Advanced Programming with ASEK DLLs Harmonic Linearization

By K. Robert Bate,
Allegro MicroSystems, LLC

Introduction

Numerous applications in industries spanning from industrial automation and robotics, to electronic power steering and motor position sensing require monitoring the angle of a rotating shaft either in an on-axis or off-axis arrangement.

When using a magnet in a design, the magnetic input will most likely not be homogeneous over the entire range of rotation—it will have inherent errors. These magnetic input errors cause measurement error in the system. Linearization can reduce these input errors.

One form of linearization available on the A1332 and A1335, *harmonic linearization* applies linearization in the form of up to 15 correction harmonics whose phase and amplitude are determined by means of an FFT (fast Fourier transform) performed on the data collected from one rotation of the magnet around the angle sensor IC. This technique can be readily implemented using Allegro-provided software to calculate coefficients and program on-chip EEPROM. This application note describes the functions and the process flow the customer can use if the Allegro-provided software is not flexible enough or if custom software is to be used.

Programming Requirements

All of the software was developed on Microsoft Visual Studio 2010 using .NET 4.0. Download the Command Library (C#/.NET) for the device that you are going to use and add to the project references to the three DLLs that it contains.

Collecting the Data

First, turn off all post-linearization algorithmic processing; this includes ZeroOffset, Post-Linearization Rotation (RO), Short Stroke Inversion (IV), and the Rotate Die bit (RD). Prelinearization adjustments may be left on, such as ORATE settings, IIR filter (FI), and Prelinearization Rotation (LR).

Move the encoder in the direction of increasing angle position. If the angle sensor IC output does not also increase, then either set the LR bit to reverse the direction of the angle sensor IC output or rotate the encoder in the opposite direction for calibration, in which case the Post-Linearization Rotate bit (RO) will likely need to be set. See the A1332/A1335 programming reference for more details.

The optimal collection method is to rotate the target in equally spaced steps such that the number of resultant data points is a power of 2. Usually, 32 or 64 evenly spaced data points are sufficient. If this cannot be accomplished, then collect the points, and the data must be preprocessed as discussed in the next section.

Another technique to gather the required data points is to rotate the target many times, collecting the data at a predefined interval. Once enough points have been gathered to cover the entire rotation of the target, then they must be preprocessed as discussed in the next section.

Preprocessing the Data

If the number of data points collected is not a power of two or the data collected is not equally spaced, then the array of points must be resized and/or made to be equally spaced. To perform this action on the data, call the routine ResizePointArray.

```
double[] ResizePointArray(double[] x, double[] y,
 int newSize)
```

The x parameter is the array of encoder values and the y parameter is the device readings that were collected at that encoder value. The parameter newSize is the desired size of the resized array. If the x parameter is set to null, then the y values are assumed to have been collected with an equal spacing starting at 0 and ending at 360. If the x parameter is not null, then the input arrays are sorted before the resizing is performed.

This routine will perform a cubic-spline interpolation on the input arrays to generate an equal-spaced array with the desired number of points.

Initial Processing

Once the data has been collected and made into an array of a length which is a power of 2, then the harmonic coefficients are ready to be calculated. To calculate the coefficients, call the routine CalculateHarmonicLinearCoefficients.

```
HarmonicCoefficients[] CalculateHarmonicLinearCoefficients
 (double[] points, out bool pointError)
```

Its input is the array of angles that has been collected. The routine performs an FFT and will return the array of coefficients and a warning flag. The point error warning flag is set when one or more of the input angles is greater than 20 degrees different from what the routine calculates it should be.

For example, for an 8 entry array, the routine calculates that the angles should be [0, 45, 90, 135, 180, 225, 270, 315]. If the input array is [0, 45, 90, 135, 180, 204, 270, 315], then the routine will set the pointError because the 6th array entry has an error greater than 20 degrees.

Selection of Harmonics

Once all of the harmonic coefficients have been calculated, the desired harmonics must be selected. In general, the number of harmonics generated by the calculate routine will exceed the number of harmonics the device can support, so some algorithm to select the relevant harmonics must be chosen.

The number of harmonics used is also dependent on which device and which features are used. For the A1332, the maximum number of harmonics is 15, but if the maximum is used, a number of programmable features will use the defaults, such as short-stroke configurations and specific I^2C and SPI settings. The maximum number of harmonics without using the defaults for those programmable features is 9. For the A1335, the maximum number of harmonics is 11, but to get this number, a number of programmable features will use the defaults, such as the short-stroke settings. The maximum number of harmonics without using the defaults for those programmable features is 8.

The simplest algorithm to use is to select the first harmonic through to the desired number of harmonics. While easy, it will select harmonics that will not significantly influence the output.

The current algorithm that is used in the Allegro A1335 Samples Programmer is to select the harmonic where the amplitude is greater than 0.3. One limitation in the current hardware to note is that only 4 harmonics can be skipped between selected harmonics. If there is a jump greater than 4, then as many harmonics as needed between the last harmonic selected and the desired harmonic also need to be selected.

Programming the Device

Once the harmonics have been selected, then the values to be written into the device can be generated by calling the routine GenerateHarmonicLinearizationDeviceValues.

```
HarmonicDeviceValues[] GenerateHarmonicLinearizationDeviceValues
  (HarmonicCoefficients[] coefficients)
```

The harmonic coefficients are passed into this routine, and it returns an array of the values needed to program the device. The only exception this routine will throw is the case where there are more than 4 harmonic coefficients skipped between selected coefficients.

To program the device for harmonic linearization, the HL flag must be set, the HAR_MAX field must be set to the number of coefficients to be used, and the HARMONIC_PHASE_n, ADV_n, and HARMONIC_AMPLITUDE_n fields must be written.

APPENDIX

Example Code

```
using System;
using Allegro.ASEK;

namespace HarmonicLinearizationExample
{
    public class HarmonicLinearizationExample
    {
        public HarmonicLinearizationExample()
        {
        }

        public void ProgramHarmonicLinearization(string filePath, ASEK asekProgrammer)
        {
            try
            {
                HarmonicCoefficients[] hc;
                bool pointError = false;
                double[] points = null;
                string fieldBuffer = File.ReadAllText(filePath);
                string line;
                List<double> encoderReadings = new List<double>();
                List<double> deviceReadings = new List<double>();

                // 1.1 Collecting the data
                // Read in the angles from a text file. Blank lines or lines starting with a # are ignored.
                if (!string.IsNullOrEmpty(fieldBuffer))
                {
                    using (StringReader sr = new StringReader(fieldBuffer))
                    {
                        while ((line = sr.ReadLine()) != null)
                        {
                            line = line.Trim();
                            if (string.IsNullOrEmpty(line) || line.StartsWith("#"))
                            {
                                continue;
                            }

                            // Each line can be in 1 of 2 forms.
                            // The first contains an encoder angle then the angle from the device separated by a comma (22.125,23.543)
                            // or just the angle from the device. (23.543)
                            // If the angles are not equally spaced then both values are needed.
                            string[] values = line.Split(‘,’);

                            if (values.Length > 1)
                            {
                                double encoder = Convert.ToDouble(values[0]);
                                while (encoder >= 360.0)
                                {
                                    encoder -= 360.0;
                                }
                                while (encoder < 0.0)
                                {
                                    encoder += 360.0;
                                }
                                encoderReadings.Add(encoder);
                                deviceReadings.Add(Convert.ToDouble(values[1]));
                            }
                            else
                            {
                                deviceReadings.Add(Convert.ToDouble(values[0]));
                            }
                        }
                    }

                    // 1.2 Preprocessing the data
                    if (!powerOfTwo(deviceReadings.Count()))
                    {
                        // If the number of points is off by one then just remove the last one,
                        if (powerOfTwo(deviceReadings.Count() - 1))
                        {
                            deviceReadings.RemoveAt(deviceReadings.Count() - 1);
                            points = deviceReadings.ToArray();
                        }
                        else
                        {
                            // otherwise calculate the number of desired samples.
                            // If the number of samples is less then 64 then round
                            // up to the nearest power of two, otherwise round down.
                            int desiredSamples = 8;
                            while (desiredSamples < deviceReadings.Count())
                            {
                                desiredSamples *= 2;
                            }

                            if (deviceReadings.Count() > 64)
                            {
                                desiredSamples /= 2;
                            }

                            // If there are no encoder readings, assume the devices readings are equally spaced.
                            if (encoderReadings.Count() != deviceReadings.Count())
                            {
                                // Convert the list of angles to an array and then resize it.
                                points = ((IHarmonicLinearization)asekProgrammer).ResizePointArray(null, deviceReadings.ToArray(), desiredSamples);
                            }
                            else
                            {
                                // Convert the list of angles to an array and then resize it.
                                points = ((IHarmonicLinearization)asekProgrammer).ResizePointArray(encoderReadings.ToArray(), deviceReadings.
ToArray(), desiredSamples);
                            }
                        }
                    }
                    else
                    {
                        // Convert the list of angles to an array
                        points = deviceReadings.ToArray();
                    }

                    // 1.3 Initial Processing
                    // Calculate the coefficients from the array of points.
                    hc = ((IHarmonicLinearization)asekProgrammer).CalculateHarmonicLinearCoefficients(points, out pointError);

                    // A point error is what happens when one or more of the angles is more then 20 degrees different then what the routine
thinks it should be.
                    // For example if an array of 8 values is to be used [0, 45, 90, 135, 180, 204, 270, 315] the calculate would flag a warning
for the 6th
                    // entry in the array because it should be closer to 225 than it is.
                    if (pointError)
                    {
                        MessageBox.Show("One of the angles is more then 20 degrees off from what is desired.");
                    }

                    // 1.4 Selection of Harmonics
                    // Once the array of harmonic coefficients has been calculated, the coefficients need to be selected. The number of coeffi-
cients that
                    // the calculate routine returns will generally exceed the number of coefficients the devices can support so some method of
limiting the number
                    // of coefficents is needed. The first 8 could be picked or another method could be used.
                    int numberOfHarmonicComponents = hc.Length;
                    int numberOfSelectedHarmonicComponents = 0;
                    int lastHarmonicComponentSelected = 0;
                    int maxHarmonicComponentsSelected = 8; // maximum number of harmonics that can be used before impacting other features on the
device

                    // For this example, the first 8 harmonics where the amplitude exceed 0.3 are selected.
                    for (int index = 0; index < numberOfHarmonicComponents; ++index)
                    {
                        if ((hc[index].amplitude > 0.3) && (numberOfSelectedHarmonicComponents < maxHarmonicComponentsSelected))
                        {
                            // if the number of harmonics between the harmonic that is to be selected
                            // and the last harmonic selected is greater then 4 then some of the
                            // harmonics between then need to be selected.
                            int skip = index - lastHarmonicComponentSelected;
                            if (skip > 4)
                            {
                                // Make sure that the number of harmonics that will need to be selected
                                // does not exceed the number that is desired.
                                int numberNeeded = skip / 4;
                                if ((numberNeeded + numberOfSelectedHarmonicComponents) <= maxHarmonicComponentsSelected)
                                {
                                    for (int jndex = 1; jndex <= numberNeeded; ++jndex)
                                    {
                                        hc[jndex].select = true;
                                        ++numberOfSelectedHarmonicComponents;
                                    }
                                    hc[index].select = true;
                                    ++numberOfSelectedHarmonicComponents;
                                }
                                else
                                {
                                    // The code is unable to select the desired harmonic without exceeding
                                    // the maximum number of coefficients selected so it will stop selecting.
                                    break;
                                }
                            }
                            else
                            {
                                hc[index].select = true;
                                ++numberOfSelectedHarmonicComponents;
                            }
                            lastHarmonicComponentSelected = index;
                        }
                    }

                    // If there are no harmonics selected, then select the first 8.
                    if (numberOfSelectedHarmonicComponents == 0)
                    {
                        for (int i = 0; (i < numberOfHarmonicComponents) && (numberOfSelectedHarmonicComponents < 8); ++i)
                        {
                            hc[i].select = true;
                            ++numberOfSelectedHarmonicComponents;
                        }
                    }

                    // 1.5 Programming the Device
                    // Generate the values needed to be written into the eeprom.
                    HarmonicDeviceValues[] eepromValues = ((IHarmonicLinearization)asekProgrammer).GenerateHarmonicLinearizationDeviceValues(hc);

                    // Make sure the power is on for the device
                    asekProgrammer.SetVcc(5.0);

                    // Enable writing to the device's eeprom, this requires making the SRAM writable and to stop the processor
                    ((ISRAMWriteAccessMode)asekProgrammer).SetSRAMWriteAccessMode();
                    ((IProcessorMode)asekProgrammer).SetProcessorIdle();

                    // Turn on harmonic linearization in the eeprom
                    ((IRegisterAccess)asekProgrammer).WritePartialRegister(MemoryAccessType.extended, 0x306, 1, 15, 15); // HL = 1

                    // Set the number of harmonics to be used
                    ((IRegisterAccess)asekProgrammer).WritePartialRegister(MemoryAccessType.extended, 0x309, eepromValues.Length, 19, 16); //
HAR_MAX = number of harmonics

                    // For the harmonics
                    for (uint index = 0; index < eepromValues.Length; ++index)
                    {
                        uint registerValue = (uint)(((eepromValues[index].phase << 12) & 0x0FFF000) +
                                                    ((eepromValues[index].advance << 10) & 0x0C00) +
                                                    (eepromValues[index].amplitude & 0x03FF));
                        ((IRegisterAccess)asekProgrammer).WriteRegister(MemoryAccessType.extended, 0x30C + index, registerValue); // HARMONIC_
PHASE, ADV and HARMONIC_AMPLITUDE
                    }

                    // Turn the power off then back on to make sure the device is using the new linearization values.
                    asekProgrammer.SetVccOff();
                    asekProgrammer.SetVcc(5.0);
                }
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message);
            }
        }

        private bool powerOfTwo(int value)
        {
            int log2npoints = 0;
```

```
            int j = value;

            while ((j > 0) && ((j & 1) == 0))      // Compute log base 2 of input value
            {
                log2npoints++;
                j >>= 1;
            }

            if ((value < 2) || (value != (1 << log2npoints)))
            {
                return false;
            }

            return true;
        }
    }
}
```

Angles Input File Format

This file contains a list of angle values. If there are two values separated by a comma, then the first value is the encoder angle and the second value is the device angle. Lines can be blank, or if they start with #, then they are considered comments.

Example of Angles Input file:

```
329.59
354.81
6.832
13.566
17.592
20.228
22.638
24.638
25.956
27.454
28.77
30.054
30.966
```

With two columns:

```
0,123
22.5,145.5
45,168
67.5,190.5
90,213
112.5,235.5
135,258
157.5,280.5
180,303
202.5,325.5
225,348
247.5,10.5
270,33
292.5,55.5
315,78
337.5,100.5
```

Revision History

Revision	Revision Date	Description of Revision
–	January 28, 2016	Initial release

For the latest version of this document, visit our website:

www.allegromicro.com