_angle = PANG[0] - init_err;
```

9. Adjust Phases for CCW

If the `CCWflag` is set to `true`, the function negates the harmonic phases:

```
if (CCW) {
    H1Phase *= -1;
    H2Phase *= -1;
    H3Phase *= -1;
    H8Phase *= -1;
}
```

10. Calculate Phase Corrections

The function calculates the phase corrections:

```
double H1PHcor = H1Phase - (1 * init_angle - 90);
double H2PHcor = H2Phase - (2 * init_angle - 90);
double H3PHcor = H3Phase - (3 * init_angle - 90);
double H8PHcor = H8Phase - (8 * init_angle - 90);
```

CALIBRATION STEPS

11. Get Modulo from 360

The function ensures the phase corrections are within the range [0, 360):

```
H1PHcor = (int)H1PHcor % 360;
H2PHcor = (int)H2PHcor % 360;
H3PHcor = (int)H3PHcor % 360;
H8PHcor = (int)H8PHcor % 360;
```

12. Scale Harmonic Magnitudes

The function scales the harmonic magnitudes:

```
H1Mag = H1Mag * 0.6072;
H2Mag = H2Mag * 0.6072;
H3Mag = H3Mag * 0.6072;
H8Mag = H8Mag * 0.6072;
```

13. Derive Register-compatible Values

The function derives register-compatible values for the harmonic magnitudes and phases:

```
double mag_scale_factor_11bit = 11.2455 / (1 << 11);
double mag_scale_factor_8bit = 1.40076 / (1 << 8);
HAR_MAG_1 = (int)(H1Mag / mag_scale_factor_11bit) & (0x7FF); // 11 bit
HAR_MAG_2 = (int)(H2Mag / mag_scale_factor_11bit) & (0x7FF); // 11 bit
HAR_MAG_3 = (int)(H3Mag / mag_scale_factor_8bit) & (0xFF); // 8 bit
HAR_MAG_8 = (int)(H8Mag / mag_scale_factor_8bit) & (0xFF); // 8 bit

double pha_scale_factor_12bit = 360.0 / (1 << 12); // in Deg
HAR_PHASE_1 = (int)(H1PHcor / pha_scale_factor_12bit) & (0xFFF); // 12bit number
HAR_PHASE_2 = (int)(H2PHcor / pha_scale_factor_12bit) & (0xFFF); // 12bit number
HAR_PHASE_3 = (int)(H3PHcor / pha_scale_factor_12bit) & (0xFFF); // 12bit number
HAR_PHASE_8 = (int)(H8PHcor / pha_scale_factor_12bit) & (0xFFF); // 12bit number
```

14. Display Register-compatible Values

The function displays the derived register-compatible values:

```
cout << "HMAG1: " << HAR_MAG_1 << "\n";
cout << "HMAG2: " << HAR_MAG_2 << "\n";
cout << "HMAG3: " << HAR_MAG_3 << "\n";
cout << "HMAG8: " << HAR_MAG_8 << "\n";

cout << "HPHASE1: " << HAR_PHASE_1 << "\n";
cout << "HPHASE2: " << HAR_PHASE_2 << "\n";
cout << "HPHASE3: " << HAR_PHASE_3 << "\n";
cout << "HPHASE8: " << HAR_PHASE_8 << "\n";
```

MAIN FUNCTION

The main function demonstrates how to call the calibrate function with sample data:

```
int main()
{
    int cycleCount = 11;
    bool CCW = false;

    calibrate(sampleCalibrationData, cycleCount, CCW);

    return 0;
}
```

C++ CALIBRATION CODE SNIPPET

```

#include <iostream>
#include <vector>
#include <algorithm>
#include <cstdlib>
#include <cmath>
#include <math.h>
#include <numeric>
#include <complex>
#include <iterator>
#include <cstdint>

using namespace std;

int HAR_MAG_1, HAR_MAG_2, HAR_MAG_3, HAR_MAG_8, HAR_PHASE_1, HAR_PHASE_2, HAR_PHASE_3, HAR_PHASE_8; vector<double> angleError, FFTAngleErrorMagnitude, FFTAngleErrorPhase;
/* Calibration angle data must be coherent for proper calibration */vector<double> sampleCalibrationData = { 359.63745, 15.07874, 30.58594, 46.01624, 61.47400, 76.99768, 92.34009, 107.70447,
123.34900, 138.72437, 154.34143, 170.01343, 185.48767, 201.18713, 216.79871, 232.11914, 247.71973,
263.23792, 278.42102, 293.72498, 309.33655, 324.54712, 339.94446, 355.48462, 10.73914, 26.35071,
41.88538, 57.21680, 72.88879, 88.13782, 103.47473, 119.14673, 134.59351, 150.05127, 165.71228,
181.30737, 196.93542, 212.56897, 227.94434, 243.44604, 258.96423, 274.24072, 289.58313, 305.09583,
320.32288, 335.84656, 351.24939, 6.58630, 22.07703, 37.62268, 52.91565, 68.49976, 83.89709, 99.21204,
114.76318, 130.31982, 145.79407, 161.44958, 177.09412, 192.55188, 208.35022, 223.79700, 239.19983,
254.85535, 270.13733, 285.38635, 300.89355, 316.15906, 331.50696, 347.02515, 2.42798, 17.87476,
33.43140, 48.74084, 64.24255, 79.81567, 94.95483, 110.57190, 126.13953, 141.54785, 157.26379,
172.88635, 188.28369, 203.92273, 219.68811, 234.96460, 250.49927, 265.98999, 281.14014, 296.70227,
312.06665, 327.30469, 342.90527, 358.22021, 13.58459, 29.21265, 44.60449, 60.02380, 75.54199,
90.94482, 106.25427, 121.92627, 137.36755, 152.94617, 168.53027, 184.07593, 199.72046, 215.37598,
230.77881, 246.39587, 261.82617, 276.98181, 292.40112, 307.88635, 323.08594, 338.59863, 354.14429,
9.33838, 24.92798, 40.48462, 55.75012, 71.32324, 86.75903, 102.06299, 117.68005, 133.15430, 148.66699,
164.39392, 179.81873, 195.41931, 211.17371, 226.63147, 242.13318, 257.67883, 272.84546, 288.28125,
303.70056, 318.93860, 334.36890, 349.85962, 5.19653, 20.70923, 36.22192, 51.55884, 67.10999,
82.65015, 97.86072, 113.31848, 129.00696, 144.29443, 159.98291, 175.66589, 191.18408, 206.85608,
222.40173, 237.85950, 253.39417, 268.75305, 284.03503, 299.59167, 314.77478, 330.09521, 345.66833,
1.06018, 16.46301, 31.95374, 47.34009, 62.92969, 78.42041, 93.69690, 109.16016, 124.76074, 140.16907,
155.70923, 171.43616, 186.81152, 202.52197, 218.22144, 233.55835, 249.13147, 264.63318, 279.78333,
295.20264, 310.75378, 325.93140, 341.40564, 356.81396, 12.17834, 27.77893, 43.20923, 58.57361,
74.18518, 89.56604, 104.81506, 120.51453, 135.94482, 151.61682, 167.20642, 182.71912, 198.30872,
213.99170, 229.39453, 244.94019, 260.47485, 275.66895, 291.08826, 306.51855, 321.68518, 337.19238,
352.70508, 7.90466, 23.44482, 39.16077, 54.36584, 69.88403, 85.37476, 100.56885, 116.27930, 131.81396,
147.28821, 162.90527, 178.43445, 194.03503, 209.64661, 225.16479, 240.71045, 256.25610, 271.50513,
286.85303, 302.28882, 317.49390, 332.92969, 348.51929, 3.82324, 19.25903, 34.85962, 50.18005,
65.68176, 81.22192, 96.47644, 111.91223, 127.55127, 142.83325, 158.55469, 174.27063, 189.68445,
205.41138, 221.04492, 236.39282, 252.07581, 267.42920, 282.57935, 298.09753, 313.40698, 328.69446,
344.27307 };

unsigned int bitReverse(unsigned int x, int log2n)
{
    int n = 0;
    int mask = 0x1;
    for (int i = 0; i < log2n; i++) {
        n <<= 1;
        n |= (x & 1);
        x >>= 1;
    }
}

```

C++ CALIBRATION CODE SNIPPET

```

    return n;
}

template<class Iter_T>
void fft(Iter_T a, Iter_T b, int log2n)
{
    typedef typename iterator_traits<Iter_T>::value_type complex;
    const complex J(0, 1);
    int n = 1 << log2n;
    for (unsigned int i = 0; i < n; ++i) {
        b[bitReverse(i, log2n)] = a[i];
    }
    for (int s = 1; s <= log2n; ++s) {
        int m = 1 << s;
        int m2 = m >> 1;
        complex w(1, 0);
        complex wm = exp(-J * (M_PI / m2));
        for (int j = 0; j < m2; ++j) {
            for (int k = j; k < n; k += m) {
                complex t = w * b[k + m2];
                complex u = b[k];
                b[k] = u + t;
                b[k + m2] = u - t;
            }
            w *= wm;
        }
    }
}

/* For linear fitting (hard-coded based on examples and formula for polynomial fitting)
*/static int linear_fit(vector<double> x, vector<double> y, double* slope, double* intercept)
{
    /* x, y, x^2, y^2, xy, xy^2 */
    double sum_x = 0, sum_y = 0, sum_x2 = 0, sum_y2 = 0, sum_xy = 0;
    int i;

    if (x.size() != y.size())
        return -22;

    for (i = 0; i < x.size(); i++) {
        sum_x += x[i];
        sum_y += y[i];
        sum_x2 += (x[i] * x[i]);
        sum_y2 += (y[i] * y[i]);
        sum_xy += (x[i] * y[i]);
    }

    *slope = (x.size() * sum_xy - sum_x * sum_y) / (x.size() * sum_x2 - sum_x * sum_x);

    *intercept = (sum_y * sum_x2 - sum_x * sum_xy) / (x.size() * sum_x2 - sum_x * sum_x);

    return 0;
}

/* Calculate angle error based on MATLAB and C# implementation*/

```

C++ CALIBRATION CODE SNIPPET

```

int calculate_angle_error(vector<double> angle_meas, vector<double>& angle_error_ret, double* max_angle_err)
{
    vector<double> angle_meas_rad(angle_meas.size()); // radian converted input
    vector<double> angle_meas_rad_unwrap(angle_meas.size()); // unwrapped radian input
    vector<double> angle_fit(angle_meas.size()); // array for polynomial fitted data
    vector<double> x_data(angle_meas.size());
    double coeff_a, coeff_b; // coefficients generated by polynomial fitting

    // convert to radian
    for (int i = 0; i < angle_meas_rad.size(); i++)
        angle_meas_rad[i] = angle_meas[i] * M_PI / 180.0;

    // unwrap angle (extracted from decompiled Angle GSF Unit
    double num = 0.0;
    angle_meas_rad_unwrap[0] = angle_meas_rad[0];
    for (int i = 1; i < angle_meas_rad.size(); i++)
    {
        double num2 = abs(angle_meas_rad[i] + num - angle_meas_rad_unwrap[i - 1]);
        double num3 = abs(angle_meas_rad[i] + num - angle_meas_rad_unwrap[i - 1] + M_PI * 2.0);
        double num4 = abs(angle_meas_rad[i] + num - angle_meas_rad_unwrap[i - 1] - M_PI * 2.0);
        if (num3 < num2 && num3 < num4)
            num += M_PI * 2.0;

        else if (num4 < num2 && num4 < num3)
            num -= M_PI * 2.0;

        angle_meas_rad_unwrap[i] = angle_meas_rad[i] + num;
    }

    // set initial point to zero
    double offset = angle_meas_rad_unwrap[0];
    for (int i = 0; i < angle_meas_rad_unwrap.size(); ++i)
        angle_meas_rad_unwrap[i] -= offset;

    /* Generate xdata for polynomial fitting */
    iota(x_data.begin(), x_data.end(), 1);

    // linear angle fitting (generated coefficients not same with matlab and python)
    // expecting 0.26 -0.26
    // getting ~0.27 ~-0.27 as of 4/2/2024
    /* input args: x, y, *slope, *intercept */
    linear_fit(x_data, angle_meas_rad_unwrap, &coeff_a, &coeff_b);

    //cout << coeff_a << " " << coeff_b << "\n";

    // generate data using coefficients from polynomial fitting
    for (int i = 0; i < angle_fit.size(); i++) {
        angle_fit[i] = coeff_a * x_data[i];
        //cout << "angle_fit " << angle_fit[i] << "\n";
    }

    // get angle error using pass by ref angle_error_ret
    for (int i = 0; i < angle_error_ret.size(); i++) {
        angle_error_ret[i] = angle_meas_rad_unwrap[i] - angle_fit[i];
        //cout << "angle_err_ret " << angle_error_ret[i] << "\n";
    }
}

```

C++ CALIBRATION CODE SNIPPET

```

}

// Find the offset for error and subtract (using angle_error_ret)
auto minmax = minmax_element(angle_error_ret.begin(), angle_error_ret.end());
double angle_err_offset = (*minmax.first + *minmax.second) / 2;

for (int i = 0; i < angle_error_ret.size(); i++)
    angle_error_ret[i] -= angle_err_offset;

// Convert back to degrees (angle_error_ret)
for (int i = 0; i < angle_meas.size(); i++)
    angle_error_ret[i] *= (180 / M_PI);

// Find maximum absolute angle error
*max_angle_err = *minmax.second;

return 0;
}

void performFFT(const vector<double>& angle_errors, vector<double>& angle_errors_fft, vector<double>&
angle_errors_fft_phase, int cycleCount)
{
    typedef complex<double> cx;

    int L = angle_errors.size(); // Original signal length (L)
    int N = pow(2, ceil(log2(L))); // Ensure size is a power of 2 (padding if necessary)

    vector<cx> fft_in(N, cx(0, 0)); // Input signal (zero-padded if necessary)
    vector<cx> fft_out(N); // Output signal (complex)

    // Format angle errors into the fft_in vector
    for (int i = 0; i < L; i++) {
        fft_in[i] = cx(angle_errors[i], 0);
    }

    // Perform FFT
    fft(fft_in.data(), fft_out.data(), log2(N));

    // Temporary vectors to store magnitude and phase
    vector<double> angle_errors_fft_temp(N);
    vector<double> angle_errors_fft_phase_temp(N);

    // Calculate magnitude and phase for all values
    for (int i = 0; i < N; i++) {
        // Magnitude: Normalize by L (original signal length)
        angle_errors_fft_temp[i] = abs(fft_out[i]) * 2.0 / L;
        angle_errors_fft_phase_temp[i] = atan2(fft_out[i].imag(), fft_out[i].real());
    }

    // Prepare vectors for upper half of FFT (positive frequencies)
    vector<double> angle_errors_fft_upper_half(N / 2);
    vector<double> angle_errors_fft_phase_upper_half(N / 2);

    // Get upper half only (due to symmetry in real-valued signal FFT)
    for (int i = 0; i < N / 2; i++) {

```

C++ CALIBRATION CODE SNIPPET

```

    angle_errors_fft_upper_half[i] = angle_errors_fft_temp[i];
    angle_errors_fft_phase_upper_half[i] = angle_errors_fft_phase_temp[i];
}

// Resize final vectors based on cycle count (if needed)
angle_errors_fft = angle_errors_fft_upper_half;
angle_errors_fft_phase = angle_errors_fft_phase_upper_half;
}

void getPreCalibrationFFT(const vector<double>& PANG, vector<double>& angle_errors_fft_pre, vector<double>& angle_errors_fft_phase_pre, int cycleCount)
{
    // Calculate the angle errors before calibration
    double max_err_pre = 0;
    vector<double> angle_errors_pre(PANG.size());

    // Calculate angle errors
    calculate_angle_error(PANG, angle_errors_pre, &max_err_pre);

    // Store the calculated angle errors (angle_errors_pre)
    angleError = angle_errors_pre;

    // Perform FFT on pre-calibration angle errors
    performFFT(angle_errors_pre, angle_errors_fft_pre, angle_errors_fft_phase_pre, cycleCount);

    // Store the FFT Angle Error Magnitude and Phase
    FFTAngleErrorMagnitude = angle_errors_fft_pre;
    FFTAngleErrorPhase = angle_errors_fft_phase_pre;
}

void calibrate(vector<double> PANG, int cycleCount, bool CCW)
{
    string result = "";

    /* Check CCW flag to know if array is to be reversed */
    if (CCW)
        reverse(PANG.begin(), PANG.end());
}

// Declare vectors for pre-calibration FFT results
vector<double> angle_errors_fft_pre = vector<double>(PANG.size() / 2);
vector<double> angle_errors_fft_phase_pre = vector<double>(PANG.size() / 2);

// Call the new function for pre-calibration FFT
getPreCalibrationFFT(PANG, angle_errors_fft_pre, angle_errors_fft_phase_pre, cycleCount);

// Extract HMag parameters
double H1Mag = angle_errors_fft_pre[cycleCount];
double H2Mag = angle_errors_fft_pre[2 * cycleCount];
double H3Mag = angle_errors_fft_pre[3 * cycleCount];
double H8Mag = angle_errors_fft_pre[8 * cycleCount];

/* Display HMAG values */
cout << "H1Mag = " << H1Mag << "\n";
cout << "H2Mag = " << H2Mag << "\n";

```

C++ CALIBRATION CODE SNIPPET

```

cout << "H3Mag = " << H3Mag << "\n";
cout << "H8Mag = " << H8Mag << "\n";

// Extract HPhase parameters
double H1Phase = (180 / M_PI) * (angle_errors_fft_phase_pre[cycleCount]);
double H2Phase = (180 / M_PI) * (angle_errors_fft_phase_pre[2 * cycleCount]);
double H3Phase = (180 / M_PI) * (angle_errors_fft_phase_pre[3 * cycleCount]);
double H8Phase = (180 / M_PI) * (angle_errors_fft_phase_pre[8 * cycleCount]);

/* Display HPHASE values */
cout << "H1Phase = " << H1Phase << "\n";
cout << "H2Phase = " << H2Phase << "\n";
cout << "H3Phase = " << H3Phase << "\n";
cout << "H8Phase = " << H8Phase << "\n";

cout << "\n\n";

double H1 = H1Mag * cos(M_PI / 180 * (H1Phase));
double H2 = H2Mag * cos(M_PI / 180 * (H2Phase));
double H3 = H3Mag * cos(M_PI / 180 * (H3Phase));
double H8 = H8Mag * cos(M_PI / 180 * (H8Phase));

double init_err = H1 + H2 + H3 + H8;
double init_angle = PANG[0] - init_err;

double H1PHcor, H2PHcor, H3PHcor, H8PHcor;

/* Counterclockwise, slope of error FIT is negative */
if (CCW) {
    H1Phase *= -1;
    H2Phase *= -1;
    H3Phase *= -1;
    H8Phase *= -1;
}

/* Clockwise */
H1PHcor = H1Phase - (1 * init_angle - 90);
H2PHcor = H2Phase - (2 * init_angle - 90);
H3PHcor = H3Phase - (3 * init_angle - 90);
H8PHcor = H8Phase - (8 * init_angle - 90);

/* Get modulo from 360 */
H1PHcor = (int)H1PHcor % 360;
H2PHcor = (int)H2PHcor % 360;
H3PHcor = (int)H3PHcor % 360;
H8PHcor = (int)H8PHcor % 360;

// HMag Scaling
H1Mag = H1Mag * 0.6072;
H2Mag = H2Mag * 0.6072;
H3Mag = H3Mag * 0.6072;
H8Mag = H8Mag * 0.6072;

// Derive register compatible HMAG values
double mag_scale_factor_11bit = 11.2455 / (1 << 11);
double mag_scale_factor_8bit = 1.40076 / (1 << 8);

```

C++ CALIBRATION CODE SNIPPET

```
HAR_MAG_1 = (int)(H1Mag / mag_scale_factor_11bit) & (0x7FF); // 11 bit
HAR_MAG_2 = (int)(H2Mag / mag_scale_factor_11bit) & (0x7FF); // 11 bit
HAR_MAG_3 = (int)(H3Mag / mag_scale_factor_8bit) & (0xFF); // 8 bit
HAR_MAG_8 = (int)(H8Mag / mag_scale_factor_8bit) & (0xFF); // 8 bit

// Derive register compatible HPHASE values
double pha_scale_factor_12bit = 360.0 / (1 << 12); // in Deg
HAR_PHASE_1 = (int)(H1PHcor / pha_scale_factor_12bit) & (0xFFF); // 12bit number
HAR_PHASE_2 = (int)(H2PHcor / pha_scale_factor_12bit) & (0xFFF); // 12bit number
HAR_PHASE_3 = (int)(H3PHcor / pha_scale_factor_12bit) & (0xFFF); // 12bit number
HAR_PHASE_8 = (int)(H8PHcor / pha_scale_factor_12bit) & (0xFFF); // 12bit number

cout << "HMAG1: " << HAR_MAG_1 << "\n";
cout << "HMAG2: " << HAR_MAG_2 << "\n";
cout << "HMAG3: " << HAR_MAG_3 << "\n";
cout << "HMAG8: " << HAR_MAG_8 << "\n";

cout << "HPHASE1: " << HAR_PHASE_1 << "\n";
cout << "HPHASE2: " << HAR_PHASE_2 << "\n";
cout << "HPHASE3: " << HAR_PHASE_3 << "\n";
cout << "HPHASE8: " << HAR_PHASE_8 << "\n";

cout << "\n\n";
}

int main()
{
    int cycleCount = 11;
    bool CCW = false;

    calibrate(sampleCalibrationData, cycleCount, CCW);

    return 0;
}
```

NOTES

**ESD Caution**

ESD (electrostatic discharge) sensitive device. Charged devices and circuit boards can discharge without detection. Although this product features patented or proprietary protection circuitry, damage may occur on devices subjected to high energy ESD. Therefore, proper ESD precautions should be taken to avoid performance degradation or loss of functionality.

Legal Terms and Conditions

By using the evaluation board discussed herein (together with any tools, components documentation or support materials, the "Evaluation Board"), you are agreeing to be bound by the terms and conditions set forth below ("Agreement") unless you have purchased the Evaluation Board, in which case the Analog Devices Standard Terms and Conditions of Sale shall govern. Do not use the Evaluation Board until you have read and agreed to the Agreement. Your use of the Evaluation Board shall signify your acceptance of the Agreement. This Agreement is made by and between you ("Customer") and Analog Devices, Inc. ("ADI"), with its principal place of business at One Analog Way, Wilmington, MA 01887-2356, U.S.A. Subject to the terms and conditions of the Agreement, ADI hereby grants to Customer a free, limited, personal, temporary, non-exclusive, non-sublicensable, non-transferable license to use the Evaluation Board FOR EVALUATION PURPOSES ONLY. Customer understands and agrees that the Evaluation Board is provided for the sole and exclusive purpose referenced above, and agrees not to use the Evaluation Board for any other purpose. Furthermore, the license granted is expressly made subject to the following additional limitations: Customer shall not (i) rent, lease, display, sell, transfer, assign, sublicense, or distribute the Evaluation Board; and (ii) permit any Third Party to access the Evaluation Board. As used herein, the term "Third Party" includes any entity other than ADI, Customer, their employees, affiliates and in-house consultants. The Evaluation Board is NOT sold to Customer; all rights not expressly granted herein, including ownership of the Evaluation Board, are reserved by ADI. CONFIDENTIALITY. This Agreement and the Evaluation Board shall all be considered the confidential and proprietary information of ADI. Customer may not disclose or transfer any portion of the Evaluation Board to any other party for any reason. Upon discontinuation of use of the Evaluation Board or termination of this Agreement, Customer agrees to promptly return the Evaluation Board to ADI. ADDITIONAL RESTRICTIONS. Customer may not disassemble, decompile or reverse engineer chips on the Evaluation Board. Customer shall inform ADI of any occurred damages or any modifications or alterations it makes to the Evaluation Board, including but not limited to soldering or any other activity that affects the material content of the Evaluation Board. Modifications to the Evaluation Board must comply with applicable law, including but not limited to the RoHS Directive. TERMINATION. ADI may terminate this Agreement at any time upon giving written notice to Customer. Customer agrees to return to ADI the Evaluation Board at that time. LIMITATION OF LIABILITY. THE EVALUATION BOARD PROVIDED HEREUNDER IS PROVIDED "AS IS" AND ADI MAKES NO WARRANTIES OR REPRESENTATIONS OF ANY KIND WITH RESPECT TO IT. ADI SPECIFICALLY DISCLAIMS ANY REPRESENTATIONS, ENDORSEMENTS, GUARANTEES, OR WARRANTIES, EXPRESS OR IMPLIED, RELATED TO THE EVALUATION BOARD INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTY OF MERCHANTABILITY, TITLE, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS. IN NO EVENT WILL ADI AND ITS LICENSORS BE LIABLE FOR ANY INCIDENTAL, SPECIAL, INDIRECT, OR CONSEQUENTIAL DAMAGES RESULTING FROM CUSTOMER'S POSSESSION OR USE OF THE EVALUATION BOARD, INCLUDING BUT NOT LIMITED TO LOST PROFITS, DELAY COSTS, LABOR COSTS OR LOSS OF GOODWILL. ADI'S TOTAL LIABILITY FROM ANY AND ALL CAUSES SHALL BE LIMITED TO THE AMOUNT OF ONE HUNDRED US DOLLARS (\$100.00). EXPORT. Customer agrees that it will not directly or indirectly export the Evaluation Board to another country, and that it will comply with all applicable United States federal laws and regulations relating to exports. GOVERNING LAW. This Agreement shall be governed by and construed in accordance with the substantive laws of the Commonwealth of Massachusetts (excluding conflict of law rules). Any legal action regarding this Agreement will be heard in the state or federal courts having jurisdiction in Suffolk County, Massachusetts, and Customer hereby submits to the personal jurisdiction and venue of such courts. The United Nations Convention on Contracts for the International Sale of Goods shall not apply to this Agreement and is expressly disclaimed. All Analog Devices products contained herein are subject to release and availability.

