How to use the CORDIC to perform mathematical functions on STM32 MCUs

Introduction

This document applies to the STM32 microcontrollers featuring the CORDIC accelerator unit, as listed in Table 1. Applicable products.

The CORDIC is a hardware accelerator designed to speed up the calculation of certain mathematical functions. These are in particular trigonometric and hyperbolic functions, compared to a software implementation.

The accelerator is particularly useful in motor control and related applications, where algorithms require frequent and rapid conversions between rectangular (x, y) and angular (amplitude, phase) coordinates.

This application note describes how the CORDIC accelerator works on STM32 microcontrollers. The document also describes the capabilities and limitations of the CORDIC accelerator. It also evaluates the speed of execution for certain calculations compared with an equivalent software implementation.

The example code given in this application note is included in the STM32CubeG4 MCU package available on www.st.com. The examples run on the NUCLEO-G474RE board.

Table 1. Applicable products

Type	Products
Microcontrollers	STM32G4 series STM32G5x3 and STM32G5x4 product lines STM32U5 series STM32H7P4/P5 product line STM32H7R3/7S3 product line STM32H7R7/7S7 product line STM32H563/573 product line STM32H723/733 product line STM32H725/735 product line STM32H730 value line STM32H5E4/F4 and STM32H5E5/F5 product lines STM32C5 series STM32H54x/5x product line

1 General information

The STM32CubeG4 MCU package runs on STM32G4 series microcontrollers, based on Arm® Cortex®-M4 processors.

Note: Arm and Cortex are registered trademarks of Arm Limited (or its subsidiaries or affiliates) in the US and/or elsewhere.

The Arm word and logo are trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All rights reserved.

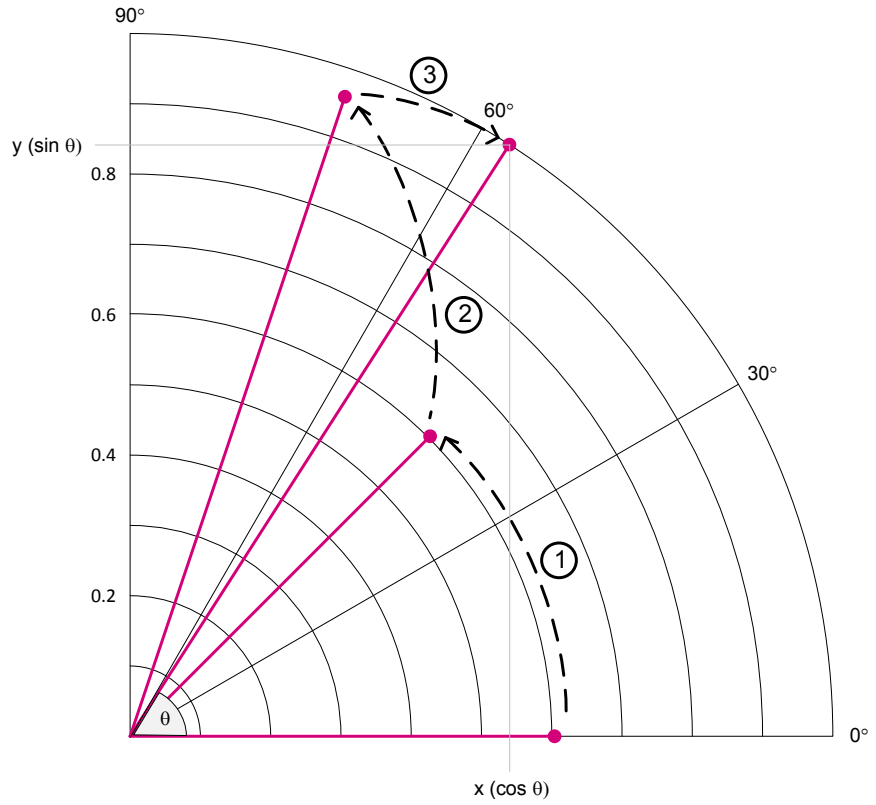
2 CORDIC introduction

The CORDIC (coordinate rotation digital computer) is a low-cost successive approximation algorithm for evaluating trigonometric and hyperbolic functions.

Originally presented by Jack Volder in 1959, it was widely used in early calculators.

In trigonometric (circular) mode, the sine and cosine of an angle θ are determined by rotating the vector [0.61, 0] through decreasing angles $\tan^{-1}(2^{-n})$ (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 Figure 1 for an angle of 60 °C. The vector undergoes a scaling by 1/0.61 (~1.65) over the course of the calculation.

Figure 1. CORDIC circular mode operation



Inversely, the angle of a vector [x, y] is determined by rotating 0.61[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 arctangent (y/x).

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

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 $\sqrt{(\cosh^2 t - \sinh^2 t)}$ as a by-product. 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:

$$\tanh x = \frac{\sinh x}{\cosh x}$$

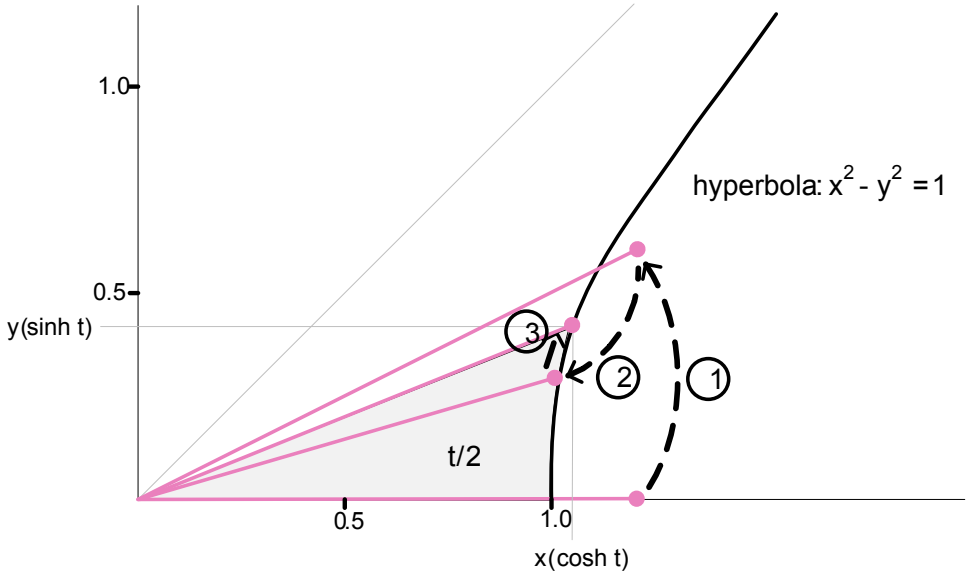
$$e^x = \sinh x + \cosh x$$

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

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

The CORDIC algorithm lends itself to hardware implementation since there are no multiplications involved (the fixed scaling factor 0.61 is preloaded). Only add and shift operations are performed. This also means that it is ideally suited to integer arithmetic.

Figure 2. CORDIC hyperbolic mode operation



2.1 Limitations

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.

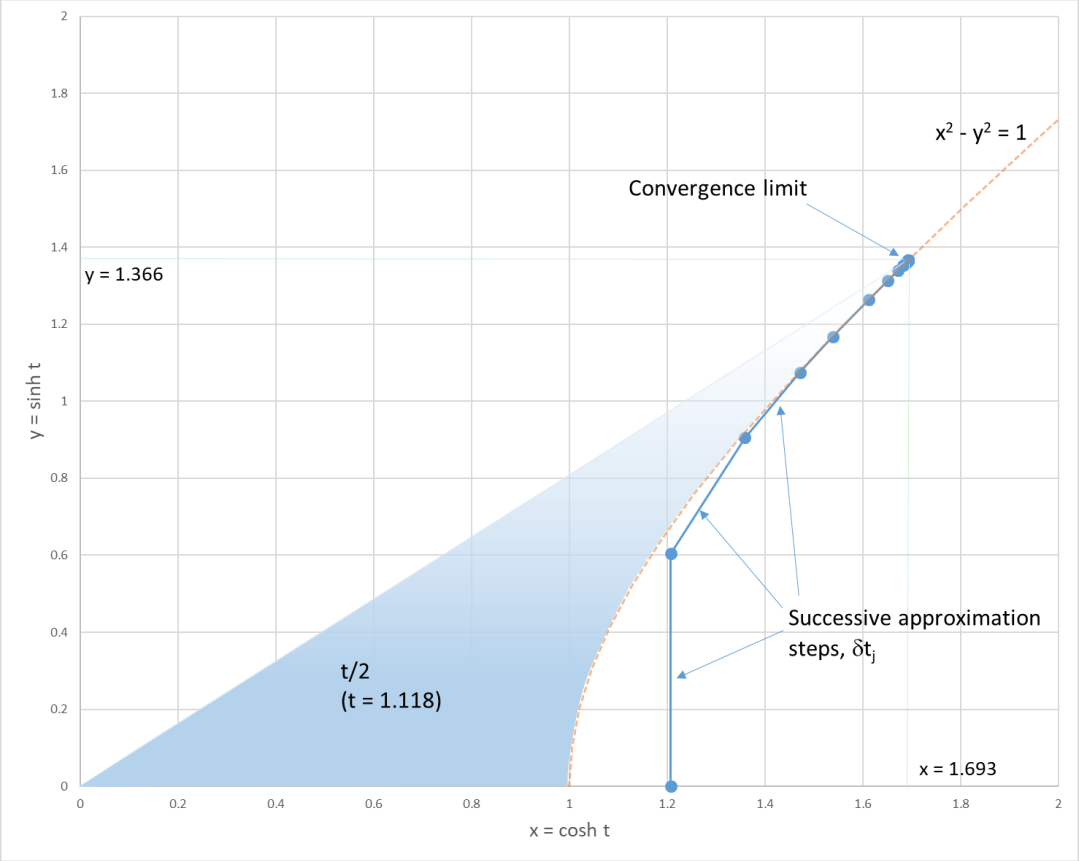
In hyperbolic mode, x = cosh t being defined only for x ≥ 1, all inputs and outputs are divided by 2 to remain in the fixed-point numeric range. Hence, the CORDIC without additional scaling supports values of x = cosh(t) in the range 1 to 2 (that is, 0.5 to 1 after division by 2), which corresponds to the hyperbolic angle magnitude t in the range 0 to 1.317 ($\cosh^{-1} 2$).

It is possible to increase the range by additional scaling. However, the CORDIC algorithm stops converging for |t| > 1.118. This is because the successive approximation step size, $|\delta t_j|$ is equal to the magnitude of the hyperbolic angle $|\tanh^{-1} 2^{-j}|$, where j is the iteration number. So, as j increases, the step size decreases. It so happens that as j tends to infinity, the sum of $|\delta t_j|$ tends to 1.118:

$$\sum_{j=1}^{\infty} \tanh^{-1} 2^{-j} = 1.118$$

This is illustrated in Figure 3.

Figure 3. Hyperbolic convergence limit



So, 1.118 is the effective limit for the hyperbolic sine and cosine functions' input magnitude. If the value of t is constrained to be smaller than this limit, then the CORDIC is used directly. Otherwise, the magnitude of t must be tested. If it is above the limit, an alternative software algorithm must be used, such as the math.h library functions coshf() and sinhf().

The magnitude of the atanh function output is similarly limited to 1.118. Hence, the input magnitude is limited to tanh 1.118 = 0.806.

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 atanh 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 $2^{-n} \cdot (x+1) < 1$, to avoid overflowing the fixed point numerical format.

In a similar way, the convergence limit for the sqrt function is calculated. Since the limit for the atanh function is given by:

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

and the inputs to the atanh 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.

2.2 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 STM32 microcontrollers supports 16-bit and 32-bit input and output data, and has an internal precision of 24 bits.

Figure 4 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 14th 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 5. 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 4. CORDIC convergence (circular mode)

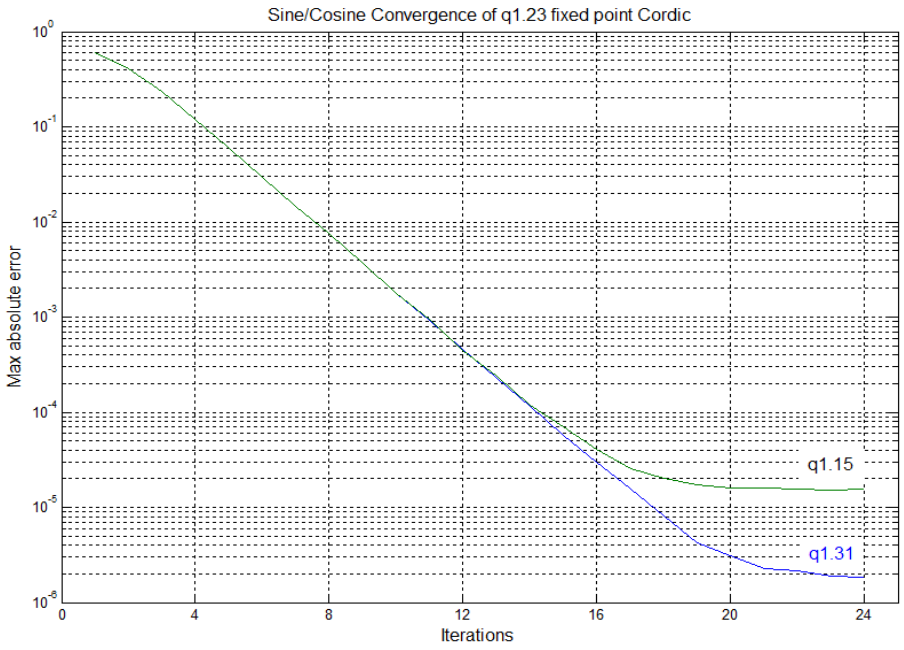
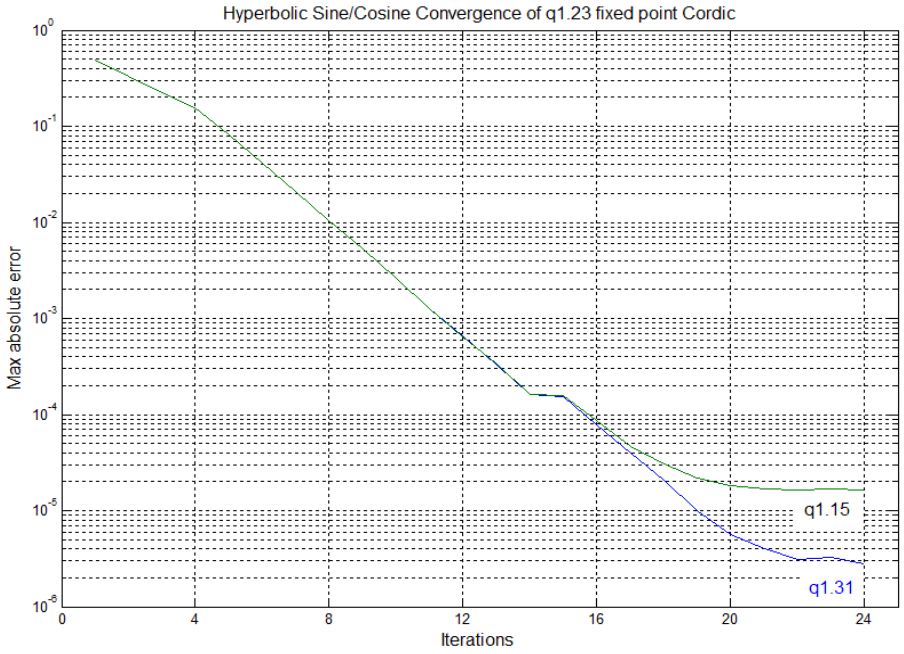


Figure 5. CORDIC convergence (hyperbolic mode)



3 Code examples and performance

3.1 Acceleration example: one-off calculation in software

The CORDIC unit is designed primarily to accelerate the evaluation of mathematical expressions compared to an equivalent function from a software library such as math.h.

An example program is in the STM32CubeG4 MCU Package, under \Projects\NUCLEO-G474RE\Examples_LL\CORDIC\CORDIC_CosSin. This example performs a polar to rectangular conversion using the cosine function. The example code uses the LL (low level) driver, which has less overhead than the HAL driver. Executing the function comprises three steps:

1. Configure the CORDIC:

```
LL_CORDIC_Config(CORDIC,
LL_CORDIC_FUNCTION_COSINE,   /* cosine function */
LL_CORDIC_PRECISION_6CYCLES, /* max precision for q1.31 cosine */
LL_CORDIC_SCALE_0,         /* no scale */
LL_CORDIC_NBWRITE_1,         /* One input data: angle. Second input data (modulus) is 1 af
ter cordic reset */
LL_CORDIC_NBREAD_2,          /* Two output data: cosine, then sine */
LL_CORDIC_INSIZE_32BITS,     /* q1.31 format for input data */
LL_CORDIC_OUTSIZE_32BITS);   /* q1.31 format for output data */
```

   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):

```
/* Write angle */
LL_CORDIC_WriteData(CORDIC, ANGLE_CORDIC);
```

   In this case, there is only one argument, the angle (defined as a constant value π/8). The other argument is the default modulus of 1, so does not need to be written.
   As soon as the expected number of arguments is written, the calculation starts.

3. Read the result(s):

```
/* Read cosine */
cosOutput = (int32_t)LL_CORDIC_ReadData(CORDIC);
/* Read sine */
sinOutput = (int32_t)LL_CORDIC_ReadData(CORDIC);
```

   There are two results expected. Since the output format is 32-bit, two reads are necessary.
   Note that no polling of the output ready flag is required, the first read only completes when the result is available.

3.1.1 Measuring the execution time

The user can measure the number of processor cycles required to execute steps 2 and 3 above by comparing the “SysTick” counter value before and after. The SysTick counter decrements each processor clock cycle. In the STM32Cube the value of the timer is read using the pointer SysTick->VAL. So to measure the time, the SysTick counter is read immediately before writing the argument to the CORDIC and again immediately after reading the results:

```
/* Read systick counter */
start_ticks = SysTick->VAL;
/* Write angle */
LL_CORDIC_WriteData(CORDIC, ANGLE_CORDIC);
/* Read cosine */
cosOutput = (int32_t)LL_CORDIC_ReadData(CORDIC);
/* Read sine */
sinOutput = (int32_t)LL_CORDIC_ReadData(CORDIC);
/* Read systick counter */
stop_ticks = SysTick->VAL;
/* Calculate number of cycles elapsed */
elapsed_ticks = start_ticks-stop_ticks;
```

The number of cycles taken to calculate the sine and cosine is the difference between the two count values.

Note: The SysTick counter reloads automatically when it reaches zero. If this happens, the elapsed ticks value is false. For short measurement periods however this is not likely to occur.

In reality an additional cycle is incurred to store the starting value in memory, but for comparison purposes this is ignored.

3.1.2 Performance comparison

The following performance figures (Table 2) are obtained using the NUCLEO-G474RE, with the Cortex®-M4F running at 170 MHz from flash (with the ART cache enabled). The CORDIC executes at the same clock frequency. The code is compiled using the IAR Embedded Workbench IDE for Arm, v8.30.1. The optimization is set to high for speed and for size.

Table 2. Execution time versus software for polar to rectangular conversion

Function used to calculate sine and cosine	CPU cycles (optimized for speed)	CPU cycles (optimized for size)
CORDIC in zero overhead mode (32-bit integer)	29	29
CORDIC in zero overhead mode with conversion from single precision floating-point to 32-bit integer and back	79	82
math.h single precision floating-point functions: float sinf(float), float cosf(float)	416	416
arm_math.h DSP library 32-bit fixed point function: arm_sin_cos_q31(q31_t, q31_t*, q31_t*)	742	733
arm_math.h DSP library 32-bit floating-point function: arm_sin_cos_f32(float32_t, float32_t*, float32_t*)	405	413
math.h double precision floating-point functions: double sin(double), double cos(double)	4036	4053

These figures clearly demonstrate the advantage of the CORDIC for polar to rectangular conversions. Compared to the Arm DSP library function in q31 fixed point, the CORDIC uses less than 4% of the CPU cycles. Even together with the conversion from floating-point to integer and back, the CORDIC uses less than 20% of the CPU cycles used by the single-precision Arm DSP library function or math.h functions. The double precision measurements are for reference. None of the other options, including the CORDIC, achieves double precision accuracy but the cost in cycles on a processor such as the Cortex®-M4 is much lower.

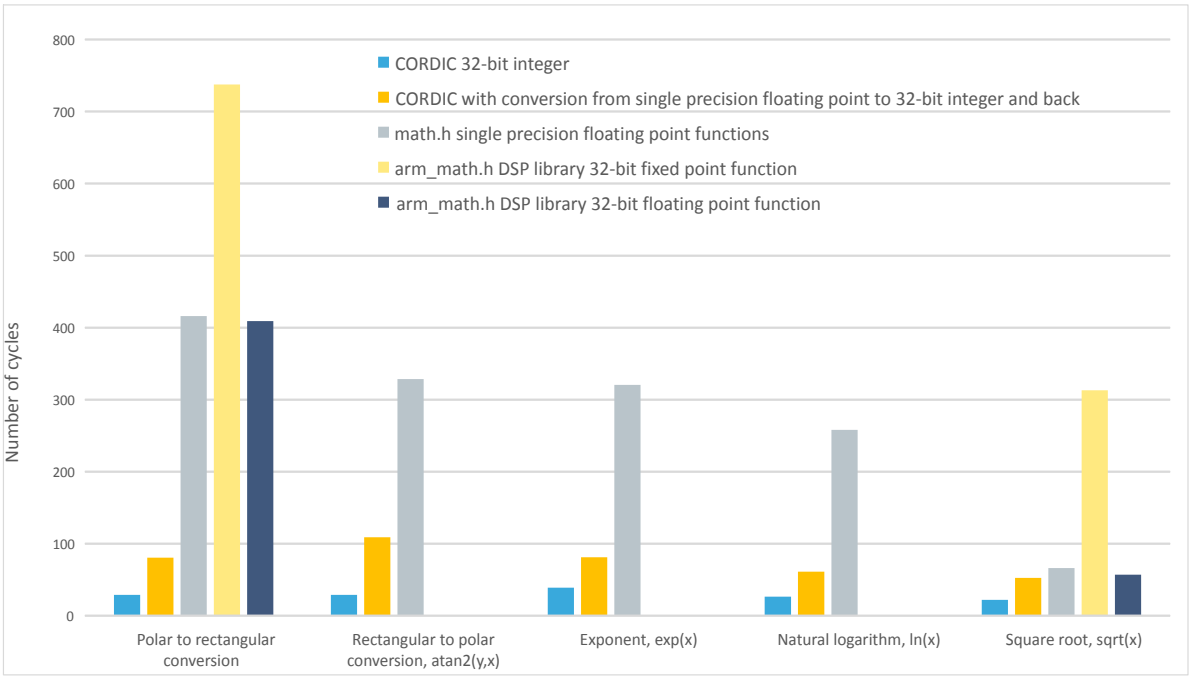
Table 3 shows the comparison for some of the other functions supported by the CORDIC.

Table 3. Execution time versus software for other functions

Function	CPU cycles (optimized for speed)	CPU cycles (optimized for size)
Rectangular to polar conversion, atan2(y,x)		
CORDIC in zero overhead mode	33	25
CORDIC in zero overhead mode with conversion from single precision floating-point to 32-bit integer and back	107	111
math.h single precision floating-point function: float atan2f(float, float)	332	325
math.h double precision floating-point function: double atan2(double, double)	4194	4209
Exponent, exp(x)		
CORDIC in zero overhead mode (sinh + cosh)	39	39
CORDIC in zero overhead mode with conversion from single precision floating-point to 32-bit integer and back	81	81

Function	CPU cycles (optimized for speed)	CPU cycles (optimized for size)
math.h single precision floating-point function: float expf(float)	319	322
math.h double precision floating-point function: double exp(double)	3604	3608
Natural logarithm, ln(x)		
CORDIC in zero overhead mode	27	26
CORDIC in zero overhead mode with conversion from single precision floating-point to 32-bit integer and back	61	61
math.h single precision floating-point function: float lnf(float)	260	256
math.h double precision floating-point function: double log(double)	2744	2740
Square root, sqrt(x)		
CORDIC in zero overhead mode	23	21
CORDIC in zero overhead mode with conversion from single precision floating-point to 32-bit integer and back	53	52
math.h single precision floating-point function: float sqrtf(float)	58	74
arm_math.h DSP library 32-bit fixed point function: arm_sqrt_q31(q31_t, q31_t*)	313	313
arm_math.h DSP library 32-bit floating-point function: arm_sqrt_f32(float32_t, float32_t*)	57	57
math.h double precision floating-point function: double sqrt(double)	814	814

Figure 6. Summary of CORDIC performance versus software



The above measurements (without the double precision results) are summarized in Figure 6. The CORDIC gives a significant speed-up for all supported functions when fixed point or integer arithmetic is used. For single precision floating-point the acceleration is less, due to the vector floating-point unit present on the Cortex®-M4. Nevertheless, there is still a significant gain, except for the square root function, where the cost of converting between floating-point and integer more or less cancels the advantage of the CORDIC.

Note that the CORDIC performs the calculations in six clock cycles (3 for the square root). The remaining cycles are used up accessing the memory and the CORDIC registers. This can vary depending on the compiler optimization settings and the surrounding code. In this example the output variables are global, so the results are systematically written to memory after they are read. By declaring them locally, they are kept in registers if they are subsequently used. This saves a considerable number of cycles, for example, the figure in Table 2 for polar to rectangular conversion using the CORDIC in integer mode drops from 29 to 17 cycles.

3.2 Acceleration example: multiple data using DMA

Another use of the CORDIC unit is to perform repetitive calculations on multiple data. An example is in the STM32CubeG4 MCU package, under \Projects\NUCLEO-G474RE\Examples\CORDIC\CORDIC_Sin_DMA. This example converts a vector of angles into a sine wave. The input and output data are stored in memory, so the DMA controller can handle all the transfers between memory and the CORDIC. The processor thus plays no part in the operation, apart from initializing the CORDIC unit.

This example uses the HAL driver to initialize the CORDIC and the DMA. This has no effect on the performance, since the software does not intervene in the actual calculation and data transfer to and from memory.

As before the CORDIC needs to be configured, this time for the sine function. Only one argument and one result (both 32-bit integers) are required. The precision is again set to six cycles:

```
/*## Configure the CORDIC peripheral ######################################*/
sCordicConfig.Function = CORDIC_FUNCTION_SINE; /* sine function */sCordicConfig.Precision = C
ORDIC_PRECISION_6CYCLES; /* max precision for q1.31 sine */
sCordicConfig.Scale = CORDIC_SCALE_0; /* no scale */
sCordicConfig.NbWrite = CORDIC_NBWRITE_1; /* One input data: angle.
Second input data (modulus) is 1 after cordic reset */
sCordicConfig.NbRead = CORDIC_NBREAD_1; /* One output data: sine*/
sCordicConfig.InSize = CORDIC_INSIZE_32BITS; /* q1.31 format for input data */
sCordicConfig.OutSize = CORDIC_OUTSIZE_32BITS; /* q1.31 format for output data */if (HAL_CORD
IC_Configure(&hcordic, &sCordicConfig) != HAL_OK)
   {
/* Configuration Error */
Error_Handler();
   }
```

A DMA channel is configured to transfer data from memory to the CORDIC:

```
/* CORDIC_WRITE Init */
hdma_cordic_write.Instance = DMA1_Channel1;
hdma_cordic_write.Init.Request = DMA_REQUEST_CORDIC_WRITE;
hdma_cordic_write.Init.Direction = DMA_MEMORY_TO_PERIPH;
hdma_cordic_write.Init.PeriphInc = DMA_PINC_DISABLE;
hdma_cordic_write.Init.MemInc = DMA_MINC_ENABLE;
hdma_cordic_write.Init.PeriphDataAlignment = DMA_PDATAALIGN_WORD;
hdma_cordic_write.Init.MemDataAlignment = DMA_MDATAALIGN_WORD;
hdma_cordic_write.Init.Mode = DMA_NORMAL;
hdma_cordic_write.Init.Priority = DMA_PRIORITY_LOW;
if (HAL_DMA_Init(&hdma_cordic_write) != HAL_OK)
Error_Handler();
```

A second channel is configured to transfer data from the CORDIC to memory:

```
/* CORDIC_READ Init */
hdma_cordic_read.Instance = DMA1_Channel2;
hdma_cordic_read.Init.Request = DMA_REQUEST_CORDIC_READ;
hdma_cordic_read.Init.Direction = DMA_PERIPH_TO_MEMORY;
hdma_cordic_read.Init.PeriphInc = DMA_PINC_DISABLE;
hdma_cordic_read.Init.MemInc = DMA_MINC_ENABLE;
hdma_cordic_read.Init.PeriphDataAlignment = DMA_PDATAALIGN_WORD;
hdma_cordic_read.Init.MemDataAlignment = DMA_MDATAALIGN_WORD;
hdma_cordic_read.Init.Mode = DMA_NORMAL;
hdma_cordic_read.Init.Priority = DMA_PRIORITY_LOW;
```

```
if (HAL_DMA_Init(&hdma_cordic_read) != HAL_OK)
Error_Handler();
```

The CORDIC is started with the HAL_CORDIC_Start_DMA() function:

```
/*## Start calculation of sines in DMA mode ####################*/
if (HAL_CORDIC_Calculate_DMA(&hcordic, aAngles, aCalculatedSin,
ARRAY_SIZE, CORDIC_DMA_DIR_IN_OUT) != HAL_OK)
Error_Handler();
```

The CORDIC generates a DMA request on its write channel (DMA channel 1), upon which the DMA controller fetches a value from the table of angles in memory (int32_t aAngles[ARRAY_SIZE]) and writes it into the CORDIC WDATA register. This being the first value to be transferred, the CORDIC is idle, so as soon as the write takes place, the calculation starts. This frees up the input register, so a new DMA write request is raised, triggering the transfer of the second angle value.

When the CORDIC finishes calculating the sine of the first angle, it places the result in the RDATA register, and generates a DMA request on the read channel (DMA channel 2). The DMA controller reads the result and writes it to the result table, int32_t aCalculatedSin[ARRAY_SIZE]. The act of reading the RDATA register triggers the start of the second calculation, which in turn generates a new request on the write channel to fetch the third angle. This continues until all 64 entries in the table have been processed.

Provided the DMA can keep it supplied with data, the CORDIC is only idle while waiting for the DMA to respond to the read request. Thus, the CORDIC operates at close to maximum speed.

During this time the processor is idle, and can perform other tasks. In the example, it stays in a loop while, waiting for the DMA transfer complete interrupt, to release the CORDIC process. The number of cycles taken to process all the angles can be measured by inserting the SysTick timer read in the code, as in the previous example but at the beginning and end of the loop. To be more accurate, the second SysTick read must be performed on entry into the DMA transfer complete interrupt handler.

In this example, a total of 993 cycles are required to process 64 values. This corresponds to an average of 15.5 cycles per value. Since the CORDIC can process a value in six clock cycles, it is clear that the DMA cannot provide data at the maximum rate. Nevertheless, the average throughput is slightly better than that obtained in the previous example one calculation at a time.

3.3 Optimal performance

To achieve the maximum theoretical performance from the CORDIC, the software must perform the transfer of data from the memory to the CORDIC and back, instead of the DMA. The following code replaces the call to HAL_CORDIC_Calculate_DMA() in the previous example:

```
/* Write first angle to cordic */
  CORDIC->WDATA = aAngles[0];
/* Write remaining angles and read sine results */
  for(uint32_t i = 0; i < ARRAY_SIZE; i++)
  {
    CORDIC->WDATA = aAngles[i];
   *pCalculatedSin++ = CORDIC->RDATA;
  }
  /* Read last result */
  *pCalculatedSin = CORDIC->RDATA;
```

where *pCalculatedSin is a pointer to the start of aCalculatedSin[]:

```
/* Array of calculated sines in Q1.31 format */
static int32_t aCalculatedSin[ARRAY_SIZE];
/* Pointer to start of array */
int32_t *pCalculatedSin = aCalculatedSin;
```

Note that the first two arguments are written before the first result is read, to load the pipeline. This way the CORDIC is never idle waiting for a new argument.

With the above code, a buffer of ARRAY_SIZE = 3024 values is processed in 24261 cycles, corresponding to an average of eight cycles per value. This consists of 6 cycles of the CORDIC engine (that is, 24 iterations at four iterations per cycle) plus one cycle to transfer the WDATA contents into the CORDIC and another cycle to transfer the result into the RDATA register. Hence, the CORDIC is operating at maximum throughput.

4 Conclusion

The CORDIC offers a significant speed-up of trigonometric and hyperbolic functions, as well as natural log and exponent, compared to software implementations. The biggest gains are obtained when working in the native fixed point format. This avoids the cost of conversion from and to floating-point. Nevertheless, apart from the square root function, there is still a speed improvement of x3 to x5 when working in single precision floating-point. This acceleration helps reduce the loop delay for algorithms such as the park and the inverse park transforms used in motor control applications.

The unit is also used to perform repeated calculations on an array of values. This is in particular required for continuously variable frequency tone generation or waveform synthesis. If a maximum speed is required, the pipeline mode allows the unit to operate flat out when serviced by the processor. This offers significant speed-up with respect to doing the calculations in the software. If on the other hand the goal is to offload such tasks from the processor, and free up the processor time for other tasks, then the DMA controller is used to service the CORDIC, with little loss of performance.

Writing software for the CORDIC is simple, especially using the functions and macros provided by the HAL and LL drivers in the STM32Cube. However, care must be taken not to exceed the numerical limits of the unit, which may require additional software checks if not specified by design - not tested in production.

Revision history

Table 4. **Document revision history**

Date	Revision	Changes
23-May-2019	1	Initial release.
11-Mar-2021	2	Updated: • Section 2: CORDIC introduction. • Table 3. Execution time versus software for other functions. • Section 3.2: Acceleration example: multiple data using DMA.
6-Jul-2022	3	Updated Section Introduction
28-Sep-2022	4	Updated /* Write first angle to cordic */ CORDIC->WDATA = aAngles[0]; /* Write remaining angles and read sine results */ for(uint32_t i = 0; i < ARRAY_SIZE; i++) { CORDIC->WDATA = aAngles[i]; *pCalculatedSin++ = CORDIC->RDATA; } /* Read last result */ *pCalculatedSin = CORDIC->RDATA Replaced "BLOCKSIZE" with "ARRAY_SIZE"
27-Feb-2023	5	Changed title of the document Added STM32H563/573 product line in Table 1. Applicable products
07-Mar-2024	6	Added STM32H7Rx/Sx products in Table 1. Applicable products
27-Jan-2026	7	Added STM32H5E4/F4 and STM32H5E5/F5 product lines and STM32C5 series in Table 1. Applicable products
09-Jun-2026	8	Added STM32H54x/5x product line in Table 1. Applicable products
21-Oct-2026	9	Added the STM32G5x3, STM32G5x4, and STM32H7P4/P5 products. Table 1. Applicable products: updated.

Contents

1 General information 2
2 CORDIC introduction 3
2.1 Limitations 4
2.2 Convergence rate and precision 6
3 Code examples and performance 8
3.1 Acceleration example: one-off calculation in software 8
3.1.1 Measuring the execution time 8
3.1.2 Performance comparison 9
3.2 Acceleration example: multiple data using DMA 11
3.3 Optimal performance 12
4 Conclusion 13
Revision history 14
List of tables 16
List of figures 17

List of tables

Table 1. Applicable products . . . 1
Table 2. Execution time versus software for polar to rectangular conversion . . . 9
Table 3. Execution time versus software for other functions . . . 9
Table 4. Document revision history . . . 14

List of figures

Figure 1. CORDIC circular mode operation . . . 3
Figure 2. CORDIC hyperbolic mode operation . . . 4
Figure 3. Hyperbolic convergence limit . . . 5
Figure 4. CORDIC convergence (circular mode) . . . 6
Figure 5. CORDIC convergence (hyperbolic mode) . . . 7
Figure 6. Summary of CORDIC performance versus software . . . 10

IMPORTANT NOTICE – READ CAREFULLY

STMicroelectronics NV and its subsidiaries (“ST”) reserve the right to make changes, corrections, enhancements, modifications, and improvements to ST products and/or to this document at any time without notice.

In the event of any conflict between the provisions of this document and the provisions of any contractual arrangement in force between the purchasers and ST, the provisions of such contractual arrangement shall prevail.

The purchasers should obtain the latest relevant information on ST products before placing orders. ST products are sold pursuant to ST’s terms and conditions of sale in place at the time of order acknowledgment.

The purchasers are solely responsible for the choice, selection, and use of ST products and ST assumes no liability for application assistance or the design of the purchasers’ products.

No license, express or implied, to any intellectual property right is granted by ST herein.

Resale of ST products with provisions different from the information set forth herein shall void any warranty granted by ST for such product.

If the purchasers identify an ST product that meets their functional and performance requirements but that is not designated for the purchasers' market segment, the purchasers shall contact ST for more information.

ST and the ST logo are trademarks of ST. For additional information about ST trademarks, refer to www.st.com/trademarks. All other product or service names are the property of their respective owners.

Information in this document supersedes and replaces information previously supplied in any prior versions of this document.

© 2026 STMicroelectronics – All rights reserved