life.augmented

AN5888

Application note

Getting started with the CORDIC accelerator using Stellar E

Introduction

This document applies to the SR5E1x 32-bit Arm® Cortex®-M7 microcontrollers with CORDIC accelerator unit.

The CORDIC accelerator provides hardware support of certain mathematical functions (mainly trigonometric ones) commonly used in motor control, metering, signal processing. It speeds up the calculation of these functions compared to a software implementation, making it possible the use of a lower operating frequency, or freeing up processor cycles in order to perform other tasks.

This application note describes the main features of the CORDIC accelerator on SR5E1x microcontrollers. The document also describes the capabilities and limitations of the CORDIC accelerator and evaluates the speed of execution for certain calculations typically needed in electric motor control compared with the equivalent table implementation.

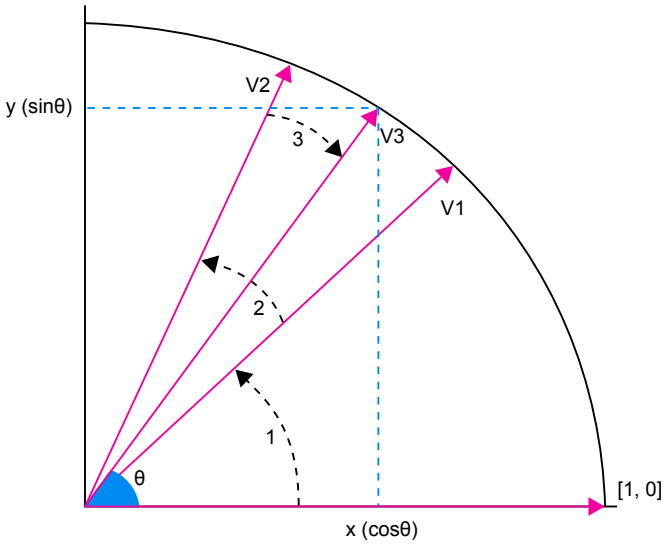
The code used in this application note is included in the SR5E1x motor control software package and running on the motor control board (please refer to STMicroelectronics local representative for reference hardware and software).

1 CORDIC introduction

The CORDIC (coordinate rotation digital computer), also known as the algorithm of Volder, is a low-cost successive approximation algorithm for evaluating trigonometric and hyperbolic functions.

In trigonometric (or circular) mode, the sine and cosine of an angle *θ* are determined by rotating the vector [1, 0] through decreasing angles *atan(2^{-n}) (n = 0, 1, 2,...)* until the cumulative sum of the rotation angles equals the input angle. The x and y cartesian components of the rotated vector then correspond respectively to the cosine and sine of *θ*. This is illustrated in the Figure 1 for an angle *θ*.

Figure 1. CORDIC circular mode operation



More formally, every iteration calculates a rotation which is done by multiplying vector v_i = *(xi, yi)* with rotation matrix R_i:

$$v_{i+1} = R_i v_i$$

The rotation matrix is given by:

$$R_i = \begin{bmatrix} \cos(\gamma_i) & -\sin(\gamma_i) \\ \sin(\gamma_i) & \cos(\gamma_i) \end{bmatrix}$$

Using the following two identities:

$$\cos\left(\gamma_i\right) = \frac{1}{\sqrt{1+\tan^2(\gamma_i)}}$$

$$\sin\left(\gamma_i\right) = \frac{\tan(\gamma_i)}{\sqrt{1+\tan^2(\gamma_i)}}$$

we have:

$$\begin{bmatrix} x_i+1 \\ y_i+1 \end{bmatrix} = \frac{1}{\sqrt{1+\tan^2(\gamma_i)}} \begin{bmatrix} 1 & -\tan(\gamma_i) \\ \tan(\gamma_i) & 1 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix}$$

Restricting the angles in order that $\tan(\gamma_i) = \pm\left(2^{-i}\right)$, the multiplication with the tangent can be replaced dividing by a power of two, which is efficiently done in digital computer hardware using a bit-shift operation.

The expression becomes:

$$\begin{bmatrix} x_i+1 \\ y_i+1 \end{bmatrix} = K_i \begin{bmatrix} 1 & -\sigma_i 2^{-i} \\ \sigma_i 2^{-i} & 1 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \end{bmatrix} \quad (1)$$

where:

$$K_i = \frac{1}{\sqrt{1 + 2^{-2i}}}$$

and σ_i is used to determine the direction of the rotation: if the angle γ_i is positive, then σ_i is +1, otherwise it is −1.

All K_i value can be ignored in an iterative process and applied *K(n)* at the end of the *n* iterations:

$$K\left(n\right) = \prod_{i=0}^{n-1} K_i = \prod_{i=0}^{n-1} \frac{1}{\sqrt{1 + 2^{-2i}}}$$

The previous Equation 1 shows that the CORDIC algorithm is suited to an hardware implementation since there are no multiplications involved (applying *K(n)* at the end of iterations). Only add and shift operations are performed.

Inversely, the angle of a vector [x, y] is determined by rotating [x, y] through successively decreasing angles to obtain the unit vector [1, 0]. The cumulative sum of the rotation angles gives the angle of the original vector, corresponding to the *atan (y/x)*.

The CORDIC algorithm can also be used for calculating hyperbolic functions (*sinh*, *cosh*, *atanh*) by replacing the circular rotations with hyperbolic angles *atanh(2^{-j}) (with j = 1, 2, 3...)*, see Figure 2.

Other functions can be derived from the basic functions described above.

The natural logarithm is a special case of the inverse hyperbolic tangent, obtained from the identity:

$$\ln x = 2\tanh^{-1}\left(\frac{x+1}{x-1}\right)$$

The square root function is also a special case of the inverse hyperbolic tangent. When calculating the *atanh* the CORDIC also calculates $\sqrt{(\cosh^2 t - \sinh^2 t)}$. So, the square root is obtained from:

$$\sqrt{x} = \sqrt{\left(x + \frac{1}{4}\right)^2 - \left(x - \frac{1}{4}\right)^2}$$

Additional functions are calculated from the above using the following appropriate identities:

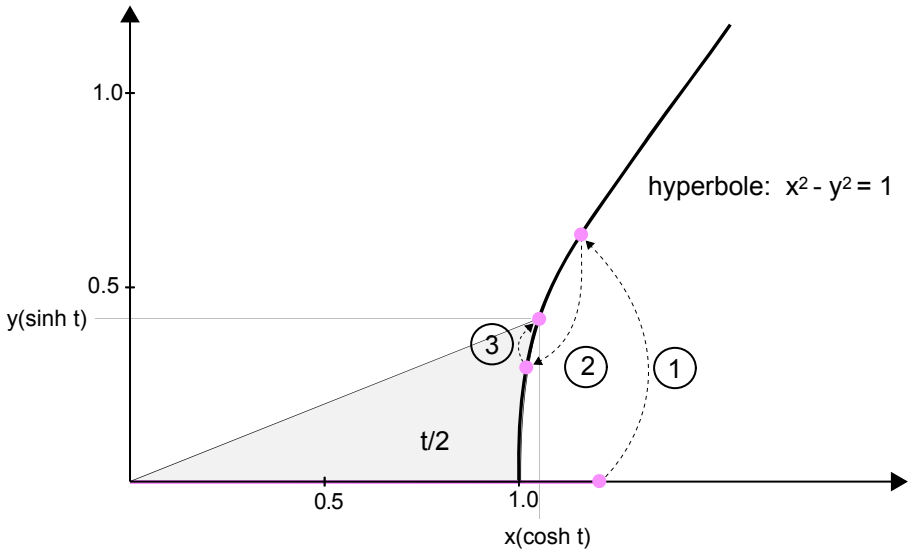
$$\tanh\left(x\right) = \frac{\sinh(x)}{\cosh(x)}$$

$$e^x = \sinh(x) + \cosh(x)$$

$$\log_2 x = (\log_2 e)(\ln x) = 1.442695041(\ln x)$$

$$\log_{10} x = (\log_{10} e)(\ln x) = 0.434294482(\ln x)$$

Figure 2. CORDIC hyperbolic mode operation



1.1 Limitations

In this paragraph are described the limitations of the functions input ranges of the CORDIC co-processor. The CORDIC operates in fixed point signed integer format. Input and output values can be either q1.31 or q1.15.

In q1.31 format, numbers are represented by one sign bit and 31 fractional bits (binary decimal places). The numeric range is therefore -1 (0x80000000) to 1 - 2^{-31} (0x7FFFFFFF).

In q1.15 format, the numeric range is -1 (0x8000) to 1 - 2^{-15} (0x7FFF). This format has the advantage that two input arguments can be packed into a single 32-bit write, and two results can be fetched in one 32-bit read.

For certain functions (*atan*, *log*, *sqrt*) a scaling factor can be applied to extend the range of the function beyond the maximum [-1, 1).

Following the list of mathematical functions implementable with CORDIC.

- Functions: cos(x) - sin(x) – phase(x, y) - modules(x, y)
  - Input range: -1 ≤ x < 1 ; -1 ≤ y < 1;
  - All input angles in the range -π to π radians are scaled multiplying by 1/π

In circular mode, the CORDIC converges for all angles in the range -π to π radians. The use of fixed point representation means that input and output values must be in the range [-1 to 1). Input angles in radians must be multiplied by 1/π and output angles must be multiplied by π to convert back to radians.

The modulus must be in the range [0 to 1), whether converting from polar to rectangular or rectangular to polar. This means that phase (1.0, 1.0) gives false results since the modulus (√2) is out of range and saturates the CORDIC engine, even if only the phase is needed.

- Functions: $atan^{-1}(x)$
  - Input range: -128 ≤ x < 128
  - If |x| > 1, a scaling factor of 2^{-n} with n = [0 7]
  - The output of CORDIC co-processor must be multiplied by (2^n)* π to obtain the angle in radians. The maximum input value allowed is *tan θ = 128*, which corresponds to an angle: θ = 89.55 degrees. For |x| > 128, a software method must be used to find *atan(x)*
- Functions: cosh (x) - sinh (x)
  - Input range -1.118 ≤ x ≤1.118
  - The input is scaled multiplying by 2^{-n} with n = 1
  - The output of CORDIC co-processor must be multiplied by 2 to obtain the correct value

In hyperbolic mode, *y = cosh x* being defined only for *y ≥ 1*, all inputs and outputs are divided by 2 to remain in the fixed point numeric range. The CORDIC algorithm stops converging for |x| > 1.118. This implies that only values of x in the range -1.118 to 1.118 are supported.

So, 1.118 is the effective limit for the hyperbolic sine and hyperbolic cosine of input magnitude function. If the value is constrained to be smaller than this limit, then the CORDIC is used directly. Otherwise, the magnitude of *x* must be tested and if it is above the limit, an alternative software algorithm must be used.

- Function: $\tanh^{-1}(x)$
  - Input range -0.806 ≤ x < 0.806
  - The input is scaled multiplying by 2^{-n} with n = 1
  - The output of CORDIC co-processor must be multiplied by 2 to obtain the correct value

The magnitude of the $tanh^{-1}(x)$ function output is similarly limited to 1.118, hence the input magnitude is limited to *tanh 1.118 = 0.806*.

- Function: natural log
  - Input range -0.107 ≤ x < 9.35
  - The input is scaled multiplying by 2^{-n} with n = [1 4]
  - The output of CORDIC co-processor must be multiplied by 2^{n+1} to obtain the correct value

  As previously stated, the natural log uses the identity:

$$\ln x = 2\tanh^{-1}\left(\frac{x+1}{x-1}\right)$$

  Since the limit for the *tanh^{-1}(x)* input magnitude is 0.806, then:

$$\left|\frac{x+1}{x-1}\right| < 0.806 \Rightarrow |x| < 9.35$$

  Hence, the limit for the natural log input is 9.35. Furthermore, the input must be scaled by the appropriate power of 2, such that $(2^{-n})^*$ (x+1) < 1, to avoid overflowing the fixed point numerical format.
  To summarize, for natural log calculation, the allowed input of x is in the range 0.107 to 9.35 using the following scaling:
  - 0.107 ≤ x < 1 the input is scaled multiplying by 2^{-n} with n = 1
  - 1 ≤ x < 3 the input is scaled multiplying by 2^{-n} with n = 2
  - 3 ≤ x < 7 the input is scaled multiplying by 2^{-n} with n = 3
  - 7 ≤ x ≤ 9.35 the input is scaled multiplying by 2^{-n} with n = 4
- Function: square root
  - Input range 0.027 ≤ x < 2.341
  - The input is scaled multiplying by 2^{-n} with n = [0 2]
  - The output of CORDIC co-processor must be multiplied by 2^{n} to obtain the correct value

Since the limit for the *tanh^{-1}(x)* function is given by:

$$\tanh(t) = \frac{\sinh(t)}{\cosh(t)} \leq 0.806$$

and the inputs to the *tanh^{-1}(x)* function are *sinh (t)* and *cosh (t)*, then the input to the square root function must satisfy the expression:

$$\frac{x-0.25}{x+0.25} = \frac{\sinh(t)}{\cosh(t)} \leq 0.806$$

Hence, x ≤ 2.34. Again, scaling must be applied such that $(2^{-n})^*(x+0.25) < 1$ to avoid saturation.

To summarize, for square root, the allowed input of x is in the range 0.027 to 2.34 using the following scaling:

- 0.027 ≤ x < 0.75 the input is scaled multiplying by 2^{-n} (n = 0)
- 0.75 ≤ x < 1.75 the input is scaled multiplying by 2^{-n} (n = 1)
- 1.75 ≤ x ≤ 2.341 the input is scaled multiplying by 2^{-n} (n = 2)

1.2 Convergence rate and precision

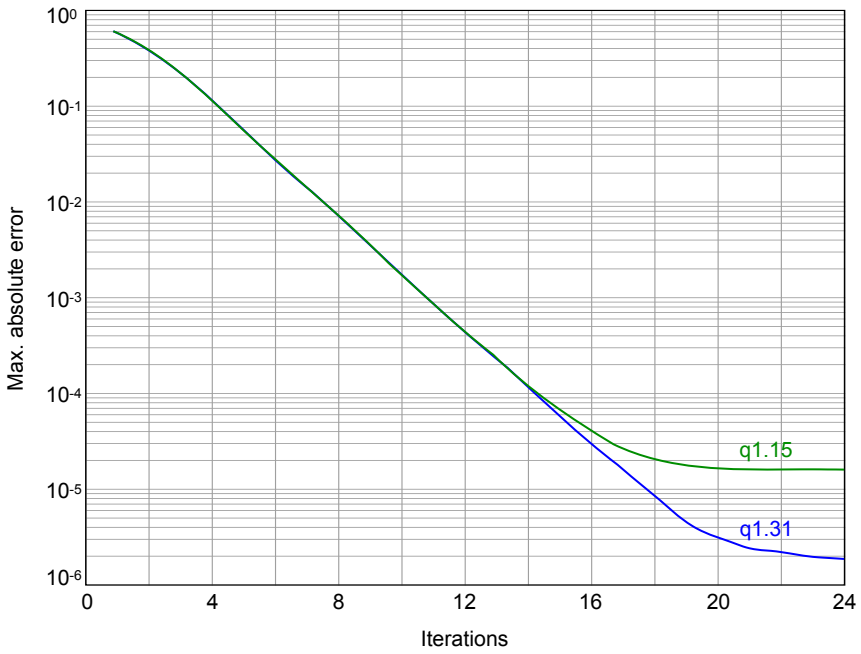
In circular mode, the CORDIC algorithm converges at a rate of 1 binary digit per iteration. This means that 16 iterations are required to achieve 16-bit precision. The maximum achievable precision is limited by the number of bits in the CORDIC “engine” (the shifters and adders, as well as the table used to store the successive rotation angles), and of course in the input and output registers. The CORDIC unit in the SR5E1x microcontrollers supports 16-bit and 32-bit input and output data, and has an internal precision of 24 bits.

The Figure 3 shows the rate of convergence for the CORDIC in circular mode. The curve labeled “q1.15” uses 16-bit input and output data. The curve labeled “q1.31” uses 32-bit data. In both cases, the CORDIC “engine” is 24-bit. When the maximum residual error approaches the limit for 16-bit precision (2^{-16} = 1.5 x 10^{-5}), the quantization error caused by truncating the result to 16 bits becomes dominant, and the curve flattens out, even though the CORDIC engine continues to converge. For 32-bit data it is the quantization error of the CORDIC engine itself, which starts to become significant after around 20 iterations. After 24 iterations, the successive rotation angle becomes zero and no more convergence is possible. The maximum residual error in this case is 1.9 x 10^{-6}, which corresponds to 19-bit precision.

In hyperbolic mode, a particularity of the algorithm requires that certain iterations must be performed twice in order to converge over the whole range. In a 24-bit CORDIC, the fourth and the 14^{th} iterations are repeated, that is to say that the same rotation angle is applied twice in a row instead of dividing by two. This means that the convergence is not linear, as seen in Figure 4. The error decreases more gradually at the beginning and remains the same between iteration 14 and 15. Therefore, the CORDIC takes two more iterations to achieve the same precision in hyperbolic mode than in circular mode.

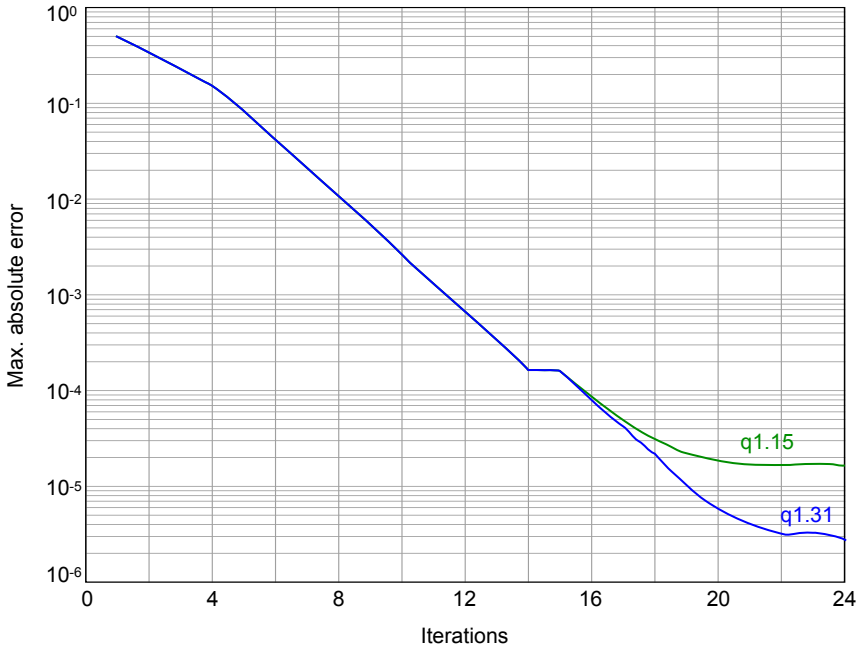
Square root is an exception to the above. It converges approximately twice as fast as the other hyperbolic mode functions.

Figure 3. CORDIC convergence (circular mode)



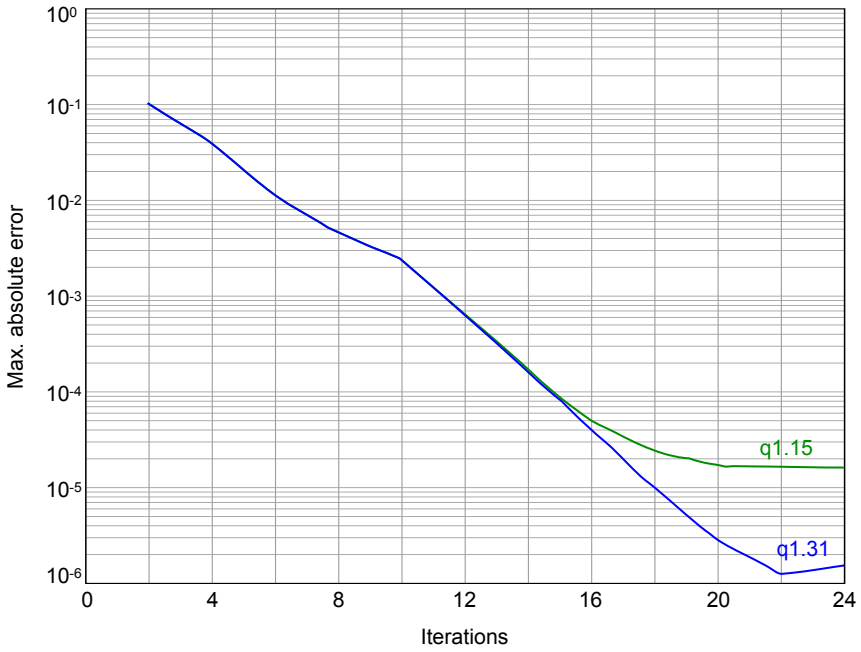
Note: Sine/cosine convergence of q1.23 fixed point CORDIC.

Figure 4. CORDIC convergence (hyperbolic mode)



Note: Hyperbolic sine/cosine convergence of q1.23 fixed point CORDIC.

Figure 5. CORDIC convergence (square root)



Note: Square root convergence of q1.23 fixed point CORDIC.

2 Code examples and performance

2.1 Forward and reverse park

The performances of CORDIC are assessed for two mathematical transforms used in the field-oriented control algorithm to run an electric motor: the forward park transformation and the reverse park transformation.

The forward park transforms the two-axis orthogonal stationary reference frame quantities into the rotating reference frame quantities by the following equations:

$$I_d = I_\alpha \times \cos(\theta) + I_\beta \times \sin(\theta)$$
$$I_q = I_\beta \times \cos(\theta) - I_\alpha \times \sin(\theta)$$

where, I_d, I_q are rotating reference frame quantities, I_α, I_β are orthogonal stationary reference frame quantities, θ is the rotation angle.

The inverse park transforms the rotating reference frame quantities into two-axis orthogonal stationary reference frame quantities by the following equations:

$$V_\alpha = V_d \times \cos(\theta) - V_q \times \sin(\theta)$$
$$V_\beta = V_q \times \cos(\theta) + V_d \times \sin(\theta)$$

where, V_α, V_β are orthogonal stationary reference frame quantities, V_d, V_q are rotating reference frame quantities.

At the base of the two park transformations there is the calculation of the trigonometric functions *cos(θ)* and *sin(θ)* and three methods have been used for their calculation: CORDIC co-processor, a sin/cos table-based calculation and the function *arm_sin_cos_q31(q31_t, q31_t*, q31_t*)* available in ARM DSP 32-bit fixed point library. The respective performances have then been compared using SR5E1x electric motor control demonstrator software and hardware.

Below details about CORDIC coprocessor configuration and its use are given. Three steps are needed for CORDIC use:

1. Configure the CORDIC:

```
WRITE_REG(CORDIC->CSR, CORDIC_CONFIG_COSINE);

/* CORDIC FUNCTION: COSINE q1.15 */
#define CORDIC_CONFIG_COSINE (LL_CORDIC_FUNCTION_COSINE | LL_CORDIC_PRECISION_4CYCLES |
 LL_CORDIC_SCALE_0 | LL_CORDIC_NBWRITE_1 | LL_CORDIC_NBREAD_1 |                 LL_CORD
IC_INSIZE_16BITS | LL_CORDIC_OUTSIZE_16BITS)
```

   If only this configuration is used, this step is done once at initialization. Otherwise, it must be repeated each time one of the above parameters changes.

2. Write the input argument(s):
   In this case there is only one argument, the angle for the park transforms calculation. The other argument is the default modulus of 1, so does not need to be written.

```
LL_CORDIC_WriteData(CORDIC, 0x7FFF0000 + uint32_t) hAngle);
```

   As soon as the expected number of arguments is written, the calculation starts

3. Read the result(s):

```
/* Read cosine */
CosSin.CordicRdata = LL_CORDIC_ReadData(CORDIC);
```

The function *Trig_Components MCM_Trig_Functions(int16_t hAngle)*, available in SR5E1x electric motor control demonstrator software, executes the three above mentioned steps.

```
/**
* @brief  This function returns cosine and sine functions of the angle fed in input
* @param  hAngle: angle in q1.15 format
* @retval Sin(angle) and Cos(angle) in Trig_Components format
*/

__weak Trig_Components MCM_Trig_Functions( int16_t hAngle )
{
union u32toi16x2 {
uint32_t CordicRdata;
Trig_Components Components;
} CosSin;
/* Configure CORDIC */
```

```
WRITE_REG(CORDIC->CSR, CORDIC_CONFIG_COSINE);
/* Write angle */
LL_CORDIC_WriteData(CORDIC, 0x7FFF0000 + (uint32_t) hAngle);
/* Read angle */
CosSin.CordicRdata = LL_CORDIC_ReadData(CORDIC);
return (CosSin.Components);
}
```

The following code shows an alternative implementation of the *sin(θ) cos(θ)* using a sin/cos table-based calculation:

```
/**
  * @brief  This function returns cosine and sine functions of the angle fed in
  *         input
  * @param  hAngle: angle in q1.15 format
  * @retval Sin(angle) and Cos(angle) in Trig_Components format
  */

Trig_Components MCM_Trig_Functions(int16_t hAngle)
{
  int32_t shindex;
  uint16_t uhindex;

  Trig_Components Local_Components;

  /* 10 bit index computation  */
  shindex =((int32_t)32768 + (int32_t)hAngle);
  uhindex = (uint16_t)shindex;
  uhindex /= (uint16_t)64;


  switch ((uint16_t)(uhindex) & SIN_MASK)
  {
  case U0_90:
    Local_Components.hSin = hSin_Cos_Table[(uint8_t)(uhindex)];
    Local_Components.hCos = hSin_Cos_Table[(uint8_t)(0xFFu-(uint8_t)(uhindex))];
    break;

  case U90_180:
     Local_Components.hSin = hSin_Cos_Table[(uint8_t)(0xFFu-(uint8_t)(uhindex))];
     Local_Components.hCos = -hSin_Cos_Table[(uint8_t)(uhindex)];
    break;

  case U180_270:
     Local_Components.hSin = -hSin_Cos_Table[(uint8_t)(uhindex)];
     Local_Components.hCos = -hSin_Cos_Table[(uint8_t)(0xFFu-(uint8_t)(uhindex))];
    break;

  case U270_360:
     Local_Components.hSin =  -hSin_Cos_Table[(uint8_t)(0xFFu-(uint8_t)(uhindex))];
     Local_Components.hCos =  hSin_Cos_Table[(uint8_t)(uhindex)];
    break;
  default:
    break;
  }
  return (Local_Components);
```

Finally, we have the implementation of the *sin(θ) cos(θ)* calculation using the function *arm_sin_cos_q31(int32_t theta, int32_t *pSinVal, int32_t *pCosVal)* of the ARM DSP 32-bit fixed point library, calculation where *hAngle* is shifted of 16 bits and casted to *int32_t*:

```
/**
  * @brief  This function returns cosine and sine functions of the angle fed in
  *         input
  * @param  hAngle: angle in q1.15 format
  * @retval Sin(angle) and Cos(angle) in Trig_Components format
  */

Trig_Components MCM_Trig_Functions(int16_t hAngle)
{

  Trig_Components Local_Components;

   int32_t hAngle32;
   int32_t *pSinVal;
   int32_t *pCosVal;
   int32_t add1, add2;
   pSinVal=&add1;
   pCosVal=&add2;

   hAngle32=(int32_t)(hAngle<<16);
   arm_sin_cos_q31(hAngle32,pSinVal,pCosVal);

   Local_Components.hSin = (int16_t) (*pSinVal>>16);
   Local_Components.hCos = (int16_t) (*pCosVal>>16);

  return (Local_Components);
}
```

SR5E1x electric motor control software implements the two park transformations in the following functions:

- *MCM_Park(Ialphabeta, hElAngle)*
- *MCM_Rev_Park(Vqd, hElAngle)*

The execution time of this two functions was measured for each one of the three alternative previously described implementations of *MCM_Trigger_Functions(int16_t, hAngle)* and the results have been compared.

2.1.1 Execution time measuring

The user can measure the number of processor cycles required to execute the *MCM_Park* and *MCM_Rev_Park* as difference of the DWT counter values read before the start of function and read again after the end of function using *dwt_get_ticks()* function.

```
/* Read systick counter */
start_tick=dwt_get_ticks();
/* Write angle */
Iqd = MCM_Park(Ialphabeta, hElAngle);
/* Read systick counter */
Stop_ticks=dwt_get_ticks();
/* Calculate number of cycles elapsed */
elapsed_ticks = start_ticks-stop_ticks;

/* Read systick counter */
start_tick=dwt_get_ticks();
/* Write angle */
Valphabeta = MCM_Rev_Park(Vqd, hElAngle);
/* Read systick counter */
Stop_ticks=dwt_get_ticks();
/* Calculate number of cycles elapsed */
elapsed_ticks = start_ticks-stop_ticks;
```

2.1.2 Performance comparison

The following performance figures in Table 1 are obtained using SR5E1x 32-bit Arm® Cortex®-M7 microcontrollers running at 300 MHz from flash memory. The code is compiled using the GNU ARM Cross C compiler. The optimization is set to high for speed and for size.

The average CPU cycle values are shown in the table, since on each electric motor control period the park transformations calculation give variable results.

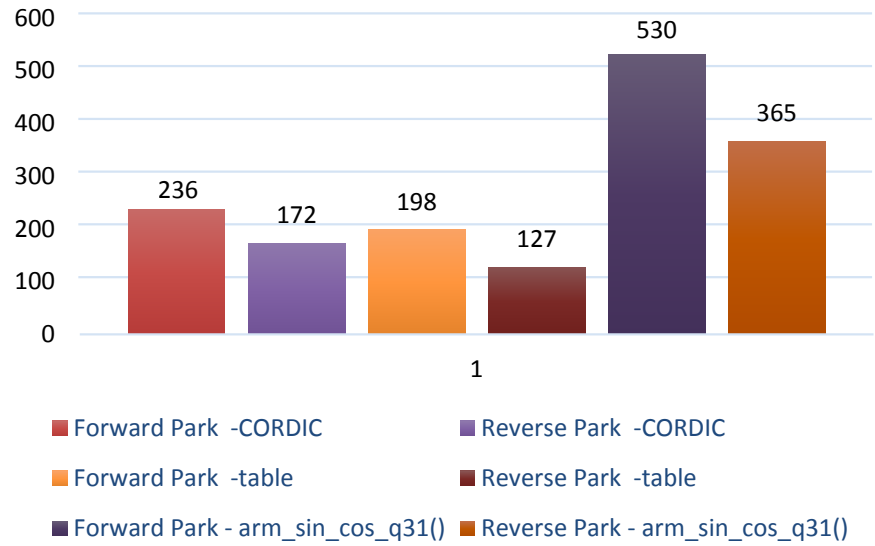
Table 1. Execution time vs software for reverse and forward park

Function used to calculate sine and cosine	Average CPU cycles (optimized for speed)
Forward Park - CORDIC	236
Reverse Park - CORDIC	172
Forward Park - hSin_Cos_Table	198
Reverse Park - hSin_Cos_Table	127
Forward Park - arm_sin_cos_q31()	530
Reverse Park - arm_sin_cos_q31()	365

These figures demonstrate that the performance in terms of clock cycles of park transforms with CORDIC and with *hSin_Cos_Table* usage are comparable, with a slight advantage of the table implementation, and both performance are clearly better than *arm_sin_cos_q31()* that needs about the double of the clock cycles. The very important advantage of the CORDIC respect to *hSin_Cos_Table* implementation is that it is a coprocessor very immediate to configure with few commands lines. The wanted precision of function calculation can be easily modified with no need to change every time a table that needs also memory space to be stored. Moreover, the CORDIC usage also disengages the CPU that can be used for other tasks. Finally, it can be used to calculate many other mathematical functions as well as trigonometric.

The measurements of Table 1 are summarized in Figure 6.

Figure 6. Summary of park transforms performance



Revision history

Table 2. **Document revision history**

Date	Revision	Changes
08-Feb-2023	1	Initial release.

Contents