ArcTangent in PSoC® 1 Assembler

Author: Dave Van Ess

Associated Project: Yes
Associated Part Family: Any PSoC 1 Device
Software Version: PSoC Designer™ 5.4
Related Application Notes: None

Many control applications require calculating an angular position when the Cartesian position data is given. The arctangent function makes this possible. A major setback to this is that the trigonometric functions supplied with the 'C' math library uses and returns float variables. Although very accurate, the processing overhead resulting from using the floating-point math routines can be prohibitive. Techniques are discussed to calculate an arctangent to a specific resolution. Software is presented using these techniques for an arctangent function that returns the calculated angle in 100ths of a degree resolution using signed 8-bit X and Y values.

Introduction

You have Cartesian data and require an angular position. It may be a compass, calculating direction from magnetic field detectors or it may be to determine a shaft position using Hall Effect sensors. This is why arctangent (atan and atan2) functions supplied with the 'C' math library is included in PSoC Designer™. These functions have two major limitations.

The first limitation is that they return the angle value in radians. Radians, although frequently used in math and physics, are not as useful to most engineers. 2 pi radians make up a circle, as opposed to 360 degrees. The second limitation is that the function uses floating point. This results in extremely accurate answers but at the cost of computation time. In embedded applications, most data comes out of an ADC in an integer format and the resolution required is on the order of 10°, 5°, 1°, or 0.1° degrees.

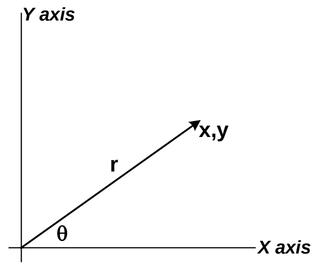
The function developed for this Application Note uses signed 8-bit X position and Y position data.

The angle is returned in integer format in hundredths of a degree resolution. It is a signed integer and has a range of ±18000. The accuracy is ±0.25° (or 25 counts).

Polar and Cartesian Coordinates

Figure 1 shows a two-dimensional position in terms of its Cartesian and Polar parameters.

Figure 1. Angular Geometry



Cartesian coordinates define the distance away from the origin x units horizontally and y units vertically. The polar coordinates define the position r units away from the origin at an angle **θ** above the horizontal axis. These values are different ways of defining the same unique position. Their relationship is shown in these equations.

$$r = \sqrt{x^2 + y^2}$$ **Equation 1**

$$\theta = f(x, y)$$ **Equation 2**

The function f is defined as the arctangent and is usually defined as:

$$f(x, y) = \text{atan}(y / x)$$ **Equation 3**

y/x is defined as the slope. This function has two difficult characteristics. First, this slope can vary anywhere in the range of ±∞. This makes it hard to find angle close to the vertical axis. (atan(255/1) = 89.78°). Second, this function generates the same solution for two different positions (because y/x = -y/-x).

Another arctangent function is the atan2 function, which is defined below:

$$f(x,y) = \operatorname{atan}2(x,y) \qquad \textbf{Equation 4}$$

Both difficulties with the first function do not exist with this new function. It does require two variables be passed to it. This is the function implemented in this Application Note.

Vector Math 101

If the position vector is expressed as a complex number then the following equation holds true:

$$c_n = x_n + i \cdot y_n = r_n \angle \theta_n \qquad \textbf{Equation 5}$$

If two vectors are multiplied, the result is:

$$\begin{aligned} c_n c_m &= (x_n x_m - y_n y_m) + i(x_n y_m + y_n x_m) \\ &= r_n r_m \angle(\theta_n + \theta_m) \end{aligned} \qquad \textbf{Equation 6}$$

This equation shows that a vector can be moved by a specific angle. If a vector has an angle of 46°, multiplying it with a vector having an angle of -45° results in a vector with an angle of 1°. Clearly, a series of vector multiplication can be used to move the vector to the horizontal axis. The sum of all the applied vector angles equals the original vector angle and the arctangent calculation is complete.

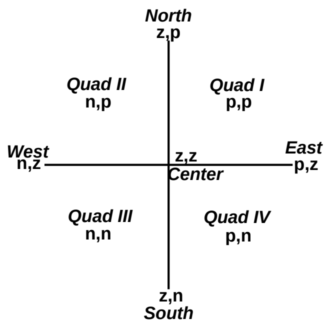
Quadrant Decoding and Translation

Each X value and Y value has three possible states. They are:

- Negative
- Zero
- Positive

This allows for nine different combinations as shown in Figure 2.

Figure 2. Nine States for X-Y Data



It is necessary to determine in which of the nine states the data falls. The following code shows how a state variable is generated.

Code 1. Quadrant Encoding

```
#define EAST     0x0d
#define NORTH    0x07
#define WEST     0x01
#define SOUTH    0x04
#define CENTER   0x05
#define QUADI    0x0f
#define QUADII   0x03
#define QUADIII  0x00
#define QUADIV   0x0c

bStatus = 0;
if(cXval >= 0) bStatus|= 0x08;
if(cXval == 0) bStatus|= 0x04;
if(cXval >= 0) bStatus|= 0x02;
if(cXval == 0) bStatus|= 0x01;
```

This status variable can be used to generate a jump table or switch statement.

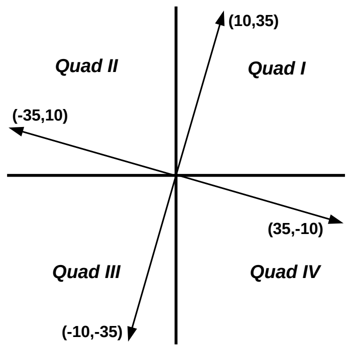
Of these nine states North, South, East, and West are easily calculated. Center value is indeterminate, but must receive some value. For this routine it is defined to be 32,767. (The largest integer value, it is much larger than the maximum in-range value of 18,000.) Code to set the angle for these five states is shown below.

Code 2. Quadrant Angle Offset

```
if(bStatus == EAST)  iAngle =     0;
if(bStatus == NORTH) iAngle =  9000;
if(bStatus ==WEST)   iAngle =-18000;
if(bStatus ==SOUTH)  iAngle = -9000;
if(bStatus == CENTER)iAngle = 32767;
```

For the remaining states it is necessary to move them to the Quad I state. Quad II vectors are rotated 90° counter clockwise to move into Quad I. Quad III vectors are rotated 180° to move into Quad I. Quad IV vectors are rotated 90° clockwise to move into Quad I. In each case, the Cartesian coordinates of the rotated vector only change in either polarity or swap values. This is shown in the figure below.

Figure 3. Vector Position in Different Quadrants



For each quadrant, that vector must be moved into Quad I and the angle must be appropriately adjusted as shown in the following code.

Code 3. Vector Movement between Quadrants

```
if(bStatus == QUADI)
   iAngle = iATanRoutine( Xval, Yval);
if(bStatus == QUADII)
   iAngle = iATanRoutine(-Yval, Xval) +
9000;
if(bStatus == QUADIII)
   iAngle = iATanRoutine(-Xval,-Yval) –
18000;
if(bStatus == QUADIV)
   iAngle = iATanRoutine( Yval,-Xval)-
9000;
```

A Complete Arctangent Function

Now that the vector has been moved into Quad I, it is necessary to write an efficient arctangent routine.

For this algorithm, a series of vectors are used to move the vector to the horizontal position. If the vector rotation caused the Yval to become negative, the vector has moved into Quad IV and this rotation is ignored. If not, the rotated vector stayed in Quad I and the angle of the rotated vector is added to the accumulated angle value.

Equation (6) shows the four simple multiplications required to perform a complex vector multiplication. To simplify this computation, the X and Y values of the rotation vectors selected will both be powers of two. This is shown in the equation below.

$$c_n = 2^n - i_n$$ **Equation 7**

Using Equation (6) with Equation (7) the production of the rotation vector and the vector is shown in Equation (8) below:

$$\left(2^n Xval + Yval\right) + i\left(2^n Yval - Xval\right)$$ **Equation 8**

Equation (8) shows that the rotation is necessary only if 2^nYval >= Xval.

A table of the rotation vectors and calculated rotation angle is shown below.

Table 1. Rotation Vectors and Calculated Angles

n	$X_{rot}(2^n)$	Y_{rot}	θ
0	1	-1	45.0000°
1	2	-1	26.5651°
2	4	-1	14.0362°
3	8	-1	7.1250°
4	16	-1	3.5763°
5	32	-1	1.7899°
6	64	-1	0.8952°
7	128	-1	0.4476°
8	256	-1	0.2238°
9	512	-1	0.1119°
10	1024	-1	0.0560°

This table shows that the greater the number of vectors, the greater the resolution. Conversely, a lower resolution answer requires fewer vectors. The specific number of vectors can be determined for each particular application. For this routine, nine vectors (0 through 8) are used. An implementation is shown in the following code.

Code 4. Arctangent Calculation by Vector Rotation

```
int const Theta[9]={4500,2657,1404,713
                    ,358,179,90,45,33};

iAngle = 0;
for(n = 0; n<9; n++){
   if(Yval <<n >= Xval){
      iAngle += Theta[n];
      Xtemp = Xval;
      Ytemp = Yval;
      Yval <<=n;
      Xval <<=n;
      Xval += Ytemp;
      Yval -= Xtemp;
      if( Xval > 65535){
        Xval = (Xval + 128)>>8;
        Yval = (Yval + 128)>>8;
      }
   }
}
```

Note that when the calculated Xval exceeds 16 bits, it, and Yval, is shifted down by 8 bits. This keeps the maximum size of Xval to no larger than 24 bits, unconstrained it could get as large as 40 bits. This truncation adds an error in the range of ±0.06°, but is well worth the saved CPU overhead.

iArcTanRoutine is the function included and can be found in the project associated with this application note. It is located in *ArcTanFunction.asm*. *ArcTanFunction.h* is also included to allow 'C' programs access this function.

This function was written in assembly to minimize CPU overhead. It requires 414 byes of ROM and takes a maximum of 4071 CPU cycles to complete. For a 24 MHz CPU clock, this function takes no more than 170 µsec to complete.

A Faster ArcTangent

At present 170 µsec is a respectable number, but if a faster technique is needed, there are other methods.

The code used to determine quadrant position stays the same while any improvements in time saving come from the actual arctangent function.

For this new function, the data is going to be moved to the lower half of Quad I. It is done by multiplying the vector with 1-I whenever Yval >= Xval.

An implementation is shown in the code below.

Code 5.

```
If(Yval >= Xval){
   iAngle +=4500;
   Ytemp = Yval;
   Yval = Yval –Xval;
   Xval = Xval + Ytemp;
}
```

Now it is guaranteed that Yval is less than Xval. The next step is to normalize Yval given the following equation:

$$Yval = \frac{Yval \cdot 256}{Xval} \quad \textbf{Equation 9}$$

Yval is now an integer between 0 and 255. A lookup table can be generated to solve the arctangent for all 256 of these cases using the equation below:

$$\text{Atantable}[n] = \text{atan}\left(\frac{Yval}{256}\right) \cdot \frac{50}{9} \quad \textbf{Equation 10}$$

The atan operation is multiplied by 50/9 to get a value that best fits in an 8-bit variable. (artan(255/256)*50/9 = 249). The answer is later multiplied by 18 to get the desired 0.01° resolution. This truncation adds an error in the range of ±0.09°, but is well worth space saved in the lookup table. The normalization also contributes a truncation error. The combination of both these errors falls well within the ±0.25º requirement. An implementation is shown in the code below.

Code 6.

```
Extern int const Atantable[];

Yval = (Yval<<8 + Xval>>1)/Xval;
iAngle += (Atantable[Yval] * 18);
```

The function *iArcTanRoutineTableLookup* can be found in the project associated with this note. It is located in *ArcTanFunctionLT.asm*. *ArcTanFunctionLT.h* is also included to allow 'C' programs access to this function.

This function was written in assembly to minimize CPU overhead. It requires 526 byes of ROM and takes a maximum of 1710 CPU cycles to complete. For a 24 MHz CPU clock, this function takes no more than 72 µsec to complete. This is a significant improvement in speed for an additional 112 byes of ROM.

An Even Faster ArcTangent

As respectable as 72 µsec is, there are users that still want faster operation. This can be done by exploiting the relationship in the equation shown below:

$$\text{atan}(y/x) = 90^{o} - \text{atan}(x/y) \quad \textbf{Equation 11}$$

This means that it is only necessary to calculate the arctangent were Yval is less than Xval.

The total range of Xval and Yval is -128 through 127. Moving the vector into Quad I now limits the range of both to 0 though 128. Limiting the calculation to the range in the lower half of Quad I, limits the range to:

- 0 < Xval ≤ 128
- 0 < Yval < Xval

Also, whenever Xval is less the 65, Xval and Yval can be repeatedly doubled until:

- 65 < Xval ≤ 128
- 0 < Yval < Xval

Given these ranges, there are 6176 different vectors. It is possible to build a lookup table to store the arctangent of each value. As with Equation (10), the atan operation is multiplied by 50/9 to get a best fit into an 8-bit variable. Again, this introduces a ±0.09º error.

This table starts with 65 atan values for n/65, followed by 66 atan values for n/66, followed by 67 values for n/67, followed by all the possible atan combinations ending with atan value for 127/128. Clearly, a lookup table is needed to determine the atan values for a particular Xval. The equation below shows how to develop this table:

$$\text{ArrayPos}[n] = \begin{matrix} 0 & ; n \le 65 \\ \text{ArrayPos}[n-1] + n - 1 & ; n > 65 \end{matrix} \quad \textbf{Equation 12}$$

An implementation is shown in the code example below.

Code 7. ArcTangent Vector Table Approach

```
fextern int ArrayPos[];
extern char aTanArray[];

if(Xval == Yval){
   iAngle += 4500;
}
else if(Xval>Yval){
   while(Xval <65){Xval<<=1; Yval<<=1;}
   iAngle +=
18*aTanArray[ArrayPos[Xval]+Yval];
}
else if(Xval<Yval){
   while(Yval <65){Yval<<=1; Xval<<=1;}
   iAngle += 9000;
   iAngle -= 18*aTanArray[ArrayPos[Yval]+Xval];
}
```

The function *iArcTanRoutineInsanelyFast* can be found in the project associated with this note. It is located in *ArcTanFunctionIF.asm*. *ArcTanFunctionIF.h* is also included to allow 'C' programs access to this function. The 6176-byte lookup table is located in *WastefullyLargeTable.asm*.

This function was written in assembly to minimize CPU overhead. It takes a maximum of 591 CPU cycles to complete. For a 24 MHz CPU clock, this function takes no more than 25 µsec to complete. Although this is a significant speed improvement, it comes at the cost of a staggering 6,556 byes of ROM.

Although inefficient in code space, this implementation is useful for high-speed angular control applications. With this algorithm it is possible to read two sensors, determine the angular position, and perform some control code loop calculation all at an update rate of 10 ksps.

Even Faster Potential ArcTangents

Of the 591 CPU cycles used in the previous implementation, 202 were used to shift the data into range. It is possible to alter the lookup table to include values for Xval in the range of 1 to 128. This would increase the size of the lookup table to 8,256 bytes. It would now take no more than 389 CPU cycles to complete. For a 24 MHz CPU clock, this is less than 17 μsec. The code overhead is 8750 bytes of ROM.

Other Advantages

The arctangent functions developed in this note are all designed to output a signed integer value in the range of ±18000 100ths of a degree, with East = 0º and North = 90º. It would be easy to change this function so that North = 0º and East = -90º. There is no reason why the output has to be in degrees. Suppose your application requires calculating the angular position of a 220-tooth gear. It may be desirable to output an unsigned integer in the range of 0 to21999 100ths of a tooth. This saves the extra step of converting degrees to teeth. Any time calculations can be moved from the embedded microcontroller to constant calculation before compilation, code execution time is reduced.

Summary

The arctangent function makes this possible to convert Cartesian position data to an angular format. For real time applications it must be carried out quickly and may rule out using the functions supplied with the 'C' math library. Vector multiplication gives an easy method to quickly determine the desired angle. Lookup tables allow even faster computation of the desired angle but at the cost of increased ROM usage. Of the four examples discussed, one should be perfect for your particular application.

About the Author

Name:	Dave Van Ess
Title:	Principal Application Engineer Cypress Semiconductor
Background:	An Engineer by training, a poet by temperament, and an outlaw in Nebraska. Dave is capable of abstract thought, concrete analysis, and ruthless implementation. BSEE from University of California, Berkeley. More than 28 Years experience in circuit, signal processing, digital, software, analog, and system design. Holder of six U.S. Patents (plus three pending) for medical systems, signal processing, and digital block enhancements. Author of numerous Application Notes, web casts, and technical articles. Joined Cypress Semiconductor at the dawn of the New Millennium.

Document History

Document Title: ArcTangent in PSoC® 1 Assembler – AN2341

Document Number: 001-26179

Revision	ECN	Orig. of Change	Submission Date	Description of Change
**	1445383	MAXK	09/07/2007	New application note.
*A	3328685	MAXK	07/26/2011	Updated document title as "ArcTangent in PSoC® 1 Assembler - AN2341". Updated Software Version as "PSoC® Designer™ 5.1 SP1.1" in page 1. Updated attached associated project for PSoC Designer 5.1.
*B	4508004	MSUR	09/19/2014	Updated to new template. Completing Sunset Review.
*C	4599724	ASRI	12/17/2014	Updated Software Version as "PSoC Designer™ 5.4" in page 1. Removed reference of AN2101, AN2038 in Related Application Notes in page 1 as these application notes are obsolete. Updated attached associated project for PSoC Designer 5.4.

Worldwide Sales and Design Support

Cypress maintains a worldwide network of offices, solution centers, manufacturer's representatives, and distributors. To find the office closest to you, visit us at Cypress Locations.

Products

Automotive	cypress.com/go/automotive
Clocks & Buffers	cypress.com/go/clocks
Interface	cypress.com/go/interface
Lighting & Power Control	cypress.com/go/powerpsoc cypress.com/go/plc
Memory	cypress.com/go/memory
PSoC	cypress.com/go/psoc
Touch Sensing	cypress.com/go/touch
USB Controllers	cypress.com/go/usb
Wireless/RF	cypress.com/go/wireless

PSoC® Solutions

psoc.cypress.com/solutions

PSoC 1 | PSoC 3 | PSoC 4 | PSoC 5LP

Cypress Developer Community

Community | Forums | Blogs | Video | Training

Technical Support

cypress.com/go/support

Cypress Semiconductor	Phone	: 408-943-2600
198 Champion Court	Fax	: 408-943-4730
San Jose, CA 95134-1709	Website	: www.cypress.com