TWO-POINT PROGRAMMING FOR THE A1330

By Robert Bate
Allegro MicroSystems

ABSTRACT
This application note serves as a guide for using a two-point programming algorithm to calculate and program the values needed for short-stroke rotational position sensing with the A1330 angle sensor IC. It will outline the procedure for setting the EEPROM registers needed for A1330 short-stroke applications.

INTRODUCTION
Accurate, low-cost, and noncontact rotational position sensing is often achieved using a diametric puck magnet and a magnetic sensor IC. The magnet is attached to the rotating object and the sensor IC is positioned such that the face of the magnet rotates parallel to the face of the sensor IC package. Short stroke (or fine angle scaling) is defined as magnetic angle rotations less than 360° to be represented by a full-scale output from the IC. Achieving full-scale output on sub-360° rotations allows the user to use the entire dynamic range of the ADC. Applications that are often ideal for short stroke include:

- pedal position
- fuel tank level sensing
- gear position
- throttle and/or valve position
- actuator position

The Allegro A1330 magnetic angle sensor IC is well-suited for short-stroke rotational position sensing because it provides advanced features such as:

- Analog/PWM Output: This configurable output allows easy reading and validation.
- High and Low Angle Clamps: Adjustable output saturation is highly configurable.
- User-Configurable Gain and Offset: To achieve full-scale output with little input change, GAIN and PREGAIN_OFFSET provide the ideal solution.
- Minimum and Maximum Angle Detection: Setting a minimum and maximum angle in EEPROM can provide a diagnostic check. It verifies the magnet is in a valid operating position.

ALGORITHM
One of the common uses for the A1330 is to measure a throttle position. Using a dual-die device, very often the desired output looks like Figure 1. The blue curve is the output of die 1 and the red curve is the output of die 2. When swept over the target input range, the two outputs are summed together to get a constant value. This verifies correct operation of the system. The target input range is rarely a full rotation, so the angle input needs to be gained up to use the full range of output and offset to get the output to the desired values.

![Figure 1: Throttle Position](image-url)
## Inputs

The inputs to the two-point programming algorithm are as follows:

- Measured value at position 1: The voltage or duty cycle of the A1330 at position 1, measured_position_1.
- Measured value at position 2: The voltage or duty cycle of the A1330 at position 2, measured_position_2.
- Measured rotation direction: The direction of rotation when the reading for Position 1 and 2 were made (PO flag), measured_rotation_direction.
- Desired value at position 1: What the value at position 1 is to be after the calculations, desired_position_1.
- Desired value at position 2: What the value at position 2 is to be after the calculations, desired_position_2.
- Low clamp value: The lowest allowed value for the output, if set to 0 then no low clamp is used, low_clamp_value.
- High clamp value: The highest allowed value for the output, if set to 4095 then no high clamp is used, high_clamp_value.
- Desired post-gain offset: How far to shift the curve, desired_postgain_offset.

## Outputs

The outputs of the two-point programming algorithm are shown in Table 1.

## Convert inputs into LSBs

All the calculations use LSBs as the units, so each of the inputs will have to be converted from their current units to LSBs. To do so use the following formulas.

### From Degrees

The formula for converting an angle to LSBs is as follows:

\[ \text{Value} = \text{Round}(\text{angle} \div \text{MaximumAngle}) \times 4095 \]

MaximumAngle is the maximum angle in degrees that can be represented and for the A1330 it is \((360 \div 4096) \times 4095\) or 359.912109375. If angle is a signed number for post-gain_offset, then limit Value to between –2048 and 2047 otherwise limit Value from 0 to 4095.

### From Duty Cycle

The formula for converting a duty cycle to LSBs is as follows:

\[ \text{Value} = \text{Round}((\text{duty_cycle} – 5) \div 90) \times 4095 \]

### From Percentage

The formula for converting a percent full travel to LSBs is as follows:

\[ \text{Value} = \text{Round}(\text{percentage} \div 100) \times 4095 \]

If percentage is a signed number, then limit Value to between –2048 and 2047, otherwise limit Value from 0 to 4095.

## Table 1: Two-Point Programming Outputs

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Memory Location</th>
<th>Description</th>
<th>Min.</th>
<th>Max.</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREGAIN_OFFSET</td>
<td>0x3A [23:12]</td>
<td>Pregain offset (zero adjust), at 12-bit resolution.</td>
<td>0</td>
<td>4095</td>
<td>This value is subtracted from the measured angle value, independent of short stroke.</td>
</tr>
<tr>
<td>SS</td>
<td>0x3B [25]</td>
<td>Enables “short stroke” mode. Gain and Min/Max Input angle checking are enabled.</td>
<td>0</td>
<td>1</td>
<td>1 = enabled</td>
</tr>
<tr>
<td>GAIN</td>
<td>0x3B [12:0]</td>
<td>Gain to apply full dynamic range of the output for a limited input range.</td>
<td>0</td>
<td>8191</td>
<td>Applied gain is 1 plus the total value set in this field. GAIN specified in Fixed Point unsigned form with a binary point of 5.</td>
</tr>
<tr>
<td>CE</td>
<td>0x3C [24]</td>
<td>Enable roll-over</td>
<td>0</td>
<td>1</td>
<td>1 = enabled</td>
</tr>
<tr>
<td>ROE</td>
<td>0x3C [25]</td>
<td>Enable clamps</td>
<td>0</td>
<td>1</td>
<td>1 = enabled</td>
</tr>
<tr>
<td>PO</td>
<td>0x3D [24]</td>
<td>Which magnetic rotation direction results in an increasing output value.</td>
<td>0</td>
<td>1</td>
<td>0 = clockwise 1 = counterclockwise</td>
</tr>
<tr>
<td>POSTGAIN_OFFSET</td>
<td>0x3D [23:12]</td>
<td>The output angular offset to relocate the 0° reference point for the output angle. Applied after GAIN and Min/Max Input angle comparison.</td>
<td>–2048</td>
<td>2047</td>
<td>Represented in signed 2’s complement.</td>
</tr>
<tr>
<td>HIGH_CLAMP</td>
<td>0x3D [11:6]</td>
<td>The output high angle clamp.</td>
<td>0</td>
<td>63</td>
<td></td>
</tr>
<tr>
<td>LOW_CLAMP</td>
<td>0x3D [5:0]</td>
<td>The output low angle clamp.</td>
<td>0</td>
<td>63</td>
<td></td>
</tr>
</tbody>
</table>
From Voltage
Converting from voltage to LSBs is problematic. A rough formula for converting to LSBs is as follows:

\[
\text{value} = \text{Round}((voltage - 0.25) \div 4.5) \times 4095
\]

The offset, range, and linearity of each A1330 is different, so to get the most accurate value, a conversion using the A1330 must be used. See Appendix A for the routine that is used in the DLLs.

Determine direction of slope of curve
The slope of the output curve will always be rising when the direction of rotation of the magnetic field matches the measured_rotation_direction (which was initialized with the value of the PO flag), so in order to get a curve that is falling, the rotation_direction needs to be reversed of that of the measured_rotation_direction and the desired angles swapped.

If desired_postion1 > desired_postion2:
  If measured_rotation_direction = clockwise:
    rotation_direction = counter_clockwise
  else
    rotation_direction = clockwise
  Swap(desired_postion_1, desired_postion_2)
else
  rotation_direction = measured_rotation_direction

Calculate magnitude of change of input considering rollovers and measured direction of rotation
If the measured direction of rotation was clockwise, and measured position 2 is larger than measured position 1, then the magnitude of change is measured position 1 subtracted from measured position 2. If the original direction of rotation was clockwise, and measured position 1 is larger than measured position 2, then the magnitude of change is measured position 1 subtracted from 4096 then added to measured position 2.

If the measured of rotation was counter-clockwise, and measured position 1 is larger than measured position 2, then the magnitude of change is measured position 2 subtracted from measured position 1. If the original direction of rotation was counter-clockwise, and measured position 2 is larger than measured position 1, then the magnitude of change is measured position 2 subtracted from 4096, then added to measured position 1.

To simplify future calculations, measured position 1 and measured position 2 are swapped.

If measured_rotation_direction = counter_clockwise:
  If measured_position_2 > measured_position_1:
    distance = measured_position_1 + (4096 – measured_position_2)
  Else
    distance = measured_position_1 – measured_position_2
  Swap(measured_position_1, measured_position_2)
else
  If position_2 > position_1:
    distance = measured_position_2 – measured_position_1
  Else
    distance = measured_position_2 + (4096 – measured_position_1);
If distance >= 4096:
  distance = distance – 4096
Else If distance < 0:
  distance = distance + 4096;

Calculate gain and verify it is within device limits
The gain is desired position 1 subtracted from desired position 2 and then divided by the magnitude of change calculated in the previous section. The possible range of the gain for the A1330 is from 1 to 31.99609375. To convert the gain value into the gain code value, subtract 1 from the gain value, multiply by 256, then round the resulting number up to be an integer.

\[
gain = (\text{desired_postion}_2 – \text{desired_postion}_1) \div \text{distance}
\]

If gain < 1.0:

The Gain is less than 1.0. The range of the desired angles is less than the range of the input angles. Increase the range of the desired angles or decrease the range of the input angles.

If gain > 31.99609375:

The Gain is too large. Decrease the range of the desired angles or increase the range of the input angles.

\[
gain\_code = \text{Round}((gain – 1.0) \times 256.0)
\]
Post-gain offset will be affected by gain so calculate actual post-gain offset

Using the calculated gain, calculate the actual post gain offset. If the actual post gain offset is outside the post gain offset limits of 2047 and -2048, then set the actual post gain offset to the limit.

\[
\text{calculated\_postgain\_offset} = \text{Round}(\text{desired\_postgain\_offset} \times \text{gain})
\]

If \(\text{calculated\_postgain\_offset} > 2047:\)

\[
\text{postgain\_offset} = \text{Round}(\text{calculated\_postgain\_offset} \div \text{gain})
\]

else If \(\text{calculated\_postgain\_offset} < -2048:\)

\[
\text{postgain\_offset} = \text{Round}(\text{calculated\_postgain\_offset} \div \text{gain})
\]

else

\[
\text{postgain\_offset} = \text{desired\_postgain\_offset}
\]

Pre-gain offset must consider direction of rotation and desired post-gain offset

The pre-gain offset is applied in the processing chain before the direction of rotation flag so to get the desired output, the calculated direction of rotation flag must be used. If the calculated direction of rotation is the same as the measured direction of rotation, then the pre-gain offset is desired position 1 divided by gain and subtracted from measured position 1, then the post gain offset is subtracted.

If the calculated direction of rotation is different than the measured rotation direction, then the pre-gain offset is desired position 1 divided by gain and subtracted from measured position 2, then the post gain offset is subtracted. An additional correction of a full rotation value divided by the gain is added.

\[
\text{If measured\_rotation\_direction} = \text{rotation\_direction}:
\]

\[
\text{pregain\_offset} = \text{measured\_position\_1} \div \text{gain} \quad \text{desired\_position\_1} - \text{measured\_position\_1} \div \text{gain}
\]

else

\[
\text{reversed\_desired\_position\_1} = 4096 - \text{desired\_postion\_1}
\]

If \(\text{reversed\_desired\_postion\_1} > 4095:\)

\[
\text{pregain\_offset} = \text{measured\_position\_2} - (\text{reversed\_desired\_postion\_1} \div \text{gain})
\]

\[
\text{pregain\_offset} = \text{pregain\_offset} + \text{postgain\_offset}
\]

\[
\text{pregain\_offset} = \text{pregain\_offset} + (4096 \div \text{gain})
\]

Calculate clamp values (if desired)

If the clamp values are not set to the limits, then convert the desired clamp values to the codes required.

If the desired clamp values are expressed in volts, then start by calculating the rough clamp values. These values will not be exact because the offset, range, and linearity of each A1330 is different. So, starting with the high clamp value, set the output of the device to maximum and set the high clamp value with the calculated value. Adjust the high clamp value up and down until the clamp is as close to the desired high clamp voltage as possible. Then with the low clamp value, set the output of the device to the minimum and set the low clamp value to the calculated value. Adjust the low clamp value up and down until the clamp is as close to the desired low clamp voltage as possible. The routine used in the DLLs to perform these actions are shown in Appendix B.

If the desired clamp values are not expressed in volts then they should have been converted to LSBs. Using the LSB values, find the closest value in the tables in Appendix C and the index into the table is the clamp code value that should be written into the A1330.

Write values to device

\[
P\text{REGAIN\_OFFSET} = \text{pregain\_offset}
\]
\[
SS = ss
\]
\[
GAIN = \text{gain}
\]
\[
CE = ce
\]
\[
ROE = \text{roe}
\]
\[
PO = po
\]
\[
POST\text{GAIN\_OFFSET} = \text{postgain\_offset}
\]
\[
HIGH\_CLAMP = \text{high\_clamp}
\]
\[
LOW\_CLAMP = \text{low\_clamp}
\]
EXAMPLE 1 – RISING CURVE
(BLUE CURVE IN FIGURE 1)

Table 2: Inputs

<table>
<thead>
<tr>
<th>Input Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>measured_position_1_value</td>
<td>0.813 volts</td>
</tr>
<tr>
<td>measured_position_2_value</td>
<td>1.938 volts</td>
</tr>
<tr>
<td>measured_rotation_direction</td>
<td>clockwise</td>
</tr>
<tr>
<td>desired_position_1_value</td>
<td>1 volt</td>
</tr>
<tr>
<td>desired_position_2_value</td>
<td>4 volts</td>
</tr>
<tr>
<td>desired_low_clamp_value</td>
<td>0.5 volts</td>
</tr>
<tr>
<td>desired_high_clamp_value</td>
<td>4.5 volts</td>
</tr>
<tr>
<td>desired_postgain_offset_value</td>
<td>5 degrees</td>
</tr>
</tbody>
</table>

Convert inputs to LSBs

measured_position_1 = 512 LSBs
measured_position_2 = 1536 LSBs
desired_position_1 = 683 LSBs
desired_position_2 = 3413 LSBs
desired_postgain_offset = 57 LSBs

Determine direction of slope of curve
Since desired_position_1 < desired_position_2:
rotation_direction = measured_rotation_direction

Calculate magnitude of change of input considering rollovers and measured direction of rotation

distance = measured_position_2 – measured_position_1
distance = 1536 – 512
distance = 1024

Calculate gain and verify it is within device limits

gain = (desired_position_2 – desired_position_1) ÷ distance
gain = (3413 – 683) ÷ 1024
gain = 2.666015625

Post-gain offset will be affected by gain so calculate actual post-gain offset

calculated_postgain_offset = desired_postgain_offset × gain
calculated_postgain_offset = 57 × 2.666015625
calculated_postgain_offset = 152

Since calculated postgain_offset is less than limits, use desired postgain offset
postgain_offset = desired_postgain_offset
postgain_offset = 57

Pre-gain offset must consider direction of rotation and desired post-gain offset
The measured_rotation_direction is the same as the rotation_direction so:
pregain_offset = measured_position_1 – (desired_position_1 ÷ gain)
pregain_offset = 512 – (683 ÷ 2.666015625)
pregain_offset = 256
pregain_offset = pregain_offset – postgain_offset
pregain_offset = 256 – 57
pregain_offset = 199

Calculate clamp values (if desired)
Using the code in Appendix B, the clamp values are:
low_clamp = 5
high_clamp = 5

Write the values to the device
First, convert the gain to the code value:
gain_code = (gain – 1.0) × 256
gain_code = (2.666015625 - 1.0) × 256
gain_code = 426

Enable short stroke
ss = 1

Enable Clamps
ce = 1

Disable Rollover
roe = 0
WritePartialMemory(MemoryAccessType.primary, 0x3A, pregain_offset, 23, 12)
WritePartialMemory(MemoryAccessType.primary, 0x3B, ss, 25, 25)
WritePartialMemory(MemoryAccessType.primary, 0x3B, gain_code, 12, 0)
WritePartialMemory(MemoryAccessType.primary, 0x3C, ce, 25, 25)
WritePartialMemory(MemoryAccessType.primary, 0x3C,
EXAMPLE 2 – FALLING CURVE (RED CURVE IN FIGURE 1)

Table 3: Inputs

<table>
<thead>
<tr>
<th>Input Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>measured_position_1_value</td>
<td>3.0 volts</td>
</tr>
<tr>
<td>measured_position_2_value</td>
<td>4.2 volts</td>
</tr>
<tr>
<td>measured_rotation_direction</td>
<td>clockwise</td>
</tr>
<tr>
<td>desired_position_1_value</td>
<td>4 volts</td>
</tr>
<tr>
<td>desired_position_2_value</td>
<td>1 volts</td>
</tr>
<tr>
<td>desired_low_clamp_value</td>
<td>0.5 volts</td>
</tr>
<tr>
<td>desired_high_clamp_value</td>
<td>4.5 volts</td>
</tr>
<tr>
<td>desired_postgain_offset_value</td>
<td>5 degrees</td>
</tr>
</tbody>
</table>

Convert inputs to LSBs

measured_position_1 = 2560 LSBs
measured_position_2 = 3584 LSBs
desired_position_1 = 3413 LSBs
desired_position_2 = 683 LSBs
desired_postgain_offset = 57 LSBs

Determine direction of slope of curve

Since desired_position_1 > desired_position_2 and measured_rotation_direction = clockwise:
rotation_direction = counter_clockwise
and the desired positions are swapped
desired_position_1 = 683 LSBs
desired_position_2 = 3413 LSBs

Calculate magnitude of change of input considering rollovers and measured direction of rotation

distance = measured_position_2 – measured_position_1
distance = 3584 – 2560
distance = 1024

Calculate gain and verify it is within device limits

gain = (desired_position_2 – desired_position_1) ÷ distance
gain = (3413 – 683) ÷ 1024
gain = 2.666015625

Post-gain offset will be affected by gain so calculate actual post-gain offset

calculated_postgain_offset = desired_postgain_offset × gain
calculated_postgain_offset = 57 × 2.666015625
calculated_postgain_offset = 152

Since calculated postgain_offset is less than limits, use desired postgain offset
postgain_offset = desired_postgain_offset
postgain_offset = 57

Pre-gain offset needs to consider direction of rotation and desired post-gain offset

The measured_rotation_direction is the opposite as compared to the rotation_direction so:
reversed_desired_position_1 = 4096 – desired_position_1
reversed_desired_position_1 = 4096 – 683
reversed_desired_position_1 = 3413
Since reversed_desired_position_1 < 4095, the reversed_desired_position_1 value is used.
pregain_offset = measured_position_2 – (reversed_desired_position_1 ÷ gain)
pregain_offset = 3584 – (3413 ÷ 2.666015625)
pregain_offset = 2304
pregain_offset = pregain_offset + postgain_offset
pregain_offset = 2304 + 57
pregain_offset = 2361
pregain_offset = pregain_offset + (4096 ÷ gain)
pregain_offset = 2361 + (4096 ÷ 2.666015625)
pregain_offset = 3897

Calculate clamp values (if desired)

Using the code in Appendix B, the clamp values are:
low_clamp = 5
high_clamp = 5
Write values to device

First, convert the gain to the code value:

\[ \text{gain\_code} = (\text{gain} - 1.0) \times 256 \]
\[ \text{gain\_code} = (2.666015625 - 1.0) \times 256 \]
\[ \text{gain\_code} = 426 \]

Enable short stroke

\( ss = 1 \)

Enable Clamps

\( ce = 1 \)

Disable Rollover

\( roe = 0 \)

WritePartialMemory(MemoryAccessType.primary, 0x3A, pregain_offset, 23, 12)
WritePartialMemory(MemoryAccessType.primary, 0x3B, ss, 25, 25)
WritePartialMemory(MemoryAccessType.primary, 0x3B, gain_code, 12, 0)
WritePartialMemory(MemoryAccessType.primary, 0x3C, ce, 25, 25)
WritePartialMemory(MemoryAccessType.primary, 0x3C, roe, 24, 24)
WritePartialMemory(MemoryAccessType.primary, 0x3D, rotation_direction, 24, 24)
WritePartialMemory(MemoryAccessType.primary, 0x3D, postgain_offset, 23, 12)
WritePartialMemory(MemoryAccessType.primary, 0x3D, high_clamp, 11, 6)
WritePartialMemory(MemoryAccessType.primary, 0x3D, low_clamp, 5, 0)
APPENDIX A

Simple routine to convert from a voltage to LSB for a A1330. Does a binary search by forcing the A1330 to generate the voltage for a certain LSB value and searches for the desired voltage.

```java
public int ConvertVoltageToLSBs(ASEK20_A1330 device, double voltage) {
    uint[] addresses = { 0x2A, 0x2B, 0x2C, 0x2D, 0x2E }; // Save dac trim slope values
    uint[] oldValues = { 0, 0, 0, 0, 0 }; // Save the backend bandwidth selection
    uint[] newValues = { 0, 0, 0, 0, 0 }; // Save the backend bandwidth selection
    int high = 4095;
    int low = 0;
    int mid;
    double value;

    oldValues = device.ReadMemory(MemoryAccessType.shadow, addresses, null);
    newValues[0] = oldValues[0] & 0xFFF; // Save dac trim slope values
    newValues[1] = 0;
    newValues[2] = 0;
    WriteMemory(MemoryAccessType.shadow, addresses, newValues, null);

    // Set the point where the codes will be inserted into the digital chain
    device.WriteMemory(MemoryAccessType.shadow, 1, 4);

    do {
        mid = (high + low) / 2;
        // Force the output to the desired value
        device.WriteMemory(MemoryAccessType.shadow, 0, 0x20000U | (((uint)mid << 4) & 0xFFF0));
        device.FlushCommandBuffer();
        value = device.ReadOutputVoltage(20);

        if (voltage < value) {
            high = mid;
        } else if (voltage > value) {
            low = mid;
        } else {
            break;
        }
    }

    return mid;
}
```
// If the limits are right next to each other,
// see if mid should be up or down.
if (high <= (low + 1))
{
    if (voltage > value)
    {
        mid = high;
    }
    else if (voltage < value)
    {
        mid = low;
    }

    break;
}
}
}

// Restore the old settings
device.WriteMemory(MemoryAccessType.shadow, addresses, oldValues, null);

return mid;
APPENDIX B

Convert the voltage for the clamp values to the code values.

```c
void getClampValues(ASEK20_A1330 device, double lowClampVoltage, double highClampVoltage, out int lowClampCode, out int highClampCode)
{
    uint[] dataAddresses = { 1, 0 };  
    uint[] data = { 0, 0 }; 
    uint[] registerAddresses = { 0x2A, 0x2B, 0x2C, 0x2D }; 
    uint[] registerValues = { 0, 0, 0, 0 }; 
    uint[] oldRegisterValues = { 0, 0, 0, 0 }; 
    double measuredVoltageOffset; 
    double measuredVoltageRange; 

    GetDacOffsetAndRange(device, out measuredVoltageOffset, out measuredVoltageRange); 
    double code0Voltage = measuredVoltageOffset; 
    double code4095Voltage = measuredVoltageRange + measuredVoltageOffset; 

    oldRegisterValues = device.ReadMemory(MemoryAccessType.shadow, registerAddresses, null); 

    registerValues[0] = ASEK.SetBitfield(oldRegisterValues[0], 0, 23, 12); 
    registerValues[1] = 0; 
    registerValues[2] = 0x2000000; 
    registerValues[3] = ASEK.SetBitfield(oldRegisterValues[3], 0, 24, 0); 

    device.WriteMemory(MemoryAccessType.shadow, registerAddresses, registerValues, null); 

    // Calculate the low clamp code starting value 
    lowClampCode = 0; 
    if (lowClampVoltage >= code0Voltage) 
    { 
        lowClampCode = (int)(Math.Abs(lowClampVoltage - code0Voltage) / 0.05); 
    }

    if (lowClampCode > 63) 
    { 
        lowClampCode = 63; 
    } 
    else if (lowClampCode < 0) 
    { 
        lowClampCode = 0; 
    }

    // Calculate the high clamp code starting value 
    highClampCode = 0; 
    if (highClampVoltage <= code4095Voltage) 
    { 
        highClampCode = (int)(Math.Abs(code4095Voltage - highClampVoltage) / 0.05); 
    }
```
if (highClampCode > 63)
{
    highClampCode = 63;
}
else if (highClampCode < 0)
{
    highClampCode = 0;
}

registerValues[3] = ASEK.SetBitfield(registerValues[3], highClampCode, 11, 6); // HIGH_CLAMP
registerValues[3] = ASEK.SetBitfield(registerValues[3], lowClampCode, 5, 0); // LOW_CLAMP

device.WriteMemory(MemoryAccessType.shadow, 0x2D, registerValues[3]);

// Set the point where the codes will be inserted into the digital chain
data[0] = 3;
data[1] = 0x2FFF0U;
device.WriteMemory(MemoryAccessType.shadow, dataAddresses, data, null);
device.FlushCommandBuffer();

double previous_value = device.ReadOutputVoltage(10);
double value = previous_value;
if (value > highClampVoltage)
{
    while (value > highClampVoltage)
    {
        previous_value = value;
        ++highClampCode;

        if (highClampCode >= 64)
        {
            highClampCode = 63;
            break;
        }
        registerValues[3] = ASEK.SetBitfield(registerValues[3], highClampCode, 11, 6); // HIGH_CLAMP
        device.WriteMemory(MemoryAccessType.shadow, 0x2D, registerValues[3]);
        device.FlushCommandBuffer();

        value = device.ReadOutputVoltage(10);
    }
    if (Math.Abs(highClampVoltage - previous_value) < Math.Abs(highClampVoltage - value))
    {
        --highClampCode;
    }
}
else if (value < highClampVoltage)
{
while (value < highClampVoltage)
{
    previous_value = value;
    --highClampCode;
    if (highClampCode < 0)
    {
        highClampCode = 0;
        break;
    }
    registerValues[3] = ASEK.SetBitfield(registerValues[3], highClampCode, 11, 6); // HIGH_CLAMP
    device.WriteMemory(MemoryAccessType.shadow, 0x2D, registerValues[3]);
    device.FlushCommandBuffer();

    value = device.ReadOutputVoltage(10);
}
if (Math.Abs(highClampVoltage - previous_value) < Math.Abs(highClampVoltage - value))
{
    ++highClampCode;
}

// Set the point where the codes will be inserted into the digital chain
data[0] = 3;
data[1] = 0x20000U;
device.WriteMemory(MemoryAccessType.shadow, dataAddresses, data, null);
device.FlushCommandBuffer();

previous_value = device.ReadOutputVoltage(10);
value = previous_value;
if (value < lowClampVoltage)
{
    while (value < lowClampVoltage)
    {
        previous_value = value;
        ++lowClampCode;

        if (lowClampCode >= 64)
        {
            lowClampCode = 63;
            break;
        }
        registerValues[3] = ASEK.SetBitfield(registerValues[3], lowClampCode, 5, 0); // LOW_CLAMP
        device.WriteMemory(MemoryAccessType.shadow, 0x2D, registerValues[3]);
        device.FlushCommandBuffer();

        value = device.ReadOutputVoltage(10);
    }
if (Math.Abs(lowClampVoltage - previous_value) < Math.Abs(lowClampVoltage - value))
{
    --lowClampCode;
}
else if (value > lowClampVoltage)
{
    while (value > lowClampVoltage)
    {
        previous_value = value;
        --lowClampCode;
        if (lowClampCode < 0)
        {
            lowClampCode = 0;
            break;
        }
        registerValues[3] = ASEK.SetBitfield(registerValues[3], lowClampCode, 5, 0);  // LOW_CLAMP
        device.WriteMemory(MemoryAccessType.shadow, 0x2D, registerValues[3]);
        device.FlushCommandBuffer();
        value = device.ReadOutputVoltage(10);
    }
    if (Math.Abs(lowClampVoltage - previous_value) < Math.Abs(lowClampVoltage - value))
    {
        ++lowClampCode;
    }
}

data[0] = 7;  // Unforce input
data[1] = 0;
device.WriteMemory(MemoryAccessType.shadow, dataAddresses, data, null);
device.WriteMemory(MemoryAccessType.shadow, registerAddresses, oldRegisterValues, null);

public void GetDacOffsetAndRange(ASEK20_A1330 device, out double offset, out double range)
{
    double voltage;
    double voltageLow;
    double voltageMid;
    double voltageHigh;
    double correctionPercentage;
    int inset = 512;
    uint[] addresses = { 0x2A, 0x2B, 0x2C, 0x2D, 0x2E };  // LOW_CLAMP
    uint[] oldValues = { 0, 0, 0, 0, 0 };
    uint[] newValues = { 0, 0, 0, 0, 0 };
```csharp
uint reg23 = device.ReadMemory(MemoryAccessType.shadow, 0x23);

oldValues = device.ReadMemory(MemoryAccessType.shadow, addresses, null);
newValues[0] = oldValues[0] & 0xFFF; // Save dac trim slope values
newValues[1] = 0;
newValues[2] = 0;
device.WriteMemory(MemoryAccessType.shadow, addresses, newValues, null);

// Set the point where the codes will be inserted into the digital chain
device.WriteMemory(MemoryAccessType.shadow, 1, 4);
device.WriteMemory(MemoryAccessType.shadow, 0, 0x20000U | (uint)((inset << 4) & 0xFFF0)); // Force input
device.FlushCommandBuffer();

voltageLow = device.ReadOutputVoltage(20);

// Set the point where the codes will be inserted into the digital chain
device.WriteMemory(MemoryAccessType.shadow, 1, 4);
device.WriteMemory(MemoryAccessType.shadow, 0, 0x20000U | (uint)((4096 - inset) << 4) & 0xFFF0)); // Force input
device.FlushCommandBuffer();

voltageHigh = device.ReadOutputVoltage(20);

// Set the point where the codes will be inserted into the digital chain
device.WriteMemory(MemoryAccessType.shadow, 1, 4);
device.WriteMemory(MemoryAccessType.shadow, 0, 0x20000U | (uint)((4096 - inset) << 4) & 0xFFF0)); // Force input
device.FlushCommandBuffer();

voltageMid = device.ReadOutputVoltage(20);

device.WriteMemory(MemoryAccessType.shadow, 1, 7);
device.WriteMemory(MemoryAccessType.shadow, 0, 0); // Unforce input
device.WriteMemory(MemoryAccessType.shadow, addresses, oldValues, null);
device.FlushCommandBuffer();

double voltsPerLSB = (voltageHigh - voltageLow) / (4096 - (inset * 2));
double voltageOffset = voltageMid - (voltsPerLSB * 2048);
double voltageRange = voltsPerLSB * 4096.0;

if (GetBitfield(reg23, 20, 20) == 0)
{
    voltage = device.GetOutputVoltage(20, SupplyVoltageADC); // Measure Vcc
correctionPercentage = 5.0 / voltage;
    voltsPerLSB *= correctionPercentage;
    voltageOffset *= correctionPercentage;
}
```
voltageRange *= correctionPercentage;
}

offset = voltageOffset;
range = voltageRange;
}
APPENDIX C

LSB values corresponding to the clamp code values.

```c
int[] lowClampValues = {
    0,   46,   91,  137,  182,  228,  273,  319,
    364,  410,  455,  501,  546,  592,  637,  683,
    728,  774,  819,  865,  910,  956, 1001, 1047,
   1092, 1138, 1183, 1229, 1274, 1320, 1365, 1411,
   1456, 1502, 1547, 1593, 1638, 1684, 1729, 1775,
   1820, 1866, 1911, 1957, 2002, 2048, 2094, 2139,
   2185, 2230, 2276, 2321, 2367, 2412, 2458, 2503,
   2549, 2594, 2640, 2685, 2731, 2776, 2822, 2867
};
```

```c
int[] highClampValues = {
    4095, 4050, 4005, 3959, 3914, 3868, 3823, 3777,
    3732, 3686, 3641, 3595, 3550, 3504, 3459, 3413,
    3368, 3322, 3277, 3231, 3186, 3140, 3095, 3049,
    3004, 2958, 2913, 2867, 2822, 2776, 2731, 2685,
    2640, 2594, 2549, 2503, 2458, 2412, 2367, 2321,
    2276, 2230, 2185, 2139, 2094, 2048, 2002, 1957,
    1911, 1866, 1820, 1775, 1729, 1684, 1638, 1593,
   1547, 1502, 1456, 1411, 1365, 1320, 1274, 1229
};
```
APPENDIX D

Example using the DLL routines.

```csharp
Dictionary<string, object> optionsDie1 = new Dictionary<string, object>();
Dictionary<string, object> optionsDie2 = new Dictionary<string, object>();
double desiredPosition1Die1 = 0.5;
double desiredPosition2Die1 = 4.5;
double desiredPosition1Die2 = 4.5;
double desiredPosition2Die2 = 0.5;

// Move the target to position 1
device.SetDieSelection(0);
object position1DataDie1 = device.TPPLGetPosition1Information(null, null);
device.SetDieSelection(1);
object position1DataDie2 = device.TPPLGetPosition1Information(null, null);

// Move the target to position 2
device.SetDieSelection(0);
object position2DataDie1 = device.TPPLGetPosition2Information(null, null);
device.SetDieSelection(1);
object position2DataDie2 = device.TPPLGetPosition2Information(null, null);

optionsDie1["Write To Chip"] = true;
optionsDie1["Low Clamp Value"] = 0.4;
optionsDie1["High Clamp Value"] = 4.6;
optionsDie1["Post Gain Offset Value"] = 0.0;
optionsDie1["Clamp Enable"] = true;
optionsDie1["Rollover Enable"] = false;
optionsDie1["Input Units"] = "degrees";
optionsDie1["Output Units"] = "volts";
optionsDie1["Diagnostics"] = true;

optionsDie2["Write To Chip"] = true;
optionsDie2["Low Clamp Value"] = 0.4;
optionsDie2["High Clamp Value"] = 4.6;
optionsDie2["Post Gain Offset Value"] = 0.0;
optionsDie2["Clamp Enable"] = true;
optionsDie2["Rollover Enable"] = false;
optionsDie2["Input Units"] = "degrees";
optionsDie2["Output Units"] = "volts";
optionsDie2["Diagnostics"] = true;

device.SetDieSelection(0);
device.TPPLCalculate(position1DataDie1, position2DataDie1, desiredPosition1Die1, desiredPosition2Die1, optionsDie1, null);
device.SetDieSelection(1);
device.TPPLCalculate(position1DataDie2, position2DataDie2, desiredPosition1Die2, desiredPosition2Die2, optionsDie2, null);
```
Copyright 2020, Allegro MicroSystems.

The information contained in this document does not constitute any representation, warranty, assurance, guaranty, or inducement by Allegro to the customer with respect to the subject matter of this document. The information being provided does not guarantee that a process based on this information will be reliable, or that Allegro has explored all of the possible failure modes. It is the customer’s responsibility to do sufficient qualification testing of the final product to insure that it is reliable and meets all design requirements.

Copies of this document are considered uncontrolled documents.