DT0089
Design tip

The Goertzel algorithm to compute individual terms of the discrete Fourier transform (DFT)

By Andrea Vitali

Main components	
STM32L031C4/E4/F4/G4/K4 STM32L031C6/E6/F6/G6/K6	Access line ultra-low-power 32-bit MCU Arm®-based Cortex®-M0+, up to 32 KB Flash, 8 Kbytes SRAM, 1 KB EEPROM, ADC
STM32F031C4/F4/G4/K4 STM32F031C6/E6/F6/G6/K6	Arm®-based 32-bit MCU with up to 32 Kbytes Flash, 9 timers, ADC and communication interfaces, 2.0 - 3.6 V
STM32L476xx/486xx	Ultra-low-power Arm® Cortex®-M4 32-bit MCU+FPU, 100 DMIPS, up to 1 MB Flash, 128 Kbytes SRAM, USB OTG FS, LCD, analog, audio
STM32F411xx	Arm® Cortex®-M4 32-bit MCU+FPU, 125 DMIPS, 512 Kbytes Flash,128 Kbytes RAM, USB OTG FS, 11 TIMs, 1 ADC, 13 comm. interfaces

Purpose and benefits

This design tip explains how to compute individual terms of the discrete Fourier transform using the Goertzel algorithm.

- The Goertzel algorithm is derived and implemented as an iteration loop and as an IIR filter. A reference implementation based on the FFT is also included for verification.
- The Goertzel algorithm is generalized to the case of non-integer frequency index.
- An application example is included: dual-tone multi-frequency (DTMF) decoding. A MATLAB® script is used to generate a test signal and perform floating point decoding. A fixed-point C implementation is also provided.

Description

In the Goertzel algorithm, a set of N signal samples x() is transformed into a set of N frequency coefficients y() using the discrete Fourier transform (DFT): y(k) = sum of W(nk) x(n), where n and k go from 0 to N-1 and the twiddle factor W(t) is defined as exp(-j 2pi/N t).

The twiddle factor has the property that W(-Nk)=1 for all k, therefore the k-th coefficient y(k) is y(k) = sum of W(k(n-N)) x(n). The summation can be unfolded and manipulated to make it recursive:

$y(k) = W(-k)\ x(N-1) + W(-2k)\ x(N-2) + W(-3k)\ x(N-3) + \ldots + W(-k(N-1))\ x(1) + W(-k\ N\ )\ x(0)$

$y(k) = W(-k)\ [\ x(N-1) + W(\ -k)\ x(N-2) + W(-2k)\ x(N-3) + \ldots + W(-k(N-2))\ x(1) + W(-k(N-1))\ x(0)\ ]$

$y(k) = W(-k)\ [\ x(N-1) + W(\ -k)\ [\ x(N-2) + W(\ -k)\ x(N-3) + \ldots + W(-k(N-3))\ x(1) + W(-k(N-2))\ x(0)\ ]\ ]$

$y(k) = W(-k)\ [\ x(N-1) + W(\ -k)\ [\ x(N-2) + W(\ -k)\ [\ x(N-3) + \ldots + W(-k(N-4))\ x(1) + W(-k(N-3))\ x(0)\ ]\ ]\ ]$

It is evident that the frequency coefficient y(k) can be computed in N steps using the following recursive formula: $y_n(k) = W(-k)\ [\ x(n) + y_{n-1}(k)\ ]$, n goes from 0 to N-1, $y_0(k)=0$.

The recursive formula can be seen as a second order IIR filter because it is a weighted of the input sample x and the previous output y. Of course, only the final output $y_{N-1}(k)$ is of interest, intermediate outputs are not needed. It is possible to manipulate the coefficients of the filter so that the computation is done using real numbers, except for the last step where the final complex output $y_{N-1}(k)$ is computed.

The recursive formula in z-domain is $Y(z) = W(-k)\ [\ X(z) + Y(z)\ z^{-1})$. Therefore the filter transfer function is $H(z) = Y(z)/X(x) = W(-k) / (1 – W(-k)\ z^{-1})$. There is no change if the numerator and the denominator are multiplied by the same quantity $(1 – W(+k)\ z^{-1})$.

$H(z) = W(-k)\ (1-W(+k)\ z^{-1}) / [\ (1 – W(-k)\ z^{-1})\ (1-W(+k)\ z^{-1})\ ]$

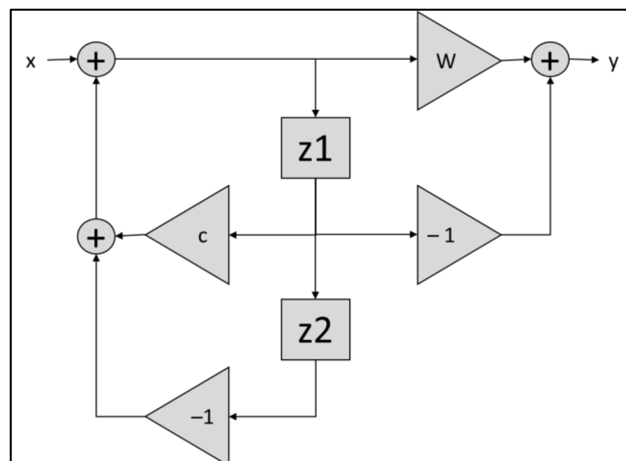
$H(z) = [\ W(-k) – W(-k)W(+k)\ z^{-1}\ ] / [\ 1 – (W(-k)+W(+k))\ z^{-1} + W(-k)W(+k)\ z^{-2})\ ]$

Remember that W(-k) = exp(j 2pi/N k) and W(+k) = exp(-j 2pi/N k), therefore W(-k)W(+k) = exp(0) = 1 and (W(-k)+W(+k)) = 2 cos(2pi/N k).

$H(z) = [\ W(-k) –z^{-1}\ ] / [\ 1 –2\cos(2pi/N\ k)\ z^{-1} + z^{-2})\ ]$

If the filter is implemented as Direct II form (Figure 1), the status registers of the filter are real numbers, and multiplications and additions in the feedback loop are real. One complex multiplication is needed at the last step to compute the real and the imaginary part of the output y(k).

Figure 1. Direct II form for $H(z) = (W – z^{-1}) / (1 –cz^{-1} + z^{-2})$, W=exp(j 2pi/N k), c=2cos(2pi/N k)



Goertzel algorithm implementation

The following function takes a vector x made of N samples, and computes the k-th frequency coefficient (k from 0 to N-1). The real part is returned in I (in-phase) and the imaginary part is returned in Q (quadrature-phase).

```
function [I,Q] = goertzel(x,k,N)
  w = 2*pi*k/N;
  cw = cos(w); c = 2*cw;
  sw = sin(w);
  z1=0; z2=0; % init
  for n = 1 : N
    z0 = x(n) + c*z1 - z2;
    z2 = z1;
    z1 = z0;
  end;
  I = cw*z1 - z2;
  Q = sw*z1;
```

Goertzel as an IIR filter

The following function returns the same results as the implementation shown above. Here the Goertzel algorithm is implemented as an IIR filter.

```
function [I,Q] = goertzelIIR(x,k,N)
  W = exp(1i*2*pi*k/N);
  c = 2*cos(2*pi*k/N);
  b = [W -1 0]; % FIR coefficients
  a = [1 -c 1]; % IIR coefficients
  y = filter(b, a, x);
  I = real(y(end));
  Q = imag(y(end));
```

Goertzel as the k-th coefficient of an N-point FFT

The following function returns the same results as the implementations shown above. Here the Goertzel algorithm is implemented as an FFT whose k-th coefficient is extracted.

```
function [I,Q] = goertzelFFT(x,k,N)
  y = fft(x);
  I = real(y(k+1));
  Q = imag(y(k+1));
```

Generalized Goertzel for non-integer k index

The Goertzel algorithm can be generalized to handle the case where the k index is not an integer. The computation is the same as shown before. A correction step is added at the end to adjust the phase of the complex output. Note that the magnitude of the output is not changed.

```
function [I,Q] = goertzelgen(x,k,N)
  w = 2*pi*k/N;
  cw = cos(w); c = 2*cw;
  sw = sin(w);
  z1=0; z2=0; % init
  for n = 1 : N
    z0 = x(n) + c*z1 - z2;
```

```
  z2 = z1;
  z1 = z0;
end;
It = cw*z1 - z2;
Qt = sw*z1;

w2 = 2*pi*k;
cw2 = cos(w2);
sw2 = sin(w2);

I =  It*cw2 + Q*sw2;
Q = -It*sw2 + Q*cw2;
```

Goertzel error bounds

The Goertzel algorithm is affected by an error bound which is proportional to N^2. The worst case is for k frequency index near 0 or near N-1. There is also a dependency on the input signal (e.g. a damped sinusoid may cause larger errors). As an example if the machine precision is $\varepsilon \sim 10^{-6}$, for N=64 the error for the worst case can be as large as $\varepsilon 10^4$.

Application example: DTMF dual-tone multi-frequency signaling

The dual-tone multi-frequency signaling (DTMF) has been developed by the Bell System in the United States. It is standardized as ITU-T Rec Q.23. It is also known as “touch-tone” as tones are generated by pushing the buttons in the keypad of telephones.

Freq.	1209 Hz	1336 Hz	1477 Hz	1633 Hz
697 Hz	1	2	3	A
770 Hz	4	5	6	B
852 Hz	7	8	9	C
941 Hz	*	0	#	D

Whenever a button is pressed, two tones are generated with the frequency indicated in the corresponding row and column. It would be inefficient to compute a full FFT to detect the presence of these tones. On the opposite, the Goertzel algorithm can be used to efficiently compute the frequency coefficients of the 8 tones that can be present in the signal.

DTMF encoding and decoding, MATLAB® script

In the following script, the sampling frequency is set to 4 kHz. N is set to 200, so that every 200 samples a new set of frequency coefficients will be available. The minimum symbol duration is then N/Fs = 50 msec.

All possible symbols are encoded. The duration of each symbol is randomly selected in the range N – 1.5N. As the duration is variable, on the output some symbol may appear twice.

The signal is quantized to 8-bit and it is saved to file, so that the C implementation can run on the same input. Because of the quantization noise, on the output some symbol may be missing.

In the code the block of N samples is windowed, using a Hamming window, to enhance the signal-to-noise ratio of the coefficients. For every block, a set of 8 frequency coefficients is computed. Coefficients that are above the threshold are selected. There should be one selected coefficient among the first four (row frequency) and another one among the last

four (column frequency). If there are less or more than two selected coefficients, the symbol will not be decoded and a blank space will be printed instead.

```
% DTMF test, dual-tone multi-frequency
frow = [ 697,  770,  852,  941]; % frequencies for 1st tone
fcol = [1209, 1336, 1477, 1633]; % frequencies for 2nd tone
sym = ['1', '4', '7', '*', '2', '5', '8', '0', ... % symbols
       '3', '6', '9', '#', 'A', 'B', 'C', 'D' ];
symrow = [1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4]; % 1st tone for given symbol
symcol = [1 1 1 1 2 2 2 2 3 3 3 3 4 4 4 4]; % 2nd tone for given symbol
symmtx = [1 5 9 13; 2, 6, 10, 14; 3, 7, 11, 15; 4, 8, 12, 16]; % decoding matrix

Fs = 4000; % Hz, sampling frequency
N = 200; % minimum number of samples per symbol

x = []; % create test signal
symoutref = []; % reference for decoded output
for i=1:length(sym), % test each symbol
  Nsym = N + round(N/2*rand(1)); % samples for current symbol
  t = [0:Nsym-1]/Fs; % time vector
  x1 = sin(2*pi*frow(symrow(i))*t); % first tone
  x2 = sin(2*pi*fcol(symcol(i))*t); % second tone
  x = [x, x1+x2];
  symoutref = [symoutref, sym(i)];
end;
bits = 8;
Q = (max(x)-min(x))/(2^bits); % quantization step for 8 bit signal
x = round(x/Q); % some noise may also be added

fbin=Fs/N; % Goertzel frequency resolution, N must be high enough
k=round([frow fcol]/fbin); % N high enough so that k is different for each tone
if any(diff(k)==0), fprintf('same k index for different tones!\n'); return; end;

% Goertzel-based DTMF decoding
xblocks = floor(length(x)/N);
myspec = []; % zeros(xblocks,length(k));
for i = 1 : xblocks,
  i1 = (i-1)*N+1;
  i2 = i1+N-1;
  xt = x(i1:i2);
  xt = xt.*hamming(N)';
  for j = 1 : length(k),
    [I,Q] = goertzel(xt,k(j),N);
    myspec(i,j) = sqrt(I*I+Q*Q);
  end;
end;

th = max(myspec(:))/2; % threshold
myspecbin = myspec>th; % tone on/off detection
symout = [];
for i = 1 : xblocks,
  i1 = find(myspecbin(i,1:4)>0); if length(i1)~=1, i1=0; end; % 1st tone
  i2 = find(myspecbin(i,5:8)>0); if length(i2)~=1, i2=0; end; % 2nd tone
  if (i1==0) || (i2==0), symdec=' '; % no symbol decoded
  else                   symdec=sym(symmtx(i1,i2)); % symbol decoded
  end;
  symout = [symout, symdec]; % append decoded symbol
end;

% printout and plot
fprintf('reference string: %s\n',symoutref);
fprintf('decoded   string: %s\n',symout);

figure; imagesc(fbin*k/1000,[0:xblocks]*N/Fs*1000,myspec);
axis xy; axis([0 Fs/2/1000 0 length(x)/Fs*1000]); colorbar;
xlabel('Frequency (kHz)'); ylabel('Time (ms)');
title(sprintf('DTMF test, %d-bit Fs=%.1f kHz, %d-point Goertzel',bits,Fs/1000,N));

figure; imagesc(fbin*k/1000,[0:xblocks]*N/Fs*1000,myspecbin);
axis xy; axis([0 Fs/2/1000 0 length(x)/Fs*1000]); colorbar;
xlabel('Frequency (kHz)'); ylabel('Time (ms)');
title(sprintf('DTMF test, %d-bit Fs=%.1f kHz, %d-point Goertzel,
th=%1.f',bits,Fs/1000,N,th));

NFFT=128; NOVL=round(0.9*NFFT); WIN=hamming(NFFT);
figure; spectrogram(x,WIN,NOVL,NFFT,Fs);
title(sprintf('DTMF test, %d-bit Fs=%.1f kHz, %d-points FFT',bits,Fs/1000,NFFT));
```

```
% save quantized test signal to file to test C implementation
h=fopen('in.txt','wt'); fprintf(h,'%d\n',x); fclose(h);
```

The script will plot the spectrogram of the generated signal (Figure 2), as well as the frequency coefficients computed by the Goertzel algorithm (Figure 3). The coefficients above threshold drive the decoding.

Figure 2. Spectrogram of the generated test signal

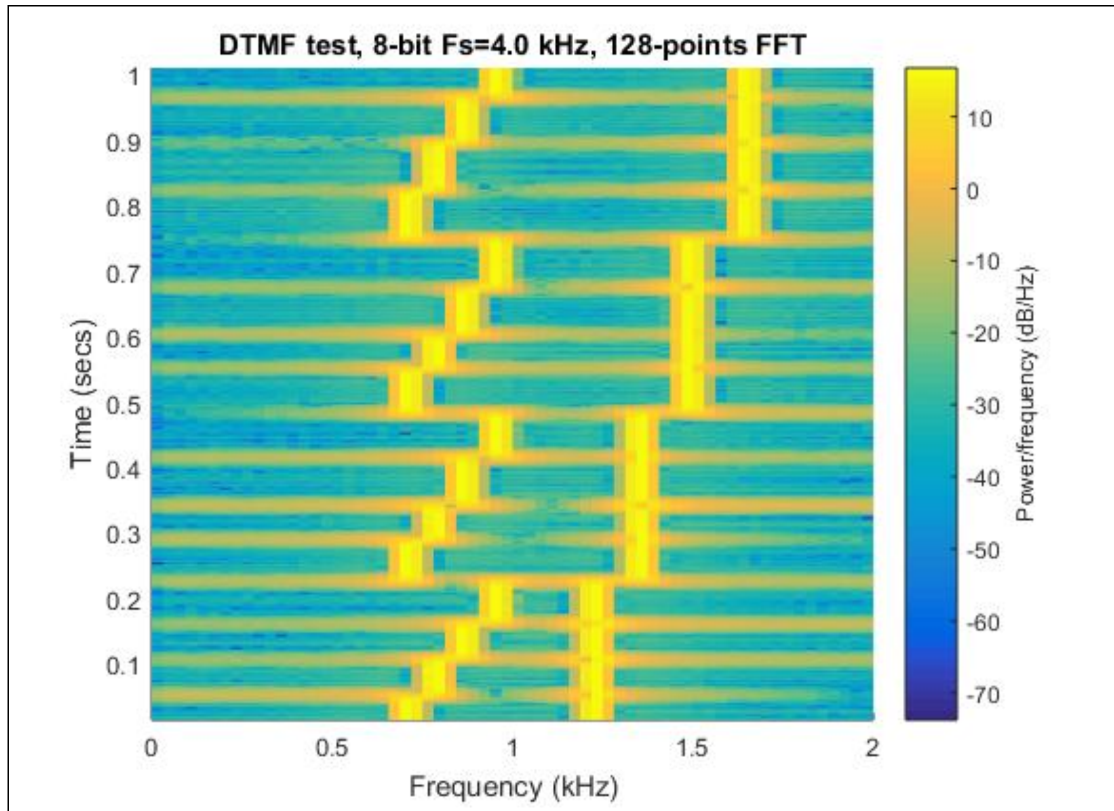
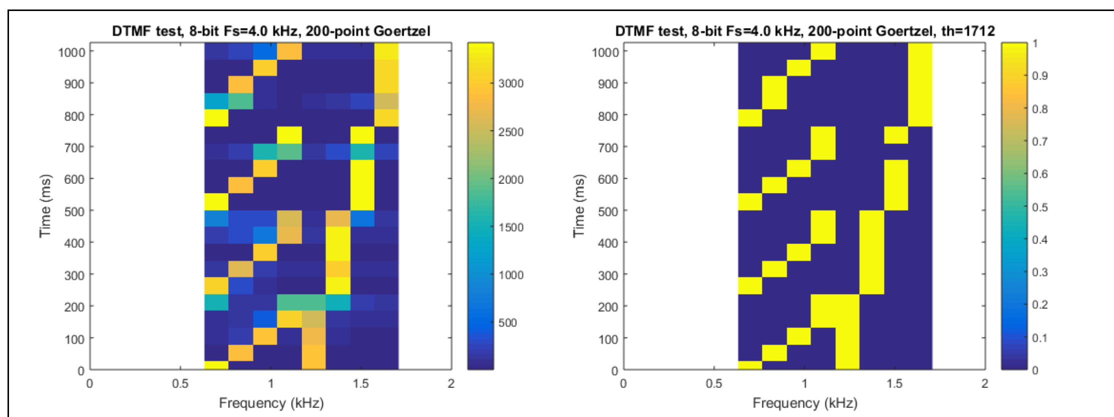


Figure 3. Frequency coefficients computed by the Goertzel algorithm (left), and coefficients above threshold (right)



The MATLAB® output for a typical run is the following:

```
reference string: 147*2580369#ABCD
decoded   string: 147**25800369 #ABBCD
```

DTMF encoding and decoding, fixed-point C

The following C program performs the DTMF decoding using the Goertzel algorithm. The first argument on the command line is the name of the file with the signal samples. This file is generated by the MATLAB® script shown in the previous paragraph. The other arguments on the command line are the sampling frequency in Hz, the block length in samples, the number of bits for the quantization of the Goerzel constants, and the threshold to be used during the decoding process.

The computation of the Goertzel constants is based on sin() and cos(). These functions can be implemented in fixed-point using the CORDIC algorithm.

In the program the squared magnitude is computed, therefore a squared threshold should be used for decoding. The magnitude can also be computed and used: the sqrt() function can be implemented in fixed-point using the CORDIC algorithm.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define MAXN 1000

// DTMF frequencies
int frow[4] = {  697,  770,  852,  941 }; // 1st tone
int fcol[4] = { 1209, 1336, 1477, 1633 }; // 2nd tone

// DTMF symbols
char sym[16] = { '1', '4', '7', '*', '2', '5', '8', '0',
                 '3', '6', '9', '#', 'A', 'B', 'C', 'D' };

// DTMF decoding matrix
int symmtx[4][4] = { { 0, 4,  8, 12 }, { 1, 5,  9, 13 },
                           { 2, 6, 10, 14 }, { 3, 7, 11, 15 } };

int win[MAXN]; // Window

// Goertzel
int c[8], cw[8], sw[8]; // Goertzel constants
int z1[8], z2[8]; // Goertzel status registers
int I[8], Q[8], M2[8]; // Goertzel output: real, imag, squared magnitude

int main(int argc, char *argv[]) {
  FILE *f;
  int i, Fs, N, b, x, n, z0, i1, i2, th;
  float w, S;

  if(argc<6) { printf("usage: %s outfile Fs N bits threshold\n",argv[0]); return 0; }
  if(NULL==(f=fopen(argv[1],"rt"))) { printf("cannot read %s\n",argv[1]); return 0; }
  Fs=atoi(argv[2]); printf("Fs = %d Hz sampling frequency\n",Fs);
  N =atoi(argv[3]); printf("N  = %d points for Goertzel\n",N);
  if(N>MAXN) { printf("max N = %d\n",MAXN); fclose(f); return 0; }
  b =atoi(argv[4]); printf("b  = %d scaling is 2^b\n",b);
  S = (float)(1<<b); // scaling factor
  th=atoi(argv[5]); printf("th = %d threshold\n",th);

  printf("\nreference string: ");
  for(i2=0;i2<4;i2++) for(i1=0;i1<4;i1++) printf("%c",sym[symmtx[i1][i2]]);
  printf("\ndecoded   string: ");

  for(i=0;i<N;i++) { // init window (Hamming)
    win[i] = (int)round(S*(0.54 -0.46*cosf(2.0*M_PI*(float)i/(float)(N-1)))); }
```

```
  for(i=0;i<4;i++) { // init Goertzel constants
    // CORDIC may be used here to compute sin() and cos()
    w = 2.0*M_PI*round((float)N*(float)frow[i]/(float)Fs)/(float)N;
    cw[i] = (int)round(S*cosf(w)); c[i] = cw[i]<<1;
    sw[i] = (int)round(S*sinf(w));
    w = 2.0*M_PI*round((float)N*(float)fcol[i]/(float)Fs)/(float)N;
    cw[i+4] = (int)round(S*cosf(w)); c[i+4] = cw[i+4]<<1;
    sw[i+4] = (int)round(S*sinf(w)); }

  for(n=0;!feof(f);) {
    i = fscanf(f,"%d",&x); if(i<1) continue;
    x = ((x*win[n])>>b);                         // windowing
    if((n%N)==0) for(i=0;i<8;i++) { z1[i]=0; z2[i]=0; } // Goertzel reset
    // **** GOERTZEL ITERATION ****
    for(i=0;i<8;i++) {
      z0    = x + ((c[i]*z1[i])>>b) - z2[i];     // Goertzel iteration
      z2[i] = z1[i]; z1[i] = z0; }               // Goertzel status update
    // **** GOERTZEL ITERATION ****
    n++; if((n%N)==0) { n=0; // finalize and decode
      for(i1=i2=-1,i=0;i<8;i++) {
        // CORDIC may be used here to compute atan2() and sqrt()
        I[i] = ((cw[i]*z1[i])>>b) - z2[i];       // Goertzel final I
        Q[i] = ((sw[i]*z1[i])>>b);               // Goertzel final Q
        M2[i] = I[i]*I[i] + Q[i]*Q[i];           // magnitude squared
        if(M2[i]>th) {                           // DTMF decoding
          if(i<4) { if(i1==-1) i1=i;   else i1=4; }   // find 1st tone, one peak allowed
          else    { if(i2==-1) i2=i-4; else i2=4; }   // find 2nd tone, one peak allowed
        } }
      if((i1>-1)&&(i1<4)&&(i2>-1)&&(i2<4)) printf("%c",sym[symmtx[i1][i2]]);
      else printf(" ");
    } }
  printf("\n\n"); fclose(f); return 0;
}
```

The C output for a typical run on the same signal as the MATLAB® is the following:

```
C:\>GoertzelDTMFdec in.txt 4000 200 12 2000000

Fs = 4000 Hz sampling frequency
N  = 200 points for Goertzel
b  = 12 scaling is 2^b
th = 2000000 threshold

reference string: 147*2580369#ABCD
decoded   string: 147* 25800369 #ABBCD
```

Support material

Related design support material
STEVAL-WESU1: Wearable sensor unit kit
STEVAL-STLKT01V1: SensorTile kit
Documentation
Design tip DT0085: Coordinate rotation digital computer algorithm (CORDIC) to compute trigonometric and hyperbolic functions
Design tip DT0087: Coordinate rotation digital computer algorithm (CORDIC) test and performance verification

Revision history

Date	Version	Changes
11-Dec-2017	1	Initial release

IMPORTANT NOTICE – PLEASE 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. 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 acknowledgement.

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 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.

ST and the ST logo are trademarks of ST. 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.

© 2017 STMicroelectronics – All rights reserved