DT0085
Design tip

Coordinate rotation digital computer algorithm (CORDIC) to compute trigonometric and hyperbolic functions

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 Kbytes Flash, 8 Kbytes SRAM, 1 Kbyte 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

Purpose and benefits

This design tip explains how to compute trigonometric and hyperbolic functions using the coordinate rotation digital computer algorithm (CORDIC).

- This algorithm has very low complexity, using only shifts and adds. No floating point math. No FPU/DSP needed, therefore it is suited for cores such as the Cortex-M0.
- This algorithm has a very low memory footprint, using only a small look up table (LUT), with as many integer entries as the number of precision bits required. The ARM CMSIS library occupies 220kB, while CORDIC may be less than 1kB.
- This algorithm is fast, performing, on average, a number of iterations equal to the number of precision bits required. With respect to the single precision FPU that takes on average a constant time (T), the ARM CMSIS can be faster (T to T/3) and the CORDIC can be slower (T to 3T). Software emulation of single/double precision floating point math is the slowest solution (10T to 30T).
- The following trigonometric functions can be computed: sin(), cos(), atan(), atan2(). The following hyperbolic functions can be computed: sinh(), cosh(), atanh(). Other functions that are computed as a byproduct: sqrt(), exp(), ln(). The CORDIC always computes functions in pairs (see table below).

A utility is provided, in C source code format, to automatically generate the look up table (LUT) used by the CORDIC algorithm for all three coordinates system: circular, linear and hyperbolic

A reference fixed-point C source code is provided for all CORDIC modes: rotation and vectoring. A specialized code is also provided for the case of trigonometric functions.

In the companion Design Tip, DT0087 test code is provided to verify performance and error bounds.

Description

The CORDIC algorithm can be seen as a sequence of micro rotations (see Figure 1), where the vector XY is rotated by an angle A. Remembering that tan(A)=sin(A)/cos(A), the formula for the single micro rotation is the following:

$$X_{n+1} = \cos(A)\,X_n - \sin(A)\,Y_n = \cos(A)\,[\,X_n - \tan(A)\,Y_n\,]$$

$$Y_{n+1} = \sin(A)\,X_n + \cos(A)\,X_n = \cos(A)\,[\,\tan(A)\,X_n + Y_n\,]$$

The rotation angle is chosen so that the tan(A) coefficient is a power of two, therefore the multiplication is reduced to a bit shift. If the components are scaled by F=1/cos(A), which is the CORDIC gain, the formula for the rotation is reduced to only bit shifts and additions:

$$X_{n+1}\,F_n = [\,X_n - Y_n / 2^n\,]$$

$$Y_{n+1}\,F_n = [\,X_n / 2^n + Y_n\,]$$

The elementary rotation angle is $A_n = \mathrm{atan}(1/2^n)$. The corresponding scaling factor is $F_n = 1/\cos(A_n) = \mathrm{sqrt}(1+1/2^{2n})$.

The unified CORDIC algorithm

The unified CORDIC algorithm uses three registers: X and Y for the vector, and Z for angle. Only shifts and adds are done. The scaling factor is compensated at the end of the loop.

$$X_{n+1}\,F_n = [\quad X_n \quad -M\,S\,Y_n / 2^n\,]$$

$$Y_{n+1}\,F_n = [\quad S\,X_n / 2^n \quad +Y_n \quad]$$

$$Z_{n+1} \quad = \quad Z_n \quad - S\,a_n$$

The factor M, the elementary angle A_n and the corresponding scaling F_n are determined by the CORDIC coordinate system in use (see Figure 1):

- Circular: $A_n = \mathrm{atan}(1/2^n)$, n from 0, $M = +1$, $F_n = \mathrm{sqrt}(1+1/2^{2n})$
- Linear: $A_n = 1/2^n$, n from 0, $M = 0$, $F_n = \mathrm{sqrt}(1)$
- Hyperbolic: $A_n = \mathrm{atanh}(1/2^n)$, **n from 1**, $M = -1$, $F_n = \mathrm{sqrt}(1 - 1/2^{2n})$

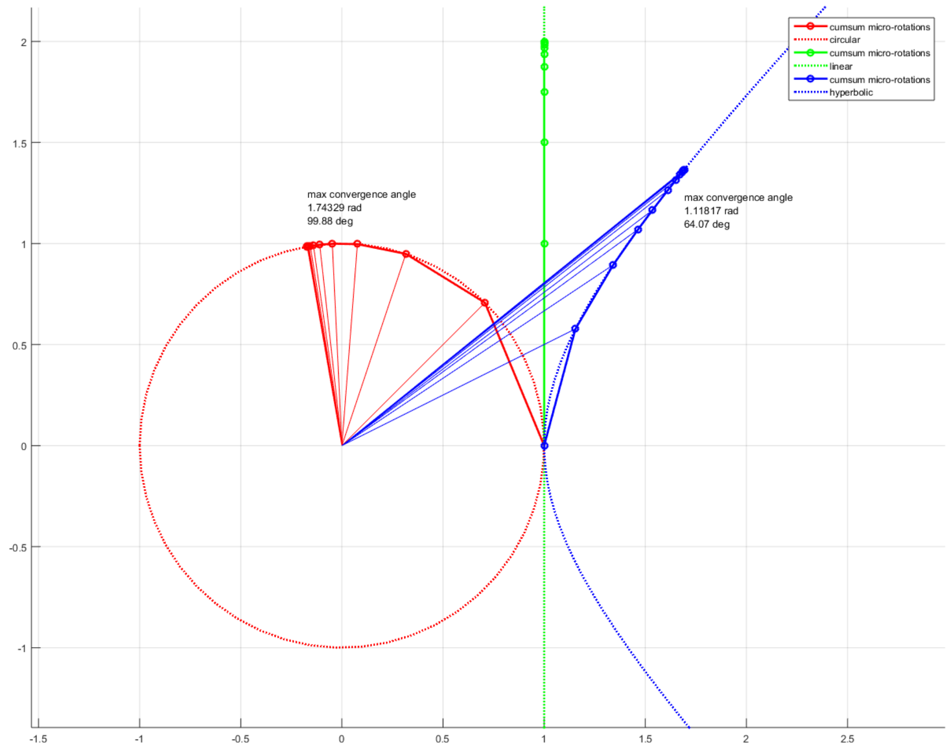
The factor S is determined by the CORDIC operation mode:

- Rotation: the angle Z is driven to 0; $S = +1$ if $Z>=0$, $S = -1$ if $Z<0$.
- Vectoring: the Y component is driven to 0; $S = +1$ if $Y<=0$, $S = -1$ if $Y>0$.

The angle in register Z must be less than the convergence angle, which is the sum of the angles A_n at each iteration. For the hyperbolic coordinate system, the following iterations must be repeated to obtain convergence: 4, 13, 40, … k, … 3k+1.

- Circular: $A = \sum A_n = 1.7432866$ radians (99.9deg), n = 0, 1, 2, 3, 4, 5, … N
- Linear: $A = \sum A_n = 2$, n = 0, 1, 2, 3, 4, 5, … N
- Hyperbolic: $A = \sum A_n = 1.1181730$ radians (64deg), n = **1**, 2, 3, **4, 4**, 5, … N

Figure 1. CORDIC micro rotations in the circular (red), linear (green) and hyperbolic (blue) coordinate system.



As only shifts and adds are done at each iteration, at the end of the loop the final vector components X and Y are scaled by the product of the scaling factors F_n.

- Circular: F = ∏ Fn = 1.64676025812107, 1/F=0.607252935008881
- Hyperbolic: F = ∏ Fn = 0.82978162013890, 1/F=1.20513635844646

Functions computed by CORDIC

The CORDIC algorithm always computes two functions simultaneously.

In the circular coordinate system: the sin() & cos() pair is frequently used in I/Q demodulators and direct digital frequency synthesizers DDFS. The atan() & sqrt() are conveniently used in PM/AM demodulators.

The function atan2(y,x) can be computed based on atan(y/x):

- If x>=0, atan2(y,x) = atan(y/x)
- If x<0 and y>=0, atan2(y,x) = atan(-y/-x) + pi
- If x<0 and y<0, atan2(y,x) = atan(-y/-x) – pi

Circular coordinate system. Rotation mode: Z is driven to 0. $\|Z_0\|<1.74$ (99.9deg), F~1.64			
$X_0 = x$, $Y_0 = y$, $Z_0 = a$	$X_N = F\ [x \cos(a) - y \sin(a)]$	$Y_N = F\ [x \sin(a) + y \cos(a)]$	$Z_N = 0$
$X_0 = m$, $Y_0 = 0$, $Z_0 = a$	$X_N = F\ m \cos(a)$	$Y_N = F\ m\ \sin(a)$	$Z_N = 0$
$X_0 = 1/F$, $Y_0 = 0$, $Z_0 = a$	$X_N = \cos(a)$	$Y_N = \sin(a)$	$Z_N = 0$
Circular coordinate system. Vectoring mode: Y is driven to 0. $\|Z_N\|<1.74$ (99.9deg), F~1.64			
$X_0 = x$, $Y_0 = y$, $Z_0 = z$	$X_N = F\ \mathrm{sqrt}(x^2 + y^2)$	$Y_N = 0$	$Z_N = z + \mathrm{atan}(y/x)$
$X_0 = 1$, $Y_0 = a$, $Z_0 = 0$	$X_N = F\ \mathrm{sqrt}(1 + a^2)$	$Y_N = 0$	$Z_N = \mathrm{atan}(a)$
$X_0 = a$, $Y_0 = 1$, $Z_0 = 0$	$X_N = F\ \mathrm{sqrt}(a^2 + 1)$	$Y_N = 0$	$Z_N = \mathrm{acot}(a)$

The linear coordinate system allows multiply-and-accumulate operations. Also, it can be used to perform division operations.

Linear coordinate system. Rotation mode: Z is driven to 0. $\|Z_0\|<2$, F=1			
$X_0 = x$, $Y_0 = y$, $Z_0 = z$	$X_N = x$	$Y_N = y + x*z$	$Z_N = 0$
Linear coordinate system. Vectoring mode: Y is driven to 0. $\|Z_N\|<2$, F=1			
$X_0 = x$, $Y_0 = y$, $Z_0 = z$	$X_N = x$	$Y_N = 0$	$Z_N = z + y/x$

Finally, the hyperbolic coordinate system is used to compute the exp() and ln() functions. Remember that exp(a) = sinh(a) + cosh(a). Also remember that atanh(y/x) = ½ ln((x+y)/(x-y)) and therefore atanh(a) = ½ ln((1+a)/(1-a)).

Hyperbolic coordinate system. Rotation mode: Z is driven to 0. $\|Z_0\|<1.11$ (64deg), F~0.82			
$X_0 = x$, $Y_0 = y$, $Z_0 = a$	$X_N = F\ [x \cosh(a) + y \sinh(a$	$Y_N = F\ [x \sinh(a) + y \cosh(a$	$Z_N = 0$
$X_0 = 1/F$, $Y_0 = 0$, $Z_0 = a$	$X_N = \cosh(a)$	$Y_N = \sinh(a)$	$Z_N = 0$
$X_0 = b/F$, $Y_0 = b/F$, $Z_0 = a$	$X_N = b \exp(a)$	$Y_N = b \exp(a)$	$Z_N = 0$
Circular coordinate system. Vectoring mode: Y is driven to 0. $\|Z_N\|<1.11$ (64deg), F~0.82			
$X_0 = x$, $Y_0 = y$, $Z_0 = z$	$X_N = F\ \mathrm{sqrt}(x^2 - y^2)$	$Y_N = 0$	$Z_N = z + \mathrm{atanh}(y/x)$
$X_0 = 1$, $Y_0 = a$, $Z_0 =$ (	$X_N = F\ \mathrm{sqrt}(1 - a^2)$	$Y_N = 0$	$Z_N = \mathrm{atanh}(a)$
$X_0 = a$, $Y_0 = 1$, $Z_0 =$ (	$X_N = F\ \mathrm{sqrt}(a^2 - 1)$	$Y_N = 0$	$Z_N = \mathrm{acoth}(a)$
$X_0 = a+b$, $Y_0 = a-b$, $Z_0 =$	$X_N = 2\ F\ \mathrm{sqrt}(a\ b)$	$Y_N = 0$	$Z_N = \ln(a/b) / 2$
$X_0 = a+1$, $Y_0 = a-1$, $Z_0 =$	$X_N = 2\ F\ \mathrm{sqrt}(a)$	$Y_N = 0$	$Z_N = \ln(a) / 2$
$X_0 = a+1/4$, $Y_0 = a-1/4$, $Z_0 =$	$X_N = F\ \mathrm{sqrt}(a)$	$Y_N = 0$	$Z_N = \ln(4\ a) / 2$

CORDIC fixed-point C implementation

Look Up Table generation code

The utility generates the file CORDICtable.c, which is to be included in CORDIC.c. The name is modified with the suffix _LIN or _HYPER when circular or hyperbolic coordinates are selected. The code below is formatted for compactness, not for readability.

```c
#include <stdio.h>
#include <math.h>
//#define M_PI 3.1415926536897932384626

int main(int argc, char **argv) {
  FILE *f; char tname[50], cname[10]; int n,n2,mp2,niter,bits,t;
  double F, A, mul, tmul; // CORDIC gain, convergence angle, multiplication factor

  printf("0)circular, 1)linear, 2)hyperbolic? "); scanf("%d",&t); switch(t) {
    case 0: sprintf(cname,"%s","");       break;
    case 1: sprintf(cname,"%s","_LIN");   break;
    case 2: sprintf(cname,"%s","_HYPER"); break;
  } sprintf(tname,"CORDICtable%s.c",cname);
  if(NULL==(f=fopen(tname,"wt"))) { printf("cannot write to %s\n",tname); return 0; }

  printf("number of bits for mantissa (e.g. 30)? "); scanf("%d",&bits);
  printf("0) mul factor is 2^n (easier output scaling), or\n"
         "1) 2pi is 2^n (easier implementation)\n ? "); scanf("%d",&mp2);
  printf("suggested multiplication factor ");
  if(mp2==0) { tmul=(double)(1<<(bits-3));      printf("2^%d = %f\n",    bits-3,tmul); }
  else       { tmul=(double)(1<<(bits-2))/M_PI; printf("2^%d/pi = %f\n",bits-2,tmul); }
  printf("multiplication factor (0 for suggested)? "); scanf("%lf",&mul);
  if(mul<0.1) { mul=tmul; printf("%f\n",mul); } else mp2=-1; // custom mul factor
  switch(t) {
    case 0:  for(n=0;     n<bits;n++)
              if((int)round( atan(pow(2.0,(double)(-n)))*mul)==0) break;    break;
    case 1:  for(n=0;     n<bits;n++)
              if((int)round(     (pow(2.0,(double)(-n)))*mul)==0) break;    break;
    case 2:  for(n=1,n2=4;n<bits;   ) {
              if((int)round(atanh(pow(2.0,(double)(-n)))*mul)==0) break;
              if(n==n2) n2=3*n+1; else n++;                           } break;
  } printf("iterations (up to %d)? ",n); scanf("%d",&niter);

  F=1.0; A=0.0; switch(t) {
    case 0: for(n=0;     n<niter;n++) {
              F=F*sqrt(1+pow(2.0,-2.0*n)); A+= atan(pow(2.0,(double)(-n))); } break;
    case 1: for(n=0;     n<niter;n++) {
              F=F*sqrt(1                ); A+=     (pow(2.0,(double)(-n))); } break;
    case 2: for(n=1,n2=4;n<niter;   ) {
              F=F*sqrt(1-pow(2.0,-2.0*n)); A+=atanh(pow(2.0,(double)(-n)));
              if(n==n2) n2=3*n+1; else n++;                                 } break; }

  fprintf(f,"//CORDIC%s, %d bits, %d iterations\n",cname,bits,niter);
  fprintf(f,"// 1.0 = %f multiplication factor\n", mul);
  switch(t) {
    case 0:  fprintf(f,"// A   = %lf convergence angle "
                       "(limit is 1.7432866 = 99.9deg)\n",A);
             fprintf(f,"// F   = %lf gain (limit is 1.64676025812107)\n",F);
             fprintf(f,"// 1/F = %lf inverse gain (limit is 0.607252935008881)\n",1.0/F);
             break;
    case 1:  fprintf(f,"// A   = %lf convergence angle (limit is 2)\n",A);
             fprintf(f,"// F   = %lf gain (limit is 1.0)\n",F);
             fprintf(f,"// 1/F = %lf inverse gain (limit is 1.0)\n",1.0/F);
             break;
    case 2:  fprintf(f,"// A   = %lf convergence angle "
                       "(limit is 1.1181730 = 64.0deg)\n",A);
             fprintf(f,"// F   = %lf gain (limit is 0.82978162013890)\n",F);
             fprintf(f,"// 1/F = %lf inverse gain (limit is 1.20513635844646)\n",1.0/F);
             break;
  }
  fprintf(f,"// pi  = %lf (3.1415926536897932384626)\n",M_PI); fprintf(f,"\n");
  fprintf(f,"#define CORDIC%s_A        %f // CORDIC convergence angle A\n",cname,A);
  fprintf(f,"#define CORDIC%s_F        0x%08X // CORDIC gain F\n",
    cname,(int)round(mul*F));
  fprintf(f,"#define CORDIC%s_1F       0x%08X // CORDIC inverse gain 1/F\n",
    cname,(int)round(mul/F));
```

```c
  fprintf(f,"#define CORDIC%s_HALFPI   0x%08X\n",cname,(int)round(mul*(M_PI/2.0)));
  fprintf(f,"#define CORDIC%s_PI       0x%08X\n",cname,(int)round(mul*(M_PI)));
  fprintf(f,"#define CORDIC%s_TWOPI    0x%08X\n",cname,(int)round(mul*(2.0*M_PI)));
  fprintf(f,"#define CORDIC%s_MUL      %f // CORDIC multiplication factor M",cname,mul);
  switch (mp2) {
         case 0:  fprintf(f," = 2^%d\n",   bits-3); break;
         case 1:  fprintf(f," = 2^%d/pi\n",bits-2); break;
         default: fprintf(f,"\n");                  break;
  }
  fprintf(f,"#define CORDIC%s_MAXITER  %d\n\n",cname,niter);

  fprintf(f,"int CORDIC%s_ZTBL[] = {",cname);
  for(n=0;n<niter;n++) {
    if((n%8)==0) fprintf(f,"\n  ");
    switch(t) {
      case 0: fprintf(f,"0x%08X",(int)round( atan(pow(2.0,(double)(-n)))*mul)); break;
      case 1: fprintf(f,"0x%08X",(int)round(     (pow(2.0,(double)(-n)))*mul)); break;
      case 2: n=n==0?1:n;
              fprintf(f,"0x%08X",(int)round(atanh(pow(2.0,(double)(-n)))*mul)); break; }
    if(n<(niter-1)) fprintf(f,", "); else fprintf(f," "); }
  fprintf(f,"};\n\n"); fclose(f); printf("table written to %s\n",tname); return 0;
}
```

CORDIC code for circular coordinates

```c
#include "CORDICtable.c"

// z less than convergence angle (limit is 1.7432866 = 99.9deg) multiplied by M
void CORDIC_rotation_Zto0(int x, int y, int z, int *xx, int *yy)
{ int k, tx;
  for (k=0; k<CORDIC_MAXITER; k++) {
    tx = x;
    if (z>=0) { x -= (y>>k); y += (tx>>k); z -= CORDIC_ZTBL[k]; }
    else      { x += (y>>k); y -= (tx>>k); z += CORDIC_ZTBL[k]; } }
  *xx = x; // x*cos(z)-y*sin(z) multiplied by M and gain F
  *yy = y; // x*sin(z)+y*cos(z) multiplied by M and gain F
}

void CORDIC_vectoring_Yto0(int x, int y, int z, int *xx, int *zz) {
  int k, tx;
  for (k=0; k<CORDIC_MAXITER; k++) {
    tx = x;
    if (y<=0) { x -= (y>>k); y += (tx>>k); z -= CORDIC_ZTBL[k]; }
    else      { x += (y>>k); y -= (tx>>k); z += CORDIC_ZTBL[k]; } }
  *xx = x; // sqrt(x^2+y^2) multiplied by gain F
  *zz = z; // z+atan2(y,x) multiplied by M
}
```

CORDIC code for linear coordinates

```c
#include "CORDICtable_LIN.c"

// z less than convergence angle (limit is 2) multiplied by M
void CORDIC_LIN_rotation_Zto0(int x, int y, int z, int *xx, int *yy)
{ int k, tx;
  for (k=0; k<CORDIC_LIN_MAXITER; k++) {
    tx = x;
    if (z>=0) { y += (tx>>k); z -= CORDIC_LIN_ZTBL[k]; }
    else      { y -= (tx>>k); z += CORDIC_LIN_ZTBL[k]; } }
  *xx = x; // x     multiplied by M  (gain F=1)
  *yy = y; // y+x*z multiplied by M  (gain F=1)
}

void CORDIC_LIN_vectoring_Yto0(int x, int y, int z, int *xx, int *zz) {
  int k, tx;
  for (k=0; k<CORDIC_LIN_MAXITER; k++) {
    tx = x;
    if (y<=0) { y += (tx>>k); z -= CORDIC_LIN_ZTBL[k]; }
    else      { y -= (tx>>k); z += CORDIC_LIN_ZTBL[k]; } }
  *xx = x; // x     as is (gain F=1)
  *zz = z; // z+y/x multiplied by M

}
```

CORDIC code for hyperbolic coordinates

```
#include "CORDICtable_HYPER.c"

// z less than convergence angle (limit is 1.1181730 = 64.0deg) multiplied by M
void CORDIC_HYPER_rotation_Zto0(int x, int y, int z, int *xx, int *yy)
{ int k, k2, tx;
  for (k=1,k2=4; k<CORDIC_HYPER_MAXITER;) {
    tx = x;
    if (z>=0) { x += (y>>k); y += (tx>>k); z -= CORDIC_HYPER_ZTBL[k]; }
    else      { x -= (y>>k); y -= (tx>>k); z += CORDIC_HYPER_ZTBL[k]; }
    if(k==k2) k2=k*3+1; else k++;                                        }
  *xx = x; // x*cosh(z)+y*sinh(z) multiplied by M and gain F
  *yy = y; // x*sinh(z)+y*cosh(z) multiplied by M and gain F
}

void CORDIC_HYPER_vectoring_Yto0(int x, int y, int z, int *xx, int *zz) {
  int k, k2, tx;
  for (k=1,k2=4; k<CORDIC_HYPER_MAXITER;) {
    tx = x;
    if (y<=0) { x += (y>>k); y += (tx>>k); z -= CORDIC_HYPER_ZTBL[k]; }
    else      { x -= (y>>k); y -= (tx>>k); z += CORDIC_HYPER_ZTBL[k]; }
    if(k==k2) k2=k*3+1; else k++;                                        }
  *xx = x; // sqrt(x^2+y^2) multiplied by gain F
  *zz = z; // z+atan2(y,x) multiplied by M

}
```

CORDIC code circular coordinates, specialized for trigonometric functions

```
#include "CORDICtable.c"

// angle is radians multiplied by CORDIC multiplication factor M
// modulus can be set to CORDIC inverse gain 1/F to avoid post-division
void CORDICsincos(int a, int m, int *s, int *c) {
  int k, tx, x=m, y=0, z=a, fl=0;
  if      (z>+CORDIC_HALFPI) { fl=+1; z = (+CORDIC_PI) - z; }
  else if (z<-CORDIC_HALFPI) { fl=+1; z = (-CORDIC_PI) - z; }
  for (k=0; k<CORDIC_MAXITER; k++) {
    tx = x;
    if (z>=0) { x -= (y>>k); y += (tx>>k); z -= CORDIC_ZTBL[k]; }
    else      { x += (y>>k); y -= (tx>>k); z += CORDIC_ZTBL[k]; } }
  if (fl) x=-x;
  *c = x; // m*cos(a) multiplied by gain F and factor M
  *s = y; // m*sin(a) multiplied by gain F and factor M
}

void CORDICatan2sqrt(int *a, int *m, int y, int x) {
  int k, tx, z=0, fl=0;
  if (x<0) { fl=((y>0)?+1:-1); x=-x; y=-y; }
  for (k=0; k<CORDIC_MAXITER; k++) {
    tx = x;
    if (y<=0) { x -= (y>>k); y += (tx>>k); z -= CORDIC_ZTBL[k]; }
    else      { x += (y>>k); y -= (tx>>k); z += CORDIC_ZTBL[k]; } }
  if (fl!=0) { z += fl*CORDIC_PI; }
  *a = z; // radians multiplied by factor M
  *m = x; // sqrt(x^2+y^2) multiplied by gain F
}

void CORDICatansqrt(int *a, int *m, int y, int x) {
  int k, tx, z=0;
  if (x<0) { x=-x; y=-y; }
  for (k=0; k<CORDIC_MAXITER; k++) {
    tx = x;
    if (y<=0) { x -= (y>>k); y += (tx>>k); z -= CORDIC_ZTBL[k]; }
    else      { x += (y>>k); y -= (tx>>k); z += CORDIC_ZTBL[k]; } }
  *a = z; // radians multiplied by factor M
  *m = x; // sqrt(x^2+y^2) multiplied by gain F
}
```

Support material

Related design support material
Wearable sensor unit reference design, STEVAL-WESU1
SensorTile development kit, STEVAL-STLKT01V1
Documentation
Design tip, DT0087, Coordinate rotation digital computer algorithm (CORDIC) test and performance verification

Revision history

Date	Version	Changes
16-Nov-2017	1	Initial release