KSP Math Library

Technical Guide

V215

R. D. Villwock
4-2-09

Table Of Contents

1.0 The Trig Functions

2.0 The Fast Log Functions

3.0 Extended Log and Exponential Functions

4.0 The Root3 Function

5.0 The ATFade Function

6.0 Format Conversion Routines

Appendix A

1.0 TRIG FUNCTIONS

1.1 Introduction

The Trig Functions for the **KSP Math Library** are implemented using the **Binary Cordic Algorithm**, usually credited to **Jack Volder** (circa 1959). While it doesn't converge too rapidly, requiring about one iteration per bit of accuracy, it has many virtues that make up for this. The iteration loop body is extremely simple and requires only signed addition and right shifting. Thus, the needed iterations can be performed very quickly. Moreover, the algorithm can provide both the **Sine** and **Cosine** in one pass.

1.2 Overview of the Cordic Algorithm

The Cordic hinges on two basic ideas. First, that any number can be approximated by an algebraic summation of an appropriately diminishing series of fixed values. And second, that such an 'appropriately diminishing series' is very conveniently produced with some rather clever simplifications of the basic vector rotation formulae from Analytic Geometry. To illustrate the general idea, suppose we want to find the **Sine** and **Cosine** of some arbitrary angle θ. We begin with a vector V_0 lying on the X-axis and then rotate it by some fixed series of reducing angles ϕ_1, ϕ_2, ϕ_3,... such as the binary series 45°, 22.5°, 11.25°, etc. We always rotate 'toward' the desired input angle θ by using both positive (counter-clockwise, CCW) and negative (clockwise, CW) rotations as needed, see **Figure 1-1**. The 1st CCW rotation of $\phi_1 = +45°$, overshoots θ so we next rotate CW by $\phi_2 = -22.5°$. This doesn't quite make it back to θ so again we rotate CW by $\phi_3 = -11.25°$ which then undershoots θ so our next rotation will be CCW by +5.625°, and so on. As we do this, the algebraic summation $\phi_S = \phi_1 + \phi_2 + \phi_3 + ...$ of these rotation angles will quickly approach (and begin to 'hover' around) the input angle θ. For example, in **Figure 1-1**, after the first 3 rotations, $\phi_S = 45° - 22.5° - 12.25° = 10.25°$. Note that for each successive rotation, ϕ_S can 'hover' both above and below θ with an ever-decreasing difference (since each new rotation angle is smaller than its predecessor).

So we begin with $X_0 = V_0$ and $Y_0 = 0$, and then iteratively calculate new XY pairs for each rotation. After any arbitrary number of rotations **n**, the vector V_n will be at an angle ϕ_{Sn}. Of course at any angle, $X_n = V_n \cdot \cos(\phi_{Sn})$ and $Y_n = V_n \cdot \sin(\phi_{Sn})$. Now, if the length of the vector V_n is one unit, then $X_n = \cos(\phi_{Sn})$ and $Y_n = \sin(\phi_{Sn})$. But, after a sufficient number of rotations **N**, ϕ_{SN} will become θ (within some desired degree of accuracy) and, the values X_N and Y_N will be the desired result. That is $\mathbf{X_N = \cos(\theta)}$ and $\mathbf{Y_N = \sin(\theta)}$. By extending the ϕ series far enough, the last value can be made as small as desired. Therefore, for a given ϕ series, the degree of accuracy by which ϕ_S can match any input angle θ is determined by the number of iterations performed. But, the key to the feasibility of this technique lies in choosing the ϕ series in such a way that the mathematics involved in all these rotations can be made very simple and 'computer-friendly'. The means by which the Cordic achieves this will be discussed next.
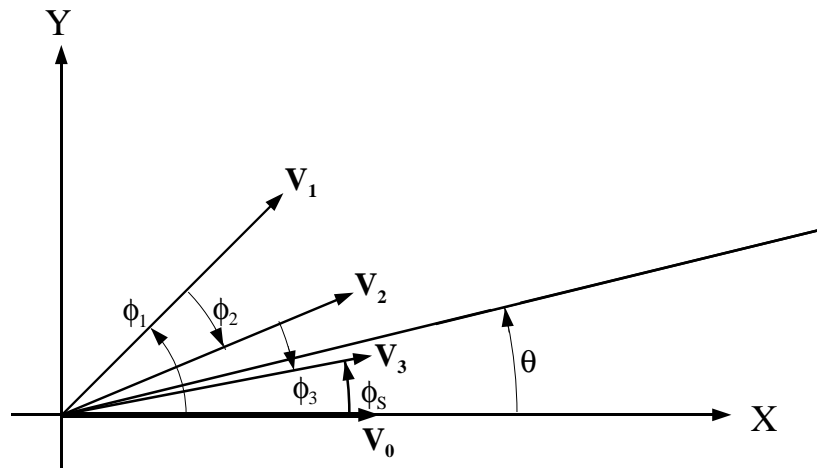

Figure 1-1

1.3 Rotation Mathematics

Refer now to **Figure 1-2**. If we have a vector of length **V** at some arbitrary angle θ, this vector can be described in terms of its X-axis and Y-axis projections where x = **V**·cos(θ) and y = **V**·sin(θ). Now, if we rotate the vector **V** by an angle φ (relative to θ), keeping the vector length the same, we can denote the new vector as **V'** (and its X-axis and Y-axis projections) as x' and y' respectively (Figure 1-2).
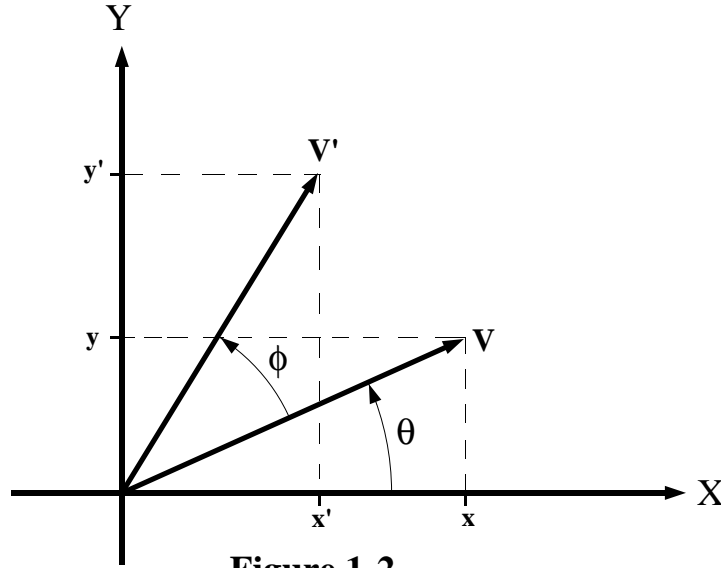

Figure 1-2

From Analytic Geometry, the coordinate rotation equations that relate x' y' to xy are as follows:

$$\textbf{(1-1)} \quad \textbf{x' = x·cos(φ) – y·sin(φ)}$$
$$\textbf{(1-2)} \quad \textbf{y' = y·cos(φ) + x·sin(φ)}$$

By factoring out cos(φ) and changing to subscript notation, equations (1-1) and (1-2) can be rewritten as:

$$\textbf{(1-3)} \quad \textbf{x}_{n+1} = \textbf{cos(φ}_n\textbf{)[x}_n – \textbf{y}_n\textbf{·tan(φ}_n\textbf{)]}$$
$$\textbf{(1-4)} \quad \textbf{y}_{n+1} = \textbf{cos(φ}_n\textbf{)[y}_n + \textbf{x}_n\textbf{·tan(φ}_n\textbf{)]}$$

Before simplifying these equations, it is important to understand the meaning of the subscript notation. Equations (1-3) and (1-4) say "If we have a vector V_n expressed in terms of its XY components, X_nY_n, the angle we will rotate V_n by (to get the next vector V_{n+1}) is $φ_n$. And, after rotation by $φ_n$, the new V_{n+1} expressed in terms of its XY components is X_{n+1} and Y_{n+1}". Note that we could then derive X_{n+2} and Y_{n+2} by simply replacing X_n and Y_n in equations (3) and (4) with X_{n+1} and Y_{n+1}. Thus, equations (1-3) and 1-4) give us the needed means of iteration.

Since it is desireable to make equations (1-3) and (1-4) as easy as possible to calculate quickly, one of the first things we need to do is to 'get rid of' the trig functions **cos(φ)** and **tan(φ)**. If we can't do this then we will have gone 'from the frying pan into the fire'. After all, what good is an algorithm to calculate trig functions if it itself requires us to calculate trig functions!

First let's tackle the tangent. Remember that to implement the basic Cordic scheme, we need an 'appropriately', diminishing series of fixed angles that we can store in a table. In the previous simplified explanation of the Cordic, we used the binary series 45°, 22.5°, 11.25°, etc. **But, it is not necessary to use that particular series.** There are many other angle series that will also work, so let's examine what attributes the φ series needs to have in order to be useable.

1.4 Choosing the Angle Series

For each iteration, the Cordic rotates its vector by the 'next' angle in the series. The only decision that the algorithm has to make is which direction to rotate, CW or CCW. In other words the decision is not whether or not to rotate but rather in which **direction** to rotate. As stated earlier, we always rotate 'toward' the input angle θ and this gives rise to something like matching ϕ_S to θ by successive approximation. ϕ_S may at times overshoot θ and at other times may undershoot θ. But as ϕ_S 'oscillates' around θ, it will do so with ever decreasing error amplitude. Since each iteration can only go one of two ways (CW or CCW rotation), the technique is not totally dissimilar to a binary search, deriving one bit at a time. However, there is one notable difference. Unlike the usual binary search which terminates as soon as the target is found, the Cordic always performs a fixed number of iterations, **even if a closer or even an 'exact' match is found earlier.** The reason for this will be discussed later but, as a result, it's quite possible that ϕ_S may 'match' θ early on and then move away from it (when the 'next' rotation is made), never to return quite as close again! However, with a properly chosen ϕ sequence, the 'final' approximation of θ by ϕ_S will always be at least as close to θ as the size of the last angle in the ϕ series.

Since the error in matching ϕ_S to θ will obviously contribute to the error in calculating the Sine and Cosine of θ, one requirement of the ϕ series is that the last angle in it should be as small or smaller than the allowable error in the sin/cos calculation. Further, since only a single 2-way decision is made per iteration, the angle series must not decrease at a rate faster than the binary series. In other words, the sum of the entire ϕ series must be able to 'reach' the largest input angle θ within $\pm$ the size of the smallest angle of the series.. As usually implemented, the Cordic handles an input angle range of $-90° \leq \theta \leq +90°$. So for example, if $\phi_0 = 45°$, ϕ_1 must be at least 22.5°, and in general, ϕ_n must be $\geq \phi_{n-1}/2$. The simple binary series of course satisfies these conditions but so also would a series like $\phi_n = \phi_{n-1} \cdot p$ where $0.5 \leq p < 1$. For example for $p = 2/3$, the series would look like 45°, 30°, 20°, 13.33°, etc. And, this series will work but, it will take a larger table and more iterations compared to the binary ($p = 0.5$) series for a given precision. It should also be mentioned that the series need not use a fixed geometric progression. For example, we could use something like 45°, 23°, 12°, 8°, 6°, etc. Just as long as each angle is **less than** its predecessor but **not less than one-half** of its predecessor. However, when each angle is exactly half its predecesser ($p = 0.5$) it will result in the shortest series for a given precision and input angle range. So unless there is good reason, we should choose the binary series or something very close to it.

1.5 Dealing With Tan(ϕ)

Now, we're going to consider an angle series that is quite close to the binary series but also has some very 'special' properties that will allow us to make the tangent calculation in equations (1-3) and (1-4) a trivial operation. Suppose we use the arctangent series $\mathbf{tan^{-1}(2^{-n})}$. Thus, for n = 0, 1, 2, 3, etc the series would be: $\tan^{-1}(2^{-0})$, $\tan^{-1}(2^{-1})$, $\tan^{-1}(2^{-2})$, $\tan^{-1}(2^{-3})$, etc. This is simply the series of angles whose tangents are the binary series 1.0, 0.5, 0.25, 0.125, etc. If we express the corresponding angles in degrees this series would be 45°, 26.57°, 14.04°, 7.13°, etc. Note that each successive angle is just a little more than one-half its predecessor. For example the ratio of $\phi_1/\phi_0 = 0.59$, the ratio of $\phi_2/\phi_1 = 0.528$, the ratio of $\phi_3/\phi_2 = 0.507$, etc. While p is not constant, it is rapidly approaching 0.5. Thus, as the angle gets smaller and smaller, the series begins to approach the binary series in that the 'next' angle will be almost exactly half of its predecessor (but never less). Clearly this series will satisfy the needed requirements that have been outlined in the previous section and, since it's also close to the binary series, it should be nearly as efficient in terms of table size (for a given precision and input angle range). So, **it will work** and, be **almost as good** as the binary series but, what does it do for us? First, let's rewrite equations (1-3) and (1-4) by replacing the angle ϕ_n with $\tan^{-1}(2^{-n})$ as shown in equations (1-5) and (1-6)

$$\textbf{(1-5)}\quad \mathbf{x_{n+1} = \cos(\phi_n)[x_n - y_n \cdot \tan(\tan^{-1}(2^{-n}))] = \cos(\phi_n)[x_n - y_n \cdot 2^{-n}]}$$
$$\textbf{(1-6)}\quad \mathbf{y_{n+1} = \cos(\phi_n)[y_n + x_n \cdot \tan(\tan^{-1}(2^{-n}))] = \cos(\phi_n)[y_n + x_n \cdot 2^{-n}]}$$

Notice that the multiplications by tan(ϕ) in equations (1-3) and (1-4) have been replaced by multiplication with a simple power of 2, which amounts to nothing more than right shifting y_n and x_n by n bits. But, before we can be totally rid of tan(ϕ), we need to discuss the roll of the sign of tan(ϕ) in equations (1-3) and (1-4). When we rotate CCW (considered a positive angle change), tan(ϕ) is positive. However, when we rotate CW (considered a negative angle change), $\phi < 0$ and since tan($-\phi$) = -tan(ϕ), clockwise rotation changes the sign of the tangent term. One way to take this into account is to introduce a 'direction' factor $\mathbf{d_n}$ as shown in equations (1-7) and (1-8) where $\mathbf{d_n}$ is defined such that $\mathbf{d_n}$ = +1 when we will rotate CCW, and $\mathbf{d_n}$ = -1 when we will rotate CW.

$$\textbf{(1-7)} \quad \mathbf{x_{n+1} = cos(\phi_n)[x_n - d_n{\cdot}y_n{\cdot}2^{-n}]}$$
$$\textbf{(1-8)} \quad \mathbf{y_{n+1} = cos(\phi_n)[y_n + d_n{\cdot}x_n{\cdot}2^{-n}]}$$

1.6 Dealing With Cos(ϕ)

Now let's discuss what we can do about the multiplication by cos(ϕ). Since both X_{n+1} and Y_{n+1} are multiplied by cos(ϕ_n), we could drop this factor with the only consequence being that the new rotated vector will be longer by a factor of 1/cos(ϕ_n). To illustrate this, let's work with just the bracketed parts of equations (1-7) and (1-8). If we start with $x_0 = xx_0 = \mathbf{V_0}$ and $y_0 = yy_0 = 0$, then $xx_1 = yy_1 = \mathbf{V_0}$ as shown in equations (1-9) and (1-10). Here we use xx and yy to denote the part of x and y due to the bracketed factor only (with the cosine factor removed). This first CCW rotation is illustrated in **Figure 1-3**. Note that after the 45° rotation, $\mathbf{VV_1}$ is now longer than $\mathbf{V_0}$. This is because it is the hypotenuse of a right triangle with both sides equal to $\mathbf{V_0}$. Thus, $\mathbf{VV_1} = \sqrt{2}{\cdot}\mathbf{V_0}$. Now if we were to multiply $\mathbf{VV_1}$ by cos(45°) = 1/$\sqrt{2}$, $\mathbf{VV_1}$ would then be exactly equal to $\mathbf{V_0}$ in length (shown in **Figure 1-3** as $\mathbf{V_1}$). So leaving out the cosine factor has caused $\mathbf{V}$ to grow longer (when rotated) by a factor of 1/cos(ϕ_0). Note also in figure 3 that x_1 and y_1 would be shorter than xx_1 and yy_1 by the same factor of cos(ϕ_0).

$$\textbf{(1-9)} \quad \mathbf{xx_1 = [xx_0 - yy_0] = [V_0 - 0] = V_0}$$
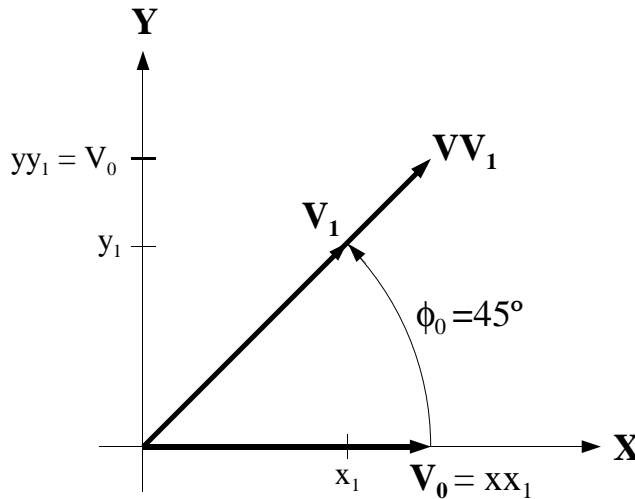$$\textbf{(1-10)} \quad \mathbf{yy_1 = [yy_0 + xx_0] = [0 + {\cdot}V_0] = V_0}$$

Figure 1-3

Now, let's rotate VV_1 CCW by ϕ_1 using just the bracketed components again. The calculation is given in equations (1-11) and (1-12) and graphed in **Figure 1-4**. From **Figure 1-4** we can see that VV_1 has grown again to VV_2. In fact, the ratio of VV_2/VV_1 is precisely $1/\cos(\phi_1)$. And, since VV_1/V_0 is precisely $1/\cos(\phi_0)$ we can easily see the pattern developing. Thus, we can do all our rotations using just the bracketed part of equations (1-7) and (1-8), as shown in equations (1-13) and (1-14). Then after any number of rotations **n**, we can correct for the increase in vector length by multiplying by $K_n = \cos(\phi_0) \cdot \cos(\phi_1) \cdot \cos(\phi_2) \cdot ... \cos(\phi_n)$ as depicted in equations (1-15) and (1-16).

$$\textbf{(1-11)} \quad \textbf{xx}_2 = [\textbf{xx}_1 - \textbf{yy}_1/2] = [\textbf{V}_0 - \textbf{V}_0/2] = \textbf{0.5V}_0$$
$$\textbf{(1-12)} \quad \textbf{yy}_2 = [\textbf{yy}_1 + \textbf{xx}_1/2] = [\textbf{V}_0 + \cdot \textbf{V}_0/2] = \textbf{1.5V}_0$$
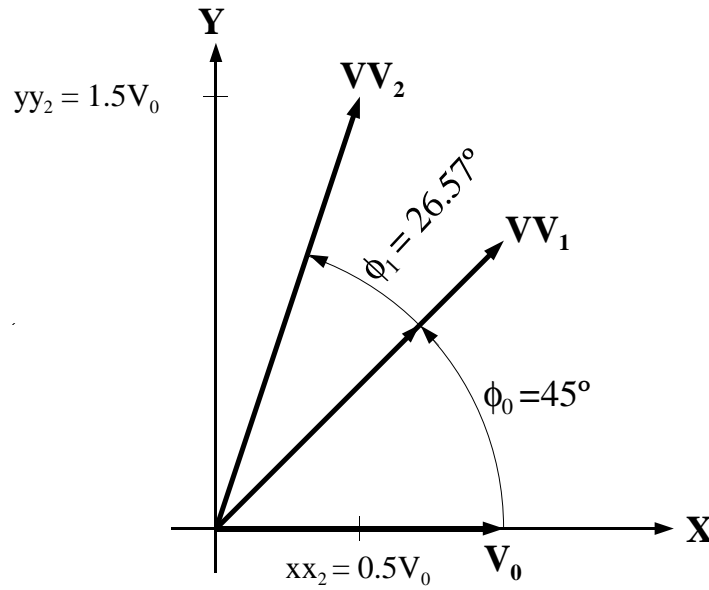
Figure 1-4

$$\textbf{(1-13)} \quad \textbf{xx}_{n+1} = [\textbf{xx}_n - \textbf{d}_n \cdot \textbf{yy}_n \cdot \textbf{2}^{-n}]$$
$$\textbf{(1-14)} \quad \textbf{yy}_{n+1} = [\textbf{yy}_n + \textbf{d}_n \cdot \textbf{xx}_n \cdot \textbf{2}^{-n}]$$

$$\textbf{(1-15)} \quad \textbf{x}_{n+1} = \textbf{K}_n \cdot \textbf{xx}_{n+1}$$
$$\textbf{(1-16)} \quad \textbf{y}_{n+1} = \textbf{K}_n \cdot \textbf{yy}_{n+1}$$

K_n is the 'chained' product of the cosines of each rotation angle up through the **n**th angle. Since the angle series is known in advance, each of these cosines is a constant that can be determined in advance. However one possible 'fly in the ointment' could be the signs of these constants because as we iterate to match ϕ_S to θ, we don't know **in advance** which steps will require CW and which steps will require CCW rotation. But, most fortunately, $\cos(-\phi) = \cos(\phi)$ and therefore all the cosines are positive and independent of d_n. This means that the product of all the cosines is positive and is also a pre-determined constant, dependent only on the **number** of rotations and **not on the rotation direction pattern**.

1.7 More About K

Another interesting property of K_n is that its value is asymptotic. As the number of iterations increase and the rotation angle gets smaller and smaller, the cosine approaches 1.0 in value. Since successive multiplication by 1 has no effect on the product, the value of K_n no longer changes when more iterations are made. To see this, refer to **Table 1-1** where the values for K_0 through K_{15} are shown to 9 significant digits. Note that ϕ_{14} and ϕ_{15} are the same. Thus, **K** has already reached its asymptote (to 9-digit accuracy) after 15 or more iterations.

ϕ_0	ϕ_1	ϕ_2	ϕ_3
0.707106781	**0.632455532**	**0.613571991**	**0.608833912**
ϕ_4	ϕ_5	ϕ_6	ϕ_7
0.607648256	**0.607351770**	**0.607277644**	**0.607259112**
ϕ_8	ϕ_9	ϕ_{10}	ϕ_{11}
0.607254479	**0.607253321**	**0.607253031**	**0.607252959**
ϕ_{12}	ϕ_{13}	ϕ_{14}	ϕ_{15}
0. 607252941	**0,607252936**	**0.607252935**	**0.607252935**

Table 1-1
The first 16 K_n Values
(with 9-Digit Accuracy)

1.8 The Bottom Line

We can now express the Cordic iteration formulae in their final simplified form as equations (1-17) and (1-18). Note that we have also simplified the notation by renaming xx and yy as simply x and y. But it needs to be clearly understood that the x and y values in equations (1-17) and (1-18) are not the same as those in equations (1-7) and (1-8). Rather, x and y now represent the values due only to the bracketed part of equations (1-7) and (1-8) with the cosine factor removed (what we have been calling xx and yy). Once we decide on the precision we need and can thus determine the total number of rotations **N** that we will use, we can use (1-17) and (1-18) iteratively to rotate through all **N** angles (realizing that the rotating vector is increasing in size with each rotation). Then, after **N** rotations have been been completed, we can multiply x_N and y_N by K_N to restore the original unit vector length.

$$\textbf{(1-17)} \quad \mathbf{x_{n+1} = [x_n - d_n {\cdot} y_n {\cdot} 2^{-n}]}$$
$$\textbf{(1-18)} \quad \mathbf{y_{n+1} = [y_n + d_n {\cdot} x_n {\cdot} 2^{-n}]}$$

Alternatively, we can even avoid those final multiplies by K_N by considering the following. The reason we multiply by K_N after all the iterations is because the vector length grows by a factor of $1/K_N$ as we perform the **N** rotations. So, instead of starting with a unity-length vector, we can start with an initial vector length of K_N. Then when it grows by a factor of $1/K_N$ after the Nth rotation, the vector will have a length of **one**. And, of course, this is precisely what we want, a unity-length vector, so that $\mathbf{x_N = cos(\theta)}$ and $\mathbf{y_N = sin(\theta)}$.

We're now about ready to 'put it all together' and sketch a flow chart that can serve as a General Model for implementing the Cordic Algorithm (as it would be used to calculate the Sine and Cosine of an input angle). Once we refine our model flow chart a bit, we'll begin the process of adapting the design to the specifics of the KSP.

1.9 The Cordic Models for SinCos

Figure 1-5 is a flow chart depicting the general logic needed to implement the Cordic Algorithm for calculating the Sine and Cosine of an input angle. As shown it can handle any input angle from $-90° \leq$ theta $\leq +90°$. However, before 'walking through' this flow chart, let's discuss the associated data elements and some of the algorithmic side issues. Whenever we refer to angles, we will use degrees ($90° = \pi/2$ radians = one right angle). However, as we will see later, when adapting the algorithm to the KSP, **the Cordic can be implemented with whatever angular unit system is most suitable**.

As to constants, we need to set up an array of arctangents. This array, refered to as **Atan**[n] in the flow chart, is a zero-based array where **Atan**[0] $= \tan^{-1}(2^{-0}) = \tan^{-1}(1) = 45°$. We can assume that we also have chosen a value for **N** (the total number of iterations that we will use). Our choice of **N** sets the size of the **Atan** array which runs from **Atan**[0] to **Atan**[**N**-1]**.** Like **Atan**[0] $= 45°$**,** the rest of the table elements are also pre-computed using the formula **Atan**[n] $= \tan^{-1}(2^{-n})$ where n = 1 to n = **N**-1. And, since we know **N**, we can also precompute the constant $\mathbf{K_N}$ (simply refered to as **K** in the flow chart).

As to variables, **X** and **Y** have their usual meanings as the XY components of the current vector. The variable **d** is the direction factor which is set to either **+1** (CCW rotation) or **-1** (CW rotation). The variables dx and dy are temporaries that hold the computed 'change in **X**' and 'change in **Y**' for the next rotation. And finally, we have the angle accumulation variable **Z**. We use this variable to decide in which direction to rotate. Remember, we always want to rotate 'toward' the input angle (denoted by the input variable named **'theta'**. So if the accumulated rotation angle, ϕ_S (from our former notation), is greater than **theta** we would rotate CW and if ϕ_S is less than **theta** we want to rotate CCW. We can accomplish the same thing by simply initializing **Z** = **theta** and then choosing the rotation direction by always trying to 'drive' **Z** toward zero. Thus, when **Z** ≥ 0, we want to rotate CCW so we would set **d** = **+1**. Conversely, when **Z** < 0, we want to rotate CW so we would set **d** = **-1**.
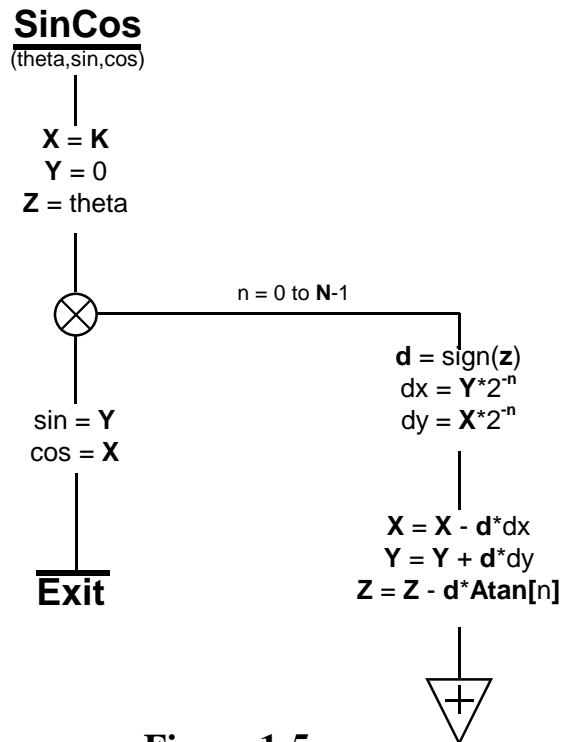

Figure 1-5
Cordic Model #1 for SinCos

Now let's 'walk through' the flow chart (**Figure 1-5**). Before entering the iteration loop, we set our initial vector to a length **K** at an angle of zero (ie **X = K, Y** = 0) so the vector lies on the X-Axis. We also initialize the angle rotation accumulator **Z** to the input **theta** (whose Sine and Cosine we want to calculate). We then execute the loop body **N** times with n = 0, n = 1, ... n = **N**-1.

In the loop body itself, we first determine the direction we need to rotate by examining the sign of **Z** and setting **d** accordingly (ie **d** = +1 if **Z** ≥ 0 and **d** = -1 if **Z** < 0). Next, the loop body computes 'dx' (the change in **X**) by shifting **Y** to the right by 'n' places. Since **Y** can be either positive or negative at times, the right shift must be an arithmetic right shift (where the sign bit of the number is shifted in on the left). If only a logical right shift is available (where a zero is always shifted in on the left), a special subroutine will have to be written to emulate an arithmetic right shift. Next, 'dy '(the change in **Y**) is computed by arithmetic-shifting **X** to the right by 'n' places. Then, the actual **XY** transformation for the rotation is performed by subtracting **d***dx from **X** and adding **d***dy to **Y.** Finally, the angle accumulator **Z** is reduced by **d*****Atan**[n]. Effectively, the angle **Atan**[n] is subtracted from **Z** if the rotation was CCW and it is added to **Z** if the rotation was CW. Then, after all N iterations are completed, the value of **Y** is returned as Sin(**theta**) and the value of **X** is returned as the Cos(**theta**).

The flowchart of **Figure 1-5** thus implements the Cordic **SinCos** in a fairly simple and straight-forward way. However, it's not necessarily the fastest nor most efficient implementation. Of course an optimum implementation will be at least somewhat dependent on the target language and/or hardware platform, but, we can make some generalizations about what is likely. Usage of the directional factor **d** has a few drawbacks. First it has to be set (by examining **Z**), and then it has to be used in 3 multiplications. While multiplication by +1 or -1 may seem trivial, many implementations may actually have to go through a full multiply cycle to accomplish it. If so, adding 3 multiplies to the loop body may impose a noticeable execution time increase relative to the rest of the basic operations done in the loop body. For example, the KSP has no **exclusive-or** operation and therefore about the only efficient way to utilize **d** is to actually multiply it by dx, dy, and **Atan**. Obtaining **d** from **Z** may also be cumbersome or slow. For the KSP it can be done several ways with one line of code such as [**d := sh_right(Z,31) .or. 1**] but it doesn't execute too fast (worse yet would be something like [**d** := **Z/abs(Z)**]. So, for most implementations, the flow chart logic of **Figure 1-6** will probably execute faster

SinCos
(theta,sin,cos)

X = **K**
Y = 0
Z = theta

n = 0 to **N**-1

$dx = Y*2^{-n}$
$dy = X*2^{-n}$

sin = **Y**
cos = **X**

Z < 0

X = **X** + dx
Y = **Y** - dy
Z = **Z** + **Atan**[n]

X = **X** - dx
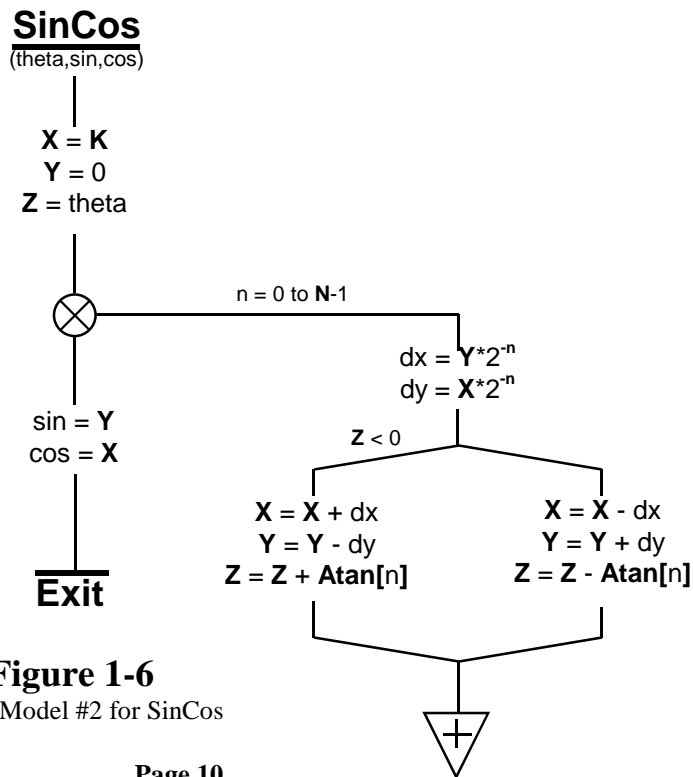Y = **Y** + dy
Z = **Z** - **Atan**[n]

Exit

Figure 1-6
Cordic Model #2 for SinCos

With the 2nd Cordic Model of **Figure 1-6**, we eliminate the **d** factor. So the loop body starts with the calculations of dx and dy and then **Z** is tested against zero. If **Z** < 0 the signs are reversed for the 3 remaining assignments. While this does require us to essentially repeat 3 lines of code with only the signs changed, it will definitely execute faster because only one set of 3 lines executes per loop pass and almost for sure a simple if-else decision will execute faster than acquiring **d** and then multiplying it by 3 values.

Another thing we can do to speed up this algorithm is to reduce the angle range. As implemented so far, the input angle can range from -90° ≤ **theta** ≤ +90°. To cover the full angle range, an outer 'shell' routine can be used to reduce and/or reflect the input angle. However, such a shell routine can easily reduce any input angle to a range of 0° to 90°. For many applications, only first quadrant angles need be accomodated anyway and, when the input angle range needs to be greater than that, we can use a shell routine. So, if we restrict the input angle range from 0° to 90°, there is very little downside but, there is some upside. If the input angle is known to be restricted to the 1st quadrant (0..90°), we can essentially save one loop iteration by doing a 'pre-rotation' of 45°, see **Figure 1-7**.

Figure 1-7
Cordic Model #3 for SinCos
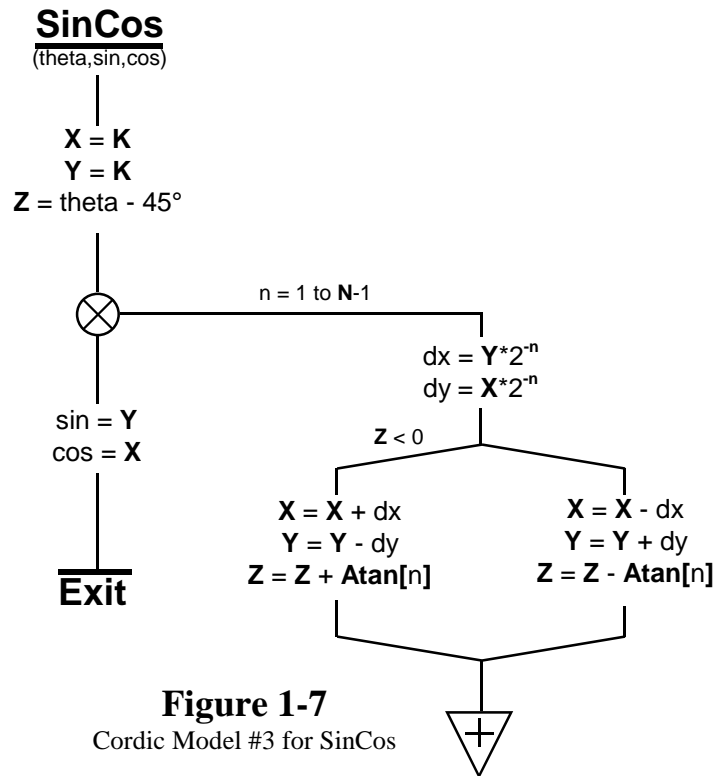
When we know that the input angle is positive (in the 1st quadrant) then we know that the first rotation will be CCW by 45°. In other words, if we execute the flowchart of **Figure 1-6** for only the first loop pass (n = 0), initially $X_0 = K$ and $Y_0 = 0$ and after the first rotation, $X_1 = Y_1 = K$ (review **Figure 1-3**). Also, the angle accumulator **Z** will be reduced by 45°, so after the first (n = 0) loop pass, **Z** = **theta** - 45°. So, in our 3rd Cordic Model of **Figure 1-7**, we simply initialize both **X** and **Y** to **K** and subtract 45° from the input angle before executing the loop. Then, because we have essentially accomplished the n = 0 pass, we only need execute the loop with n = 1 through n = **N** -1.

If the platform language supports non-zero-based arrays, we can shorten the array by eliminating **Atan**[0]**.** If the language only supports zero based arrays we can either include the **Atan**[0] value in the array and just not use it, or, we can shift all the arctangents so that **Atan**[0] = tan-1(1/2) = 26.57° . If we do that, we have to access the **Atan** array with n - 1 instead of n. However, this will require an extra subtraction in the loop body just to save one array element which is probably not a worthwhile trade.

1.10 Early Exit Considerations

Earlier it was stated that normally we don't shorten the number of iterations the Cordic performs, even when the cumulative angle ϕ_S 'hits' the input angle θ early on. Now we can take a look at why we don't try to use early termination. If we want to exit early when **Z** gets close enough to θ, we will then need to keep an array of $\mathbf{K_n}$ values since $\mathbf{K_n}$ depends on the number of iterations.. Since we will never know in advance how many iterations it might take before **Z** meets the exit criteria, we can't initialize the rotation vector length to **K** as we have been doing (because we don't know **K**'s value until **after** we know the number of iterations before the early exit. Therefore, we need to start with an initial vector length of one and when the iteration loop exits, we then multiply both **X** and **Y** by **K**[n]. Besides the time cost of the multiplies, there may be scaling problems depending on the capabilities of the platform language. For example, if we have only 32-bit integer arithmetic (as with the KSP), we can't scale **XY** to full precision if we need to leave some headroom for the final multiply by a similarly-scaled **K[n]**. So as a list, here are the disadvantages of trying to test for an early exit.

1,   We need to store an array of **N, K_n** values.
2.   The loop execution time is burdened with an additional test of **Z** against some error value.
3.   We need to perform two additional multiplies after loop exit.
4.   We need to settle for less precision than would otherwise be attainable
         because of headroom for scaling issues in accomodating the $\mathbf{K_n}$ multiplies.
5.   A noticeable increase in the number of lines of code required.

When we run an analysis of the execution time savings (if any) that we obtain with all this extra baggage, it turns out to be statistically small at best. Of the full input angle range, only a small percentage of these angles will actually result in an early exit. On balance, implementing an early-exit scheme just doesn't seem worth the price so you will seldom see the Cordic (for Trig functions) implemented with an early exit.

1.11 Adapting the Cordic to the KSP

Since the KSP has only one number type, namely 32-bit signed integers, we first need to make some choices (both internal and external) as to scaling. There are basically two issues that need to be decided, the calculation precision needed and, the scaling to be used for input and output. These two issues are somewhat interrelated. For example, if you ask that the output be scaled by 10,000 and you only calculate to a precision of one or two-digits, you might be better off not scaling the output so much. The bigger you scale the output, the less 'headroom' you will have for using the result in additonal calculations. So, generally, you don't want to scale the output by much more than enough to 'cover' the precision you are calculating to. On the other hand, there is little point in calculating to a much higher precision than you can deliver with your output scaling. Although if it costs nothing, it may be adviseable to calculate to higher precision because then you could increase output precision later (if you need it) by simply re-scaling the output. So, one way to design the **SinCos** routine for the KSP is to select an output scaling that seems to strike a good balance between the likely-to-be-needed accuracy and headroom. Then, implement the routine to calculate the highest possible accuracy that can be done without cost (or at least inexpensively). But of course, shoot for at least a comparable accuracy to what is needed by the output scaling choice.

Considering what is likely to be the typical uses for Sines and Cosines, for the **KSP Math Library**, it was decided to use an output scaling of 10,000 for the **SinCos** function. The input angle scaling is kind of a mixed bag. One of the most likely uses for the **SinCos** function will be equal-power crossfading, perhaps for adjacent pitches (as in a glider type application) or adjacent velocity layers in a Mod-wheel controlled velocity situation. If a simple 0..127 type CC is involved, the resolution will only be 128 steps per 90°. On the other hand, if the Pitch Wheel is involved there could be up to 16,384 steps of resolution available (depending on the PW hardware). In any case, an input angle unit that is an even power of two might be most appropriate for direct MIDI control.

In the case of adjacent pitch crossfading, the KSP unit is in millicents but in the case of velocity crossfading the KSP unit is the MIDI unit range 0..127. If you want the angle range to 'track' a CC you may want to use a power of 2 range. If you want the angle range to track a pitch, you may want to use a power of ten. As stated earlier, the Cordic can be implemented with any desired type of angular unit (and fractions thereof), so one possibility is to choose a binary power such as 128, 512, or 1024. Another possibility, because of its familiarity, would be to use degrees or tenths of degrees for an angle range of 0..900. No matter which scale is chosen, it will probably not be right for all situations so a simple power of ten was chosen instead as a compromise. There is an angular unit called the grad where 100 grads = 90°. This however seems a bit course for angular resolution so the deci-grad was chosen. This means that the 1st quadrant angular range is covered by 0..1000 deci-grads (or dg). If you need to cover a range of say 0..127, you can of course simply scale it up to 1000 with some preliminary calculation such as CC*1000/127. Or, alternatively, you can change the **Atan** table by re-scaling all the values appropriately to change the angular unit. Also, if you need more angular resolution, you can alter the internal scaling as will be seen later on.

Having decided to use an angular unit of deci-grads (0..1000 range) and an output scaling of 10,000 for the Sine and Cosine, we can turn our attention to scaling within the function itself. For a unity-length vector, the value of **X** or **Y** cannot exceed that of the sine or cosine. And, since the vector begins at **K** (which is less than unity) and increases to a max of unity by the last rotation, **X** and **Y** cannot exceed 1.0 in value. Since 32-bit signed integers can represent values somewhat higher than $2 \cdot 10^9$, we can scale **X**, **Y**, and **K** by that much if desired. But of course 10^9 will be more human-friendly and probably still quite adequate. So I scaled **X**, **Y** and **K** by 10^9. **K** needs to have the same scaling as **X** and **Y** because **X** and **Y** are initialized to the value of **K**. Since we have restricted the angle range to the first quadrant, **X** and **Y** never **end up negative** but either can become somewhat negative **during the rotations** that reduce **Z** to zero. This is no problem with **X** and **Y** itself because 32-bit signed integers can also handle large negative values. However, as mentioned earlier, when right shifting **X** or **Y** to calculate dx or dy we must be sure to do **arithmetic shifts**. Fortunately, **this is** what the KSP **sh_right** function does so we don't have to do anything special here.

The angle accumulator starts out at the input 'angle' value, 0..1000 dg and then oscillates its way down to zero. So its maximum range is from about -500 to 1000. So as long as we use dg for our angle unit, we can scale **Z** by up to $2 \cdot 10^6$ without running into overflow problems. For human friendliness we will scale **Z** by 10^6. Thus, the table of arctangent values must also be scaled by 10^6 so they can be added or subtracted from **Z** as we rotate. If the angular unit is changed, this scaling may also have to be changed. For example, to change the angular unit to centi-grads (ie an input range of 0..10000), we would change the scaling to 10^5. In general, the maximum angle input value times the scale must not exceed 2,147,483,647 (the maximum positive integer).

Finally, we need to determine how many iterations we need to use. If we used the binary series for an angle series, we could expect to get about one bit of precision per loop pass. However, because the binary tangent series results in an angle series not quite as efficient as the binary series itself, we can expect something less than a one bit per iteration resolution. Since we have chosen an output scale of 10^4, we want at least that much precision from our calculation. With these thoughts in mind, the number of iterations was set at 16 which gives us a precision of about one part per 65,000. However, as stated earlier we shouldn't expect to do quite that well for various reasons, but, the calculation precision ought to be at least several times better than the output scaling resolution.

Now take a look at the code for the **SinCos** routine as implemented for the KSP. The Cordic constant **K** has been named **cK** which is defined as 10^9 times the value that was shown in **Table 1-1** for ϕ_{14} and ϕ_{15}. The arctangent array is defined as **AngTbl[16]** but **AngTbl[0]** isn't actually used. It just serves as a placeholder so the loop index can be used without decrementing 'n' when accessing the table as discussed previously. Note that the **AngTbl** array is declared indirectly with the Data Function named **BuildAngleTable**.

Now, let's walk through the code for **SinCos**(ang,sin,cos). We begin by initializing local variables **X** and **Y** to the constant value of **cK.** Then, we subtract **Ang45** = 500 dg from the input angle parameter and scale it by 10^6 (since the **AngTbl** is scaled by 10^6). Thus, we have initilized **X**, **Y**, and **Z** to where they would be after an n = 0 rotation. Now we execute the 'for-loop' with n = 1 to n = 15 which will give us 15 more rotations for a total of 16 rotations. Within the for-loop we compute dx and dy by right shifting **Y** and **X** by n bits. We then 'apply' these changes in accordance with the desired rotation direction (determined by the sign of **Z**). We also update **Z** itself by adding or subtracting the angle **AngTbl[n]**. When we exit the loop we have the Sine and Cosine in **Y** and **X** respectively but, they're scaled by 10^9 rather than by 10^4 as chosen for output scaling. So, we divide **X** and **Y** by 10^5 and round the results to provide the final **sin** and **cos** outputs.

1.12 SinCos Error Analysis

Table 1-2 shows the results of running a Delphi error analysis of the **SinCos** routine. Results for the Sine and Cosine are essentially the same so only the Sine data is shown. The data was obtained by writing a Pascal emulation of the **SinCos** routine using data types and instructions that behave in a manner identical to their counterparts in the KSP (except for execution time of course). Where identical instructions didn't exist, they were synthesized. For example, Pascal has only a logical right shift available so a subroutine was written to emulate the KSP's arithmetic right shift. Finally, an 'outer' analysis program was written to invoke the **SinCos** emulation with the full range of input angles from 0 to 1000 dg. For each input value, the **SinCos** output was compared with a very accurate value (15 - 16 significant digits) from the Delphi Library, and, error statistics were accumulated and then summarized in tabular form as shown in **Table 1-2** below.

	Top 5 +/-	**Absolute Error**		**Relative Error**	
	#1 +	+7.95E-5	@Sin(23)	+1.86%	@Sin(2)
	#2 +	+7.69E-5	@Sin(112)	+1.86%	@Sin(1)
	#3 +	+7.55E-5	@Sin(65)	+0.587%	@Sin(5)
	#4 +	+7.44E-5	@Sin(42)	+0.27%	@Sin(8)
Table 1-2	**#5 +**	+7.09E-5	@Sin(186)	+0.268%	@Sin(4)
SinCos Error Stats	**#1 –**	-7.54E-5	@Sin(31)	-0.263%	@Sin(3)
16 Iterations	**#2 –**	-7.51E-5	@Sin(155)	-0.261%	@Sin(6)
Output Scaling = 10,000	**#3 –**	-7.1E-5	@Sin(246)	-0.26%	@Sin(9)
	#4 –	-7.08E-5	@Sin(144)	-0.257%	@Sin(12)
	#5 –	-6.81E-5	@Sin(159)	-0.155%	@Sin(31)
	Error Summary Data				
	Avg Error:	+7.59E-7		+0.00476%	
	Avg |Error|:	**+2.66E-5**		**+0.0143%**	

In **Table 1-2**, as well as subsequent error tables that will be given, the 5 largest positive errors and 5 largest negative errors are listed first. This is followed by 2 lines of summary data.

The first column of data contains Absolute errors. For example, for a given input angle φ, this is merely the difference between sin(φ) as calculated by **SinCos** and the more accurate sin(φ) given by the Delphi library. Of course the **SinCos** value is first converted to a real number by dividing it by 10000. The **@Sin** data reports the input angle (in deci-grads) at which the error occured.

The 2nd column contains Relative error. This is the absolute error in sin(φ), as just described, divided by sin(φ) itself (and then multiplied by 100 percent). You have to be cautious when looking at Relative error, especially for the 'Top 5' errors, because it's easy to be misled. The larger relative errors usually occur when the function itself is near zero. Since the relative error calculation contains a division by the value of the function itself, when the function is at or near zero, the relative error is often *deceptively* large. For example, suppose we calculate sin(1). **SinCos** provides a result of $16/10000 = 0.0016$. An accurate sin(1) is about 0.001570796. Note that **SinCos** is 'correct' to its 4 decimal places. The Absolute error for sin(1) is thus about 0.0000292 and this is not a very large error. However, when we divide by 0.00157+ and multiply by 100, we get +1.86% which may seem kind of big.

Another problem with Relative error is when the function itself **is zero**. For example, **sin(0) = 0**, where the calculation of relative error boils down to **0/0**. In mathematics this is an indeterminant form and most modern floating point packages will report this as NAN (not a number). For this reason, the Relative Error data was taken with the input angle range set from 1..1000 dg (instead of 0..1000 dg as for the Absolute error data).

The Summary data is probably the most useful in telling us what is going on, error-wise, especially for relative error. The **Avg Error** is the algebraic sum of all the negative and postive errors while the **Avg |Error|** is the average of the error magnitudes. Perhaps the most meaningful *single* number in the summary is the absolute **Avg |Error|** because this number represents the average **error magnitude**, independent of sign. If the errors are balanced, the **Avg Error** should be near zero or at least a lot smaller than **Avg |Error|**. If **Avg Error** is near **Avg |Error|** can be an indication that there is a 'bias' in the errors. And, often times such an error bias can be removed, resulting in an overall error magnitude reduction.

Now, let's see what we can conclude from the data in **Table 1-2** for the KSP implementation of **SinCos**. Since we are only scaling our ouput value by 10^4, even if we compute the Sine perfectly, we would expect an output error of $5 \cdot 10^{-5}$ max with an **Avg |Error|** of $2.5 \cdot 10^{-5}$ due to quantization errror (remember we have a resolution of only $0.0001 = 10^{-4}$). Looking at the maximum absolute errors we see they are less than $8 \cdot 10^{-5}$ and our **Avg |Error|** is only $2.66 \cdot 10^{-5}$. This means the biggest part of the error is due to quantization. To verify this, I changed the output scaling to 10^5, and re-ran the error analysis, see **Table 1-3**. The max absolute error dropped below $3.1 \cdot 10^{-5}$ and the **Avg |Error|** dropped to $1.02 \cdot 10^{-5}$. On the other hand, increasing the output resolution by another factor of 10, see **Table 1-4**, made only a minor improvement. The error data obtained with an output scaling of 100,000 tracks well with the theoretical 1 part in 65,000 for 16 iterations. And, comparing **Tables 1-2, 1-3, and 1-4**, the data says that our calculation accuracy is better than our output resolution but not by a great margin. So, it might be just about where it should be.

If we keep the output scaling at 10^4 and use one less iteration, the max absolute error rises to about $9.5 \cdot 10^{-5}$ and the **Avg |Error|** rises to about $3 \cdot 10^{-5}$. Although computer-dependent, the **SinCos** routine typically executes in less than 9 μsecs. If you drop one iteration, this drops by about 0.6 μs. This isn't exactly an exciting speed improvement to trade for the poorer accuracy so it seems like 16 iterations (including the first 'free' one) is a fairly good choice for an output scaling of 10^4. For more information about the execution time of the KSP Math Library routines, see the benchmark info in Appendix A.

Top 5 +/-	Absolute Error		Relative Error	
#1 +	+3.09E-5	@Sin(50)	+0.586%	@Sin(2)
#2 +	+3.01E-5	@Sin(194)	+0.163%	@Sin(6)
#3 +	+2.97E-5	@Sin(88)	+0.128%	@Sin(11)
#4 +	+2.96E-5	@Sin(218)	+0.0982%	@Sin(19)
#5 +	+2.95E-5	@Sin(23)	+0.0859%	@Sin(15)
#1 –	-3.38E-5	@Sin(46)	-0.687%	@Sin(3)
#2 –	-3.24E-5	@Sin(3)	-0.231%	@Sin(7)
#3 –	-3.13E-5	@Sin(54)	-0.209%	@Sin(4)
#4 –	-3.11E-5	@Sin(308)	-0.151%	@Sin(12)
#5 –	-3.1E-5	@Sin(246)	-0.08%	@Sin(16)
Error Summary Data				
Avg Error:		-1.51E-7		-0.000242%
Avg |Error|:		**+1.02E-5**		**+0.00625%**

Table 1-3
SinCos Error Stats
16 Iterations
Output Scaling = 100,000

Top 5 +/-	Absolute Error		Relative Error	
#1 +	+3.05E-5	@Sin(23)	+0.459%	@Sin(2)
#2 +	+2.99E-5	@Sin(50)	+0.195%	@Sin(6)
#3 +	+2.89E-5	@Sin(112)	+0.116%	@Sin(11)
#4 +	+2.87E-5	@Sin(88)	+0.0986%	@Sin(15)
#5 +	+2.84E-5	@Sin(42)	+0.0915%	@Sin(19)
#1 –	-3.04E-5	@Sin(3)	-0.645%	@Sin(3)
#2 –	-3E-5	@Sin(93)	-0.249%	@Sin(7
#3 –	-2.94E-5	@Sin(175)	-0.225%	@Sin(4)
#4 –	-2.91E-5	@Sin(155)	-0.13%	@Sin(12)
#5 –	-2.9E-5	@Sin(73)	-0.114%	@Sin(1)
Error Summary Data				
Avg Error:		+1.45E-8		-0.000453%
Avg |Error|:		**+9.76E-6**		**+0.00611%**

Table 1-4
SinCos Error Stats
16 Iterations
Output Scaling = 1,000,000

So, returning to **Table 1-2**, let's see what else we can conclude. The **Avg Error** is about 100 times lower than the max absolute error and about 30 times lower than the **Avg |Error|**. Together, this tells us that the errors are pretty evenly distributed polarity-wise and that both polarities have about the same average amplitudes.

Now take a look at some of the Relative Error data. As pointed out earlier, the max errors are always larger than the Absolute Errors, sometimes deceptively so. Let's take a look at the worst one in **Table 1-2**, which is **sin(2)** (where the relative error is shown as +**1.86%**). The absolute error for **sin(2)** isn't among the **Top 5** positive errors so we can't read its value from **Table 1-2**. But, we can easily run our test program and ask the KSP **SinCos** routine what the sine of 2 dg is. The scaled output we get when we do that is 32, which in real numbers is **0.0032**. According to my HP Scientific calculator, **sin(2) = 0.003141587**. Thus the absolute error in **sin(2)** must be 0.000058 and our calculation (plus quantization) error in determining **sin(2)** is a little more than half-way between quantization steps. This of course is not really a very large error but the problem is that it must be divided by **sin(2)** and then multiplied by 100 (to express it in percent). Thus, for smaller angles like **sin(2)**, the absolute error is multiplied by a fairly big number, about **31,831** in the case of **sin(2)**. Because of this 'error magnification' for small angles, we can expect a rather non-uniform distribution of error amplitudes, with all the larger errors crowding toward the lower angles. Note that the overall Relative Error magnitude, ie **Avg |Error|** is only **0.0143%**. Whereas the max relative error (+1.86%) is more than 100 times that!

So, overall and considering the likely uses for the **SinCos** routine, the accuracy and output scaling will probably be a good fit. However, it should be noted that if you need more accuracy in your result, you can easily obtain it by some appropriate combination of increased output scaling and/or by increasing the number of iterations. The final limit of accuracy will of course be controlled by the precision of 32-bit integers themselves. However, remember that the **X, Y,** and **K** values are scaled by 10^9, so there is quite a ways to go before you'll bump into that limit.

As mentioned earlier, you might also have some application where you need more resolution in the input angle. As it stands, the input range is 0..1000 and therefore the input angle (along with the arctangent table) can be scaled by 10^6. If you 'open up' the angle resolution, you will have to reduce the angle scaling factor proportionally so that the maximum input angle times the scaling will stay under $2 \cdot 10^9$. If you need to increase the number of iterations, you will also have to extend the arctangent table. This is very easy to do because the last angle in the table at **AngTbl[15]** is **0.019428** deci-grads, or less than **0.00175°**. When angles get this small, the tangent of the angle is almost exactly equal to the angle itself (in radians). Therefore, no matter what unit the angle is in, if we cut the tangent in half (to continue the binary series of tangents), the angle will also be cut in half. So all you have to do to extend the angle table is to continue dividing the angle by a factor of two for each new table entry.

If you want to change to a different angular unit, this too is easy to do. All the data affecting angles resides in the Data Function named **BuildAngleTable**. Suppose you want to express the input angle in degrees with a resolution of one degree. Thus, the input angle range will be 0..90°. To make this change, edit the **SetAngleUnits** function as follows. First, change the constant named **Ang90** to a value of 90 (from its present value of 1000). Note that when **Ang90** is changed, all other dependent angle constants (such as **Ang45**) will automatically be changed. Now, since the max angle has decreased by more than a factor of 10, we can increase angle scaling from 10^6 to 10^7. So, change the value of the **AngScale** constant from 1,000,000 to 10,000,000. Now let's look at the arctangent table. The conversion factor to change from an angular measure of deci-grads to degrees is 90/1000 = 0.09. But, since we are also going to scale all angles by 10^7 now instead of 10^6, we need to multiply by another factor of ten. So, we simply multiply each **AngTbl** array value by **0.9** and round it to the closest integer.

For example, **AngTbl[3]** now is **79166848** (in deci-grads scaled by 10^6), so in degrees scaled by 10^7 it would be **71250163**. Once these changes are made to the **BuildAngleTable** function, the **SinCos** routine will calculate the Sine and Cosine of any input angle from 0 to 90° (with a one-degree resolution). **NOTE:** If you change **Ang90**, be sure the new value is an even number because the pre-loop operation has to rotate the vector by **Ang45** which should be exactly half of **Ang90**. Therefore you should not choose an odd subdivision of a right angle for your new angular unit.

1.13 Extended SinCos

The **SinCos** routine is designed to work only when the input angle is in the range of 0..1000 dg (0..90°). However, since the angle series is based on the binary series of tangents, the total of all the angles in the table somewhat exceeds 90°. Therefore, if the calling program might sometimes produce angles a little outside of this 0..1000 range, **SinCos** will still deliver the correct answer. This little bonus range is about ± 110 dg but generally, if your application uses larger angles, you can use the 'shell' routine named **XSinCos**. This shell routine extends the allowed angle range to the full range of a signed integer. **XSinCos** first reduces and/or reflects the input angle to its first-quadrant equivalent and then attaches the appropriate signs according to the actual quadrant the input angle falls in. This angle reduction is performed by the support routine named **ReduceAngle**.

1.14 Tangent Routines

Since **tan(ϕ) = sin(ϕ)/cos(ϕ)**, the Math Library can easily calculate the tangent of an angle also. Thus, the library has a pair of routines that use the **SinCos** routine to provide the tangent. The routine named **Tangent** returns the tangent for only a first-quadrant angle whereas the shell routine named **XTangent** allows any input angle that can be represented by a 32-bit signed integer. Unlike the Sine and Cosine which are continous functions with a magnitude between **0.0** and **1.0**, the tangent has 'infinite' discontinuities. For example, the tangent of 90° is infinite. Therefore, the value of tan(1000) cannot be represented with any real number. However, since tan(999) is about 637, (or 6,370,000 when scaled, this is the largest normal value apart from tan(1000) that can be output. So the **Tangent** routines denote the tan(1000) situation with an output of +**100,000,000** and similary output a value of **-100,000,000** for tan(-1000), or the reduceable equivalents of these angles. This output of ±**100,000,000** should be viewed more as a flag or sentinel rather than as an actual number, since the true value of tan(±1000) cannot be properly represented.

2.0 FAST LOG FUNCTIONS

2.1 Introduction

The Fast Logarithmic Functions of the **KSP Math Library** are based on a binary log approximation presented by **John DeVos** (circa 2001). This algorithm, as refined and adapted for the KSP, has surprisingly good accuracy (about 0.01%) and fast (about 2 µsecs) execution time, see Appendix A. The procedure is very compact, needs no lookup table, and like the Cordic, is well suited to the KSP's capabilities and usage.

2.2 Log Basics and Notation

By definition, the logarithm of **X** to the base **b** is the power to which **b** must be raised to obtain **X**. Mathematically, this is usually expressed by saying that if; $Y = \log_b(X)$, then $b^Y = X$. It is important to emphasize that **b** can be any number as long as **b** > 1. The most commonly used base for logarithms is **10**. In fact $\log_{10}(X)$ is usually called the 'common log' of **X** and is often written without the 10 subscript, for example $y = \log(X)$, is usually understood to mean that the base of the logarithm is 10. Another often-used base is a transcendental number (like π) denoted by the Greek letter ε (or often just a lower case **e**) which has a value of about **2.71828...** Logarithms to the base **e** are called 'natural' logarithms. The natural log of **X** can be written as $\log_e(X)$ but is usually written with the shorthand **ln(X)**. One fortunate property of logarithms is that if we can determine logs to any base **m**, we can easily convert them to any other base **n** by simply multiplying by a constant equal to $1/\log_m(n)$. So, if we choose to compute logs to the **base 2**, we can easily write some simple shell routines that will give us logs to the **base 10** or logs to the base **e** or whatever base we might desire.

2.3 Binary Logarithms

The rest of this presentation will concentrate on approximating the binary log function, $Y = \log_2(X)$. So, as a shorthand for $\log_2(X)$, we will use the notation **lg(X)** or **Lg(X)**. When the base is an integer, a logarithm is usually considered to be made up of two parts, an **integer** part called the **characteristic**, and, a **fractional** part called the **mantissa**. Thus, we can represent **Lg(X)** as shown in equation (2-1) where **Lc** stands for the **log characteristic** and **Lm** stands for the **log manitssa**.

$$(2\text{-}1)\;\; Lg(X) = Lc(X) + Lm(X)$$

For binary logs, **Lc(X)** is the closest, **integer power of two** that is equal or less than **X** in value. Thus for example, **Lc(12) = 3** because $2^3 = 8$ (which is less than 12) and $2^4 = 16$ (which is more than 12). So, the actual value of **Lg(12)** must be greater than 3 but less than 4. When **X** is exactly an integer power of 2, then **Lm(X) = 0**. For all other **X** (greater than one and not an exact integer power of 2), **0 < Lm(X) < 1**.

If **X** is expressed as a binary number, **Lc(X)** can be computed very easily. Numbering the binary bit positions of **X** in terms of their equivalent powers of two (ie LSB = 0), **Lc(X)** can be obtained by merely finding the highest 'on bit' position in the number **X**. For example, if we write the number 44 (base 10) as a 32-bit binary number we have:

$$(2\text{-}2)\;\; 44_{10} = 0000,0000,0000,0000,0000,0000,0010,1100_2$$

Since the number 44 only has ones in bit positions 5, 3 and 2, this makes 5 the highest on-bit position. Therefore, **Lc(44) = 5** and thus **Lg(44)** must lie between 5 and 6 (since **Lm(X) < 1**). Finding the highest on-bit is a simple matter of shifting and bit testing (which are elementary operations) so, we can quickly and easily obtain **Lc(X)**. Thus the problem of finding a good approximation for **Lg(X)** reduces to one of finding a good approximation for **Lm(X)**. This situation is depicted graphically in **Figure 2-1** where a short segment of **Lg(X)** is plotted along with its **Lc(X)** component.
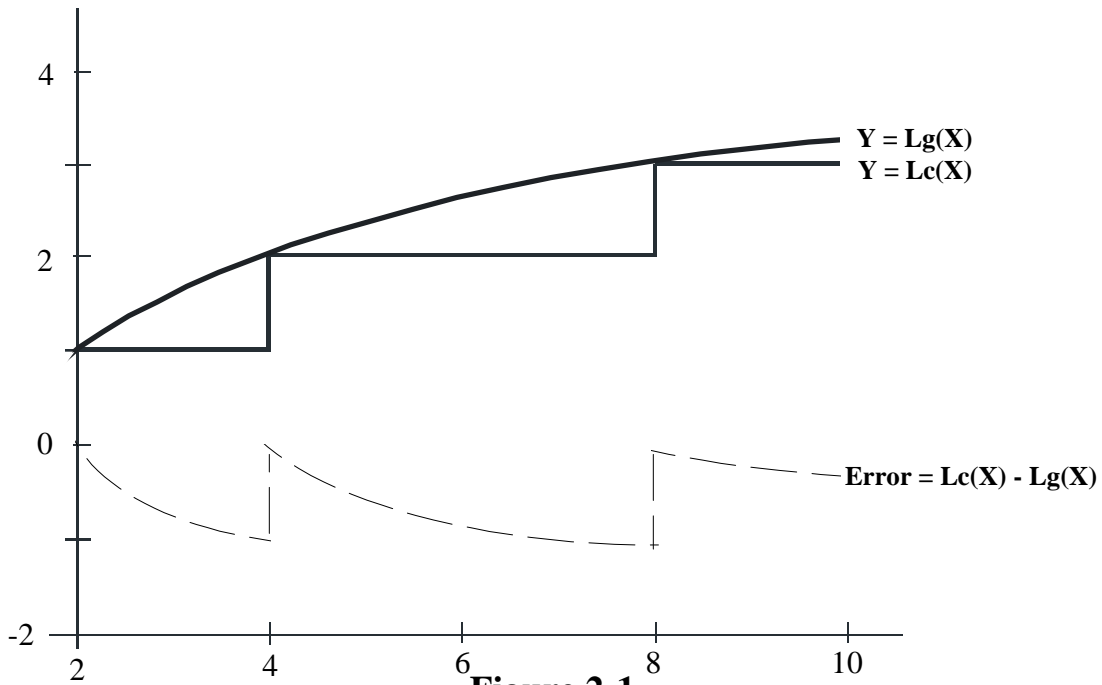
Figure 2-1
Y = Lg(X) vs Y = Lc(X)

In **Figure 2-1**, notice that **Lc(X)** has a staircase shape that only touches the **Lg(X)** curve at the points where **X** = 2, 4, and 8 in the drawing. These are the integer powers of two which of course will continue with 16, 32, 64, etc (if we were to extend the drawing on the right). If we were to use **Lc(X)** as a crude approximation for **Lg(X)**, the error in that approximation would be as shown in the bottom dashed curve. Alternatively, since we know from equation (2-1) that **Lm(X)** = **Lg(X)** – **Lc(X)**, we could simply reverse the sign of this dashed error curve (flipping it upside down) and it would be a plot of what the needed function for **Lm(X)** looks like.

Looking at the dashed error curve, we can see that the error is cyclical and it depends on **X**'s relationship to the surrounding powers of two rather than on the absolute value of **X**. Each adjacent pair of integer powers forms an interval (over which the value of **X** doubles). In **Figure 2-1**, one such interval is from **X** = 2 to **X** = 4. The next such interval is from **X** = 4 to **X** = 8, and the next such interval would be from **X** = 8 to **X** = 16 etc. In general, we will denote any such interval as beginning at X_0 and ending at X_1 (where both X_0 and X_1 are always adjacent integer powers of 2). For any of these intervals, the value of **Lm** starts at **0** and increases to a value of **1** by the end of the interval and this cycle repeats for each subsequent interval. For example, at **X** = 2, **Lm** = 0 and increases to **Lm** = 1 as **X** approaches **4**. Then **Lm** drops back to **0** only to increase to **Lm** = 1 again when **X** approaches **8**, etc. This of course produces an 'upside-down version' of the dashed error curve as already mentioned. Now, while the arithmetic interval between X_0 and X_1 keeps increasing as we move to the right, the geometric interval remains constant (ie the **difference** $X_1 - X_0$ keeps doubling for each new interval but the **ratio** X_1/X_0 remains constant at **2:1**).

Thus, in seeking an approximation for **Lm(X)**, we need only find the function for any arbitrary $X_0..X_1$ inverval. **Figure 2-2** depicts just such an arbitrary interval for analysis. In this figure, X_0 is of course assumed to fall on an integer power of two and therefore the dashed horizontal line has a **Y-value** of **Lg(X_0)** (which of course is also the value of **Lc(X)** as long as $X_0 < X < X_1$). These important relationships are expressed in equations (2-3) and (2-4).

$$(2\text{-}3) \quad X_0 = 2^{Lc(X)}$$
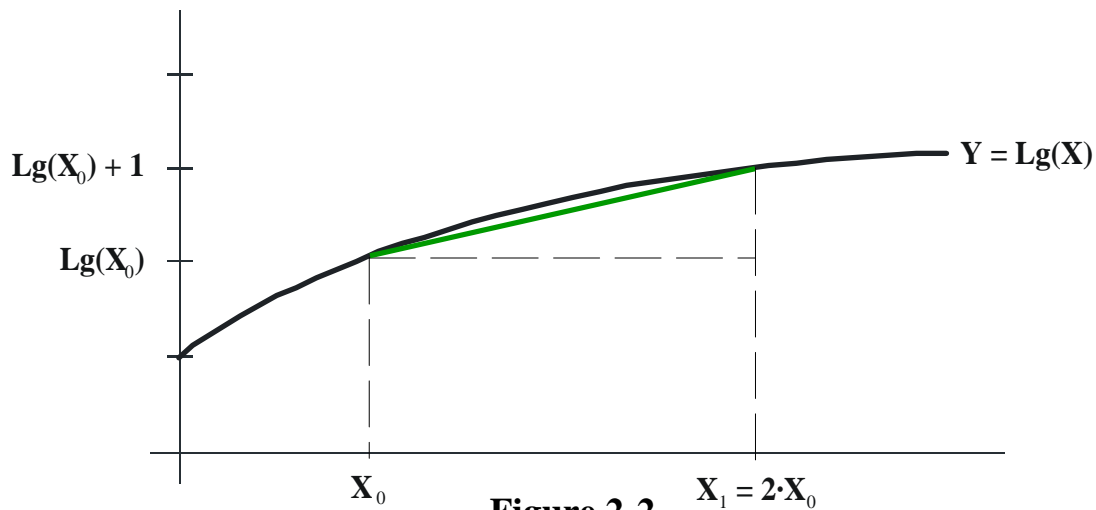$$(2\text{-}4) \quad X_1 = 2{\cdot}X_0 = 2^{Lc(X)+1}$$

Page 20

Figure 2-2
A Typical Interval from X_0 to X_1

2.4 Approximating the Mantissa

One of the first approximations for **Lm** that we might try would be to use a linear ramp for each X_0, X_1 interval such as the one depicted by the green line segment in **Figure 2-2**. But, by itself, this scheme has unacceptably large errors. And, worse than that, the general shape of the error would be very difficult to correct. So, if possible, we'd like to develop an approximation for **Lm** that's nearer to the log curve itself in shape. Now it just so happens that one unique property of a logarithmic curve is that the **slope** is a very simple algebraic function (even though the log function itself is transcendental). DeVos' idea was to exploit this property of logarithms by developing an approximation for **Lm(X)** based on the **slope** of the **Lg(X)** curve (not too dissimilar to performing a Newton-style iteration).

Refer now to **Figure 2-3** where we have modified **Figure 2-2** slightly. For one thing, we have replaced X_1 with the more general **X**. This is meant to represent any **X** value within the X_0 to X_1 interval. We have also identified the points at which X_0 and **X** intersect the log curve as P_0 and P_X respectively and, we have drawn a straight line segment connecting these two points. This line segment could be viewed as an approximation for **Lm(X)** which is correct at X_0 and **X** and is low everywhere else in the interval. The important things to remember about this line segment is that it passes through P_0 and it intersects **X** at **Y** = **Lg(X)**. Of course if we didn't already have the curve for **Y** = **Lg(X)**, we couldn't have located the point P_X (except when $X = X_1$) and therefore we couldn't have drawn the line segment from P_0 to P_X. But we're going to see if we can construct such a line segment using the slope of **Lg(X)**.
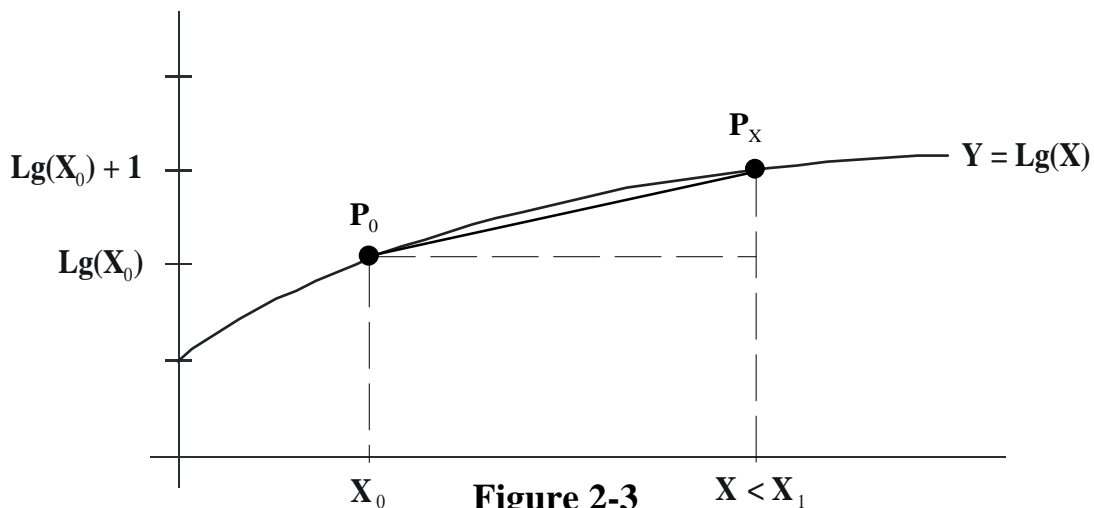

Figure 2-3
Sub Interval from X_0 to **X**

Looking at **Figure 2-3**, it's easy to see that there must be a point on the log curve, between X_0 and X, where the **slope** is the same as that of the P_0, P_X line segment. By eye, the log curve appears to have just about the right slope at a point around midway between X_0 and X as shown in **Figure 2-4**. In **Figure 2-4**, the point on the log curve that has a slope equal to that of the P_0, P_X line segment has been identified as P_M and its X coordinate has been labeled X_M.
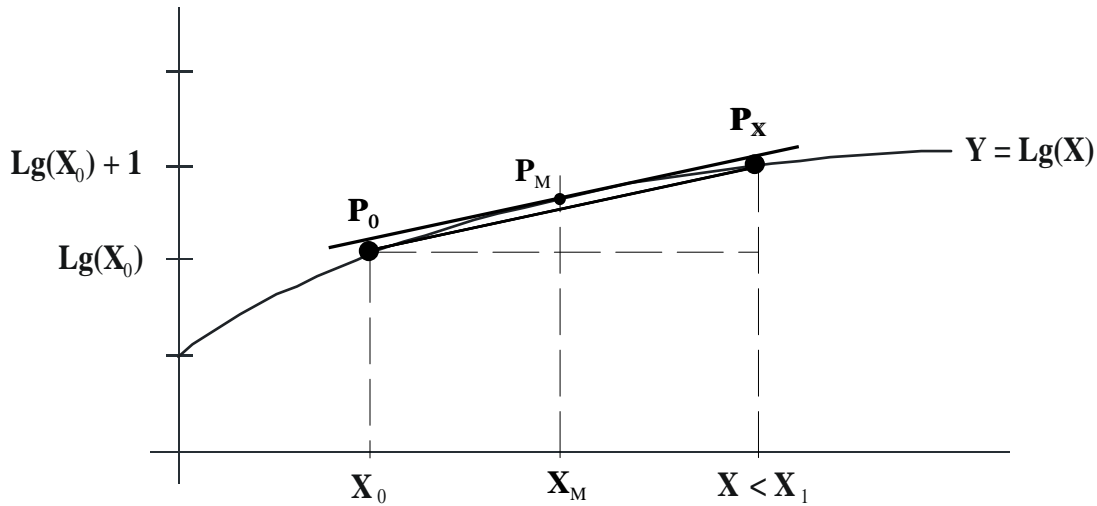
Figure 2-4
Matching Slopes

Now if we lower the P_M tangent line (keeping the same slope) until it passes through the point P_0, it will also pass through the point P_X. Thus, we **can** construct a line through P_0 and P_X if we can locate the point X_M and if we can calculate the slope of the log curve at that X_M value. From differential calculus, the **slope** (usually denoted by **m**) of a logarithmic curve such as $Y = Lg(X)$ is given by:

$$\text{(2-5)} \quad m = dy/dx = K / X$$

In equation (2-5), **K** is a **constant** (that depends on the base of the logarithm). For example $K = 1$ for the natural log function while $K = 1/\ln(2)$ for the binary log function. The important thing to note is that **K is constant** (ie independent of **X**) and therefore the slope **m** is just inversely proportional to **X**. Thus, if we have X_M, we can easily calculate the **slope** of the log curve at X_M. Referring now to **Figure 2-4**, the slope of the P_0, P_X line segment is given by equation (2-6) while the slope of the log curve at $X = X_M$ is given by equation (2-7).

$$\text{(2-6)} \quad m_L = Lm(X)/(X - X_0)$$
$$\text{(2-7)} \quad m_T = K/X_M$$

And since X_M is about midway between X_0 and X, we can express the value of X_M using something like equation (2-8) where **p** will have a value near **0.5**.

$$\text{(2-8)} \quad X_M = p{\cdot}(X_0 + X)$$

Now, substituting (2-8) in (2-7) and then equating (2-6) and (2-7), we can dervive an expression for **Lm(X)** as shown in equation (2-9).

$$\text{(2-9)} \quad Lm(X) = (K/p){\cdot}(X - X_0)/(X + X_0)$$

In equation (2-9), **K** is a constant equal to **1/ln(2)** and **p** is a value near **0.5**. It should be pointed out that as it stands, equation (2-9) is **not just an approximation** for **Lm(X)** it is *exact* (to the extent that we can determine the value of **p**). Now, it would be nice if **p** (like **K**) was a constant but unfortunately that's not possible. If **p** were a constant, we would have an algebraic formula for a transcendental function (which of course is impossible by the very definition of transcendental functions). However, while the value of **p** varies over the interval from X_0 to X_1, its variation is rather small (on the order of ±1.1% over the full interval). So, if we choose a single, representative value for **p**, equation (2-9) might provide us with a fairly-good first *approximation* for **Lm(X)**. Since **p** varies with **X** we need to choose **p** at some particular value of **X** and in order not to have a discontinuity at X_1, we need to 'force' **Lm(X) = 1** at $X = 2 \cdot X_0$. So, let's find the value of **p** at $X = X_1$. To do this, we substitute $X = 2 \cdot X_0$ in equation (2-9) and solve for **K/p**. When we do this, we obtain the simple result that **K/p = 3**. Thus, our approximation for **Lm(X)** is given by equation (2-10)

$$\textbf{(2-10)} \ \ \textbf{Lm(X)} \approx 3 \cdot (X - X_0)/(X + X_0)$$

Now combining this mantissa approximation with the characteristic we previously determined, we can express our first complete approximation for **Lg(X)** as shown in equation (2-11).

$$\textbf{(2-11)} \ \ \textbf{Lg(X)} \approx \textbf{Lc(X)} + 3 \cdot (X - X_0)/(X + X_0)$$

2.5 Error Correcting the Result

As it stands, equation (2-11) provides a fairly reasonable approximation for **Lg(X)** with an overall average relative error on the order of **0.07%**. However, when we examine a plot of the remaining error, relative to a precision value for **Lg(X),** we note that the error looks very nearly parabolic, see **Figure 2-5**. Note again that the error shape remains consistent (geometrically) from interval to interval, and thus we should be able to correct for a large part of this remaining error with a simple 2nd-order error term.
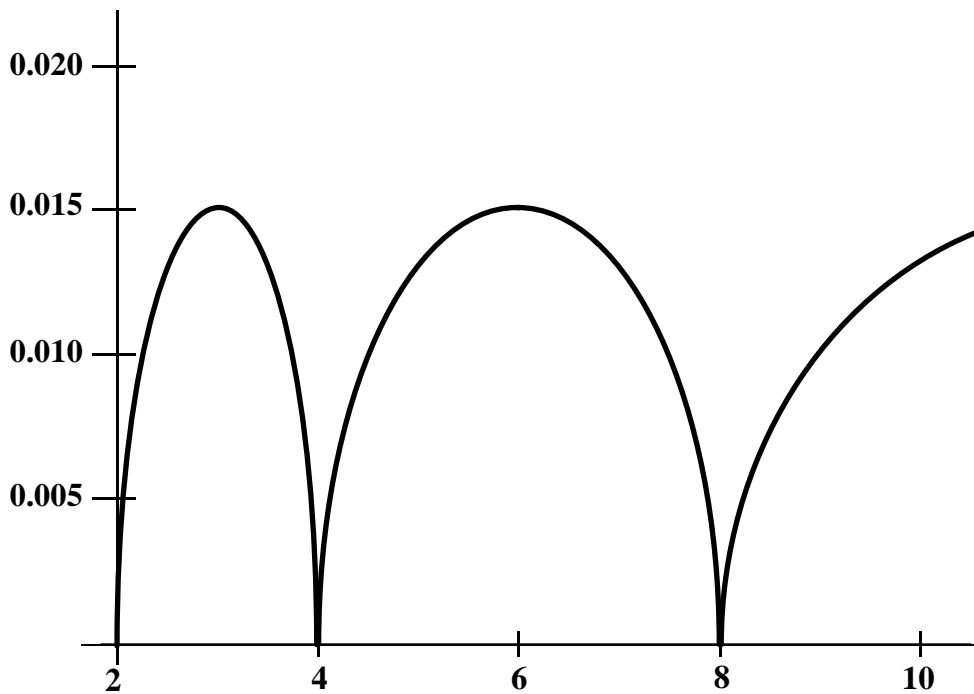
Figure 2-5
Remaining Approximation Error

An expression for a parabola such as that shown in **Figure 2-5** is given in equation (2-12). To match the peak error in **Figure 2-5, Pe** would be about **0.0150375**. However, since the residual error shape is not precisely that of a parabola, the optimum value to use for **Pe** is best determined experimentally after all the algorithmic details are in place and a numerical error analysis can be run. Meanwhile, the final expression for our **Lg(X)** approximation is shown in equation (2-13).

$$\textbf{(2-12) } err(X) = Pe\cdot[\ 1 - (2\cdot X/X_0 - 3)^2\ ]$$

$$\textbf{(2-13) } Lg(X) \approx Lc(X) + 3\cdot(X - X_0)/(X + X_0) - err(X)$$

2.6 Modeling DeVos' Method

We can now begin the process of modeling DeVos' Binary Log approximation (in preparation for adapting it to the KSP). We begin with a fairly straight-forward model that depicts the foregoing strategy in flowchart form, shown in **Figure 2-6**. This first flowchart presumes the availability of precision arithmetic so no integer scaling or operational ordering is indicated other than the scaling of the output value **Lgx**.

Log2
(X,Lgx)

Lc = TopBit
X0 = TopBitValue

Bit Lc = 0

X0 shr 1
Lc - 1

X = X0

Lgx = Lc

Lm = 3*(X – X0)/(X + X0)
err = Pe*[1 - (2*X/X0 – 3)^2]

Lgx = Lc + Lm – err

Lgx = Lgx * LogScale

Exit

Figure 2-6
DeVos Model #1 for Log2

Now, let's walk through the flow chart of **Figure 2-6** and relate it to our preceding derivation of DeVos approximation. **Log2** will be called with an input argument **X** (which is presumed to be an integer from **1** to **TopX**). The routine then returns the binary logarithm of **X** as an integer **Lgx** (the true log scaled by the value **LogScale**). Of course the reason for scaling the output is so that **Log2(X)** can be represented as an integer for the KSP.

On entry to **Log2**, we first set the variables **Lc = TopBit** and **X0 = TopBitValue**. **TopBit** and **TopBitValue** are both constants. **TopBit** is the highest on-bit pos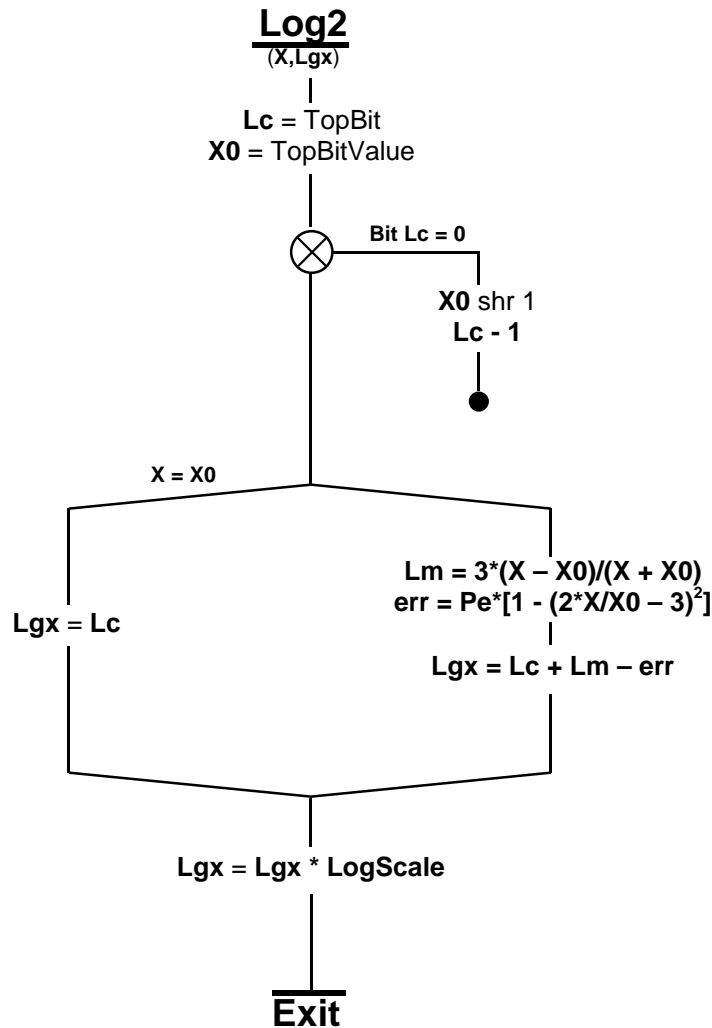ition for the maximum **X** value allowed and **TopBitValue** is the numerical value of that bit (ie **TopBitValue = 2^{TopBit}**). For example, if **TopBit = 15**, then **TopBitValue = 2^{15} = 32768**. If **TopBit is 15**, it means that the largest input **X** value cannot have bit 16 or higher set. Therefore the maximum value of **X** that can be accepted is **TopX = TopBitValue*2 – 1** (which is **65,535** in the case of **TopBit = 15**). Theoretically, for the KSP, the DeVos algorithm will operate correctly with a **TopBit** as high as 30 (bit 31 is used as the sign bit in the KSP arithmetic package). However, the lack of anything but integer arithmetic requires scaling and headroom to maintain reasonable precision. Thus, we shouldn't expect to be able to use a **TopBit** value much higher than 15. More will be said on this later.

Returning now to **Figure 2-6**, after initializing **Lc** and **X0**, we next execute a while loop to find the highest on-bit position in **X**. This can be done by using **X0** as a bit-mask and testing the corresponding bit in **X** to see if it's on. If not, **X0** is right shifted by one bit position and **Lc** (the bit number) is decremented. As long as **X** is a value between **1..TopX**, the while-loop will exit with **Lc** equal to the highest on-bit position in **X** and **X0** will be equal to the numerical value of that bit. Thus **Lc** is now the log characteristic of **X** and **X0** is the lower integer power of the interval that **X** is in.

Next, if **X = X0** (meaning that **X** is an exact integer power of two), **Lgx** is simply set to **Lc** (because the mantissa will be zero and needn't be calculated). Alternatively, if **X ≠ X0**, then the right-hand branch is taken to compute the mantissa. First the mantissa approximtion, **Lm** is computed and then the error correction term, **err**. Then **Lgx** is formed as **Lc + Lm – err**. Finally, the value of **Lgx** is multiplied by the desired output scaling factor **LogScale**. Of course when we do the real KSP version, the scaling of the output will be more distributed and will include higher interim scaling and even different amounts of scaling for the various components making up the log.

Now take a look at the flowchart shown in **Figure 2-7**. The first thing to note is we have added some 'front-end' logic to process **X** values that fall outside the range from **1 to TopX**. The log of **X = 0** is minus infinity and the log of **X < 0** is in the imaginary realm. So the first thing our 2nd model does is to simply return a value of **NAL** when **X ≤ 0**, where **NAL** is a constant currently defined as **-1,000,000**. When the routine returns **NAL**, it should be viewed as a sentinel meaning that the logarithm is 'not defined' (rather than as the correct numerical result). The next thing **Figure 2-7** does is to check **X** against **TopX**. If **X > TopX**, the routine simply returns **MaxLog** which is a constant currently defined as the log of **TopX + 1**. Only when **X** lies between **1..TopX** inclusively, does the model flowchart execute the 'guts' of the algorithm that was depicted in **Figure 2-6**.

The model algorithm of **Figure 2-7** was coded in Pascal and run using high-precision floating point arithmetic. This was done to avoid arithmetic rounding errors so that the best value for **Pe** could be determined. The best value for **Pe** was found by monitoring the error statistics while varying the value of **Pe** in the vicinity of its theoretical 0.015 value. Before running this analysis, the parabolic error function of equation (2-12) was rearranged as shown in equation (2-14).

$$(2\text{-}14) \ \ \text{err(X)} = \text{Pe} - (\text{Pe·X/(X0/2)} - 3\text{·Pe})^2/\text{Pe}$$

By arranging the expression this way, it will adapt more easily to the KSP and it will allow us to avoid a runtime multiply by 3 since we can pre-define another constant equal to **3·Pe**. We can further minimize the residual error of DeVos' approximation if we slightly adjust the value of this coefficient also.

Log2
(X,Lgx)

X ≤ 0

Lgx = NAL

X > TopX

Lc = TopBit
X0 = TopBitValue

Lgx = MaxLog

⊗ **Bit Lc = 0**

X0 shr 1
Lc - 1

●

X = X0

Lm = 3*(X − X0)/(X + X0)
err = Pe*[1 - (2*X/X0 − 3)²]

Lgx = Lc

Lgx = Lc + Lm − err

Lgx = Lgx * LogScale

Exit
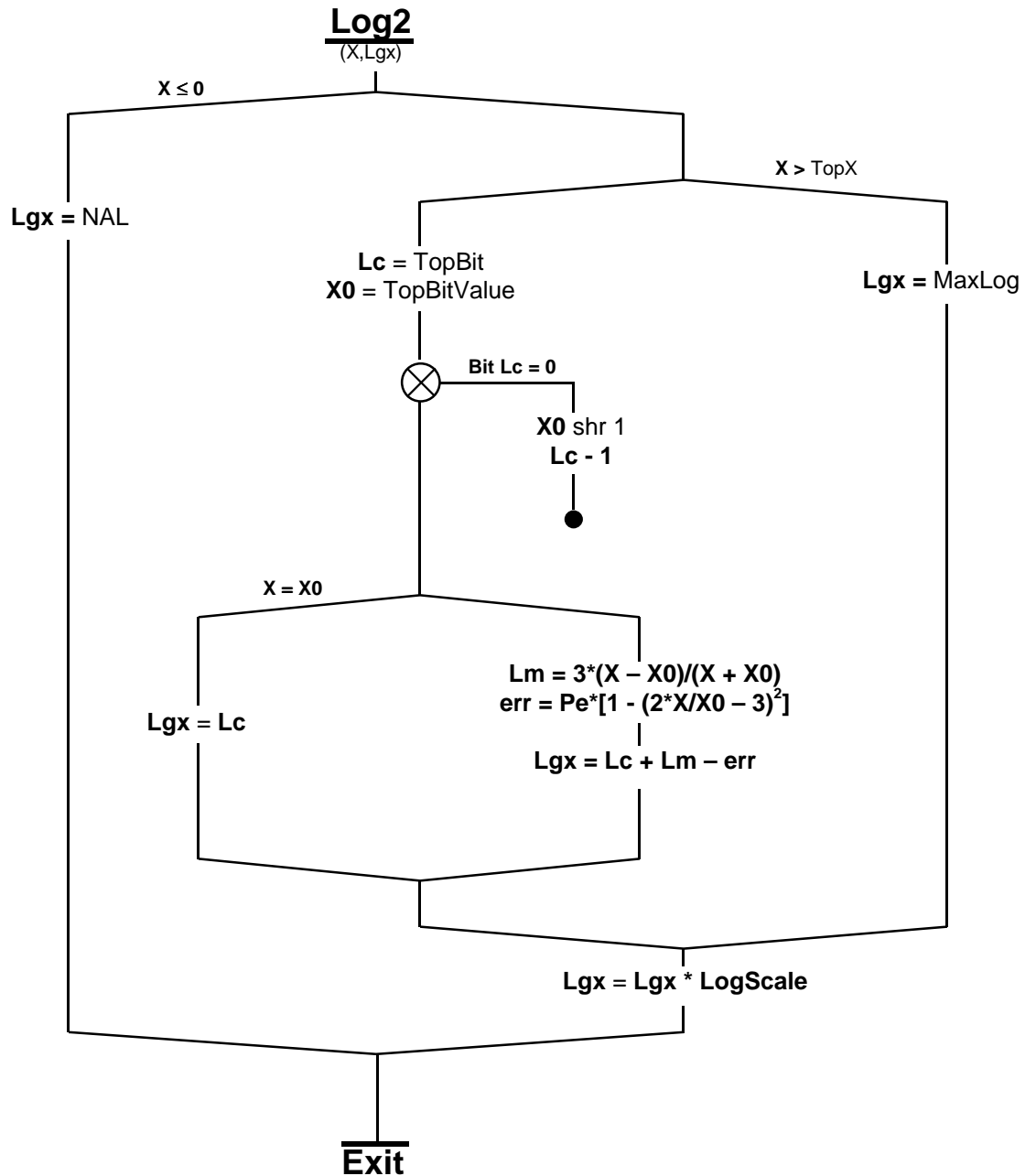
Figure 2-7
DeVos Model #2 for Log2

2.7 Minimizing Residual Error

Once it was realized that the optimum coefficient for the **3·Pe** term **wasn't going to be precisely 3**, equation (2-14) was rewritten as shown in equation (2-15) where **Pe3** is now a new constant (rather than a runtime multiply of **Pe** by 3). The final result of this error analysis revealed that the optimum values for the constants **Pe** and **Pe3** were **Pe = 0.014830** and **Pe3 = 2.991*Pe**. With these values and nearly error-less arithmetic, the residual error was analyzed and tabulated as shown in **Table 2-1**.

$$\textbf{(2-15)}\ \ \textbf{err(X)} = \textbf{Pe} - (\textbf{Pe·X/(X0/2)} - \textbf{Pe3})^2/\textbf{Pe}$$

Top 5 +/-	Absolute Error		Relative Error	
#1 +	+0.000335	@Lg(45569)	+0.015%	@Lg(3)
#2 +	+0.000335	@Lg(45797)	+0.00919%	@Lg(6)
#3 +	+0.000334	@Lg(22723)	+0.00776%	@Lg(11)
#4 +	+0.000334	@Lg(45446)	+0.0074%	@Lg(5)
#5 +	+0.000334	@Lg(22577)	+0.00662%	@Lg(12)
#1 −	-0.000335	@Lg(2095)	-0.00643%	@Lg(17)
#2 −	-0.000335	@Lg(4190)	-0.00583%	@Lg(33)
#3 −	-0.000335	@Lg(8380)	-0.00552%	@Lg(7)
#4 −	-0.000335	@Lg(16760)	-0.00517%	@Lg(34)
#5 −	-0.000335	@Lg(33520)	-0.00487%	@Lg(66)
Error Summary Data				
Avg Error:		+1.9E-5		+0.000131%
Avg |Error|:		**+0.000162**		**+0.00113%**

Table 2-1
Log2(X) Error Stats
Using High-Precision,
Floating Point Arithmetic

Input X from 3 to 65,535
with
Output Scaling = 10,000

    Table 2-1 uses the same format as the Trig error analysis **Tables 1-2, 1-3, and 1-4**. Refer to **Section 1.12** for details of this format. The main purpose of analysis with high-preciion arithmetic was to determine the best accuracy that the DeVos approximation is capable of. Unlike the Cordic where we can always do a few more iterations or like a series expansion where we could always add another term or two to increase accuracy, the DeVos method **is only as accurate as it is** and if we need more accuracy, we will probably have to seek a different algorithm. Further, we can assume that once we adapt this algorithm to the KSP, there will likely be further error introduced due to integer arithmetic limitations.

    Preliminary analysis of probable internal scaling and headroom requirements indicated that output scaling would have to be limited to about 20,000 or so. Since the most convenient scalings for users are simple powers of ten, this meant that when we adapt this algorithm to the KSP, we would likely need to set the output scale to 10,000 (since 100,000 would reduce calculation headroom too much). Therefore, for **Table 2-1**, even though our arithmetic was essentially error free, there is some quantization error introduced due to using a scaled integer for output. In other words, when taking the stats for **Table 2-1**, an output scale factor of 10,000 was used (since this was the likely KSP scaling). So, the statistics for **Table 2-1** represent about the best performance of DeVos' approximation that we can expect if we limit output resolution to 4 decimal places (10,000).

    Running the same analysis with output scaling set to **100,000** shows a modest improvement in overall accuracy. For example, the absolute **± max error** reduces to **0.00029** and the absolute **Average |Error|** becomes **0.000161**. This tells us that the DeVos approximation is a little more accurate than an output scaling of **10,000** can utilize but not much. This is essentially what we want (see the discussion on this subject in **Section 1.11**). Of course we have to realize that when we adapt this algorithm to the KSP, we can expect some further degradation of accuracy but with careful scaling and arithmetic operation ordering, we should be able to come close to the accuracy depicted in **Table 2-1**. And considering the likely uses for log functions in the KSP, the accuracy provided by this method should be sufficient for many applications.

2.8 Adapting Log2 to the KSP

When working with all-integer arithmetic, one of the biggest losses of accuracy comes from any divisions that have to be performed. For example, in integer arithmetic, 3/4 isn't 0.75 it's zero! To minimize these kinds of truncation errors, expressions need to be arranged so that as many multiplications as possible are performed before each division. The main idea is to be sure the dividend is as large as it can be made before dividing it. Thus, to keep precision high, there is a constant need to use scaling factors and to rearrange the order of operations to keep dividends large. But, on the other hand, you have to leave enough headroom so that all the scaling and other multiplying doesn't cause arithmetic overflow for some values of the input argument. Generally, you should refer to the flowchart of **Figure 2-7**, and the actual source code for **Log2**, throughout the remainder of this discussion on adapting **Log2** to the KSP.

In the normal full-calculation branch of the code, we form the output for **Lgx** by adding the 3 components **Lc**, **Lm**, and **err**, as indicated in equation (2-13), and the sum of these components is to be scaled by **LogScale** = 10000 when output. But, besides the output scaling, we want to use as much interim scaling as we can in order to minimize arithmetic truncation errors. So, even though **Lc**, **Lm**, and **err** will need to have the same scaling (in order to be added), these components can have different scalings for their interim calculations. Of course since **Lc** is a small integer of 16 or less (and its value is exact) it poses no special scaling problems. So, we will concentrate on the interim and final scaling used for **Lm** and **err**. In real numbers, **Lm** ranges from **0.0 to 1.0** and **err** ranges from **0.0 to 0.015** but the trick is to calculate these values using all-integer arithmetic while still maintaining multi-digit, fractional precision for them. Let's first concentrate on **Lm**. Equation (2-10) gives us the expression for **Lm** in real numbers but in integer arithmetic we can include a scaling factor **LmScale** as shown in equation (2-16).

$$\textbf{(2-16)} \quad \textbf{Lm(X)} = \textbf{3·LmScale·(X} - X_0)/(X + X_0)$$

To determine the maximum value we can use for **LmScale** we need to know the max value that **X** – **X0** can attain. First we note that **X** ranges from X_0 to $2·X_0$ for any interval so that $X - X_0$ can be at most X_0. Thus, the value of $\textbf{Xmax} - \textbf{X0} = 2^{TB}$ (using **TB** as a shorthand for the **TopBit** setting). And, the product of **3·LmScale·(Xmax – X0)** must of course be less than 2^{31} to avoid arithmetic overflow. Thus, we can write the inequality shown in (2-17).

$$\textbf{(2-17)} \quad \textbf{3·LmScale} < 2^{31 - TB}$$

Thus, for **TopBit** = 15, **LmScale** must be less than **21,845**. And, since final scaling for **Lm** is to be **LogScale = 10000**, it will be convenient if we keep **LmScale** a simple integer-multiple of **LogScale**. So at **TopBit = 15, LmScale = 20,000** = **2·LogScale** would seem a good choice. When we compute equation (2-16), we can avoid a runtime multiply by 3 by simply defining a new scaling constant **Lm3 = 3·LmScale**. And, since **LmScale** is **2* LogScale** (or 20,000), we have:

$$\textbf{(2-18)} \quad \textbf{Lm3} = \textbf{3·2·LogScale}$$

Now, take a look at the code line in **Log2** that first calculates **Lm,** shown here as equation (2-19). After doing this calculation, **Lm** will be scaled by **2*LogScale**. We will leave it this way until we are ready to combine it with the calculated error term.

$$\textbf{(2-19)} \quad \textbf{Lm := Lm3*(X} - X0)/(X + X0)$$

The calculation of the error term is more complex and involves more interim operations than the calculation of **Lm** as we will soon see. With real numbers, the calculation of **err** can be carried out as shown in equation (2-15). To do this calculation with all-integer arithmetic, probably the easiest way to introduce the needed scaling will be to define two integer constants named **SPe** and **SPe3** (for scaled **Pe** and scaled **Pe3** respectively). Thus, equation (2-15) can then be rewritten as equation (2-20) which will yield the scaled error term.

$$\textbf{(2-20) err = SPe} - \textbf{(SPe·X/(X}_0\textbf{/2)} - \textbf{SPe3)}^2\textbf{/SPe}$$

It is also convenient to do the calculation of (2-20) in two stages as shown in equations (2-21) and (2-22). In (2-21), we calculate the part of (2-20) that is to be squared and assign it to a temporary variable $\textbf{err}_1$. Then, in (2-22), we complete the calculation of the scaled error.

$$\textbf{(2-21) err}_1 = \textbf{SPe·X/(X}_0\textbf{/2)} - \textbf{SPe3}$$
$$\textbf{(2-22) err } = \textbf{SPe} - \textbf{err1·err1/SPe}$$

Now, to determine the max value we can use for **Pe** scaling, from (2-21) we can see that one condition is that **SPe·X** must be less than $\textbf{2}^{31}$ to avoid arithmetic overflow. Since the max value of **X** is $\textbf{2}^{16}$, we can express this condition with the inequality of (2-23).

$$\textbf{(2-23) SPe} < \textbf{2}^{15}$$

The next arithmetic overflow possibility can occur when we multiply $\textbf{err}_1$ by itself in (2-22). So, we need to determine what the max value for $\textbf{err}_1$ can be. In (2-21), for any given value of $\textbf{X} > \textbf{1}$, $\textbf{X0/2} = \textbf{2}^{\textbf{Lc}-1}$ (note for $\textbf{X} \leq \textbf{1}$, an earlier exit from the routine will be taken and **err** will not be calculated). Thus, **Lc** will not be less than one and we can perform the division of **SPe·X** by **X0/2** by simply right shifting **SPe·X** by **Lc** **– 1** bits. We know that the ratio $\textbf{2} < \textbf{ X/(X0/2)} < \textbf{4}$ and we also know that the ratio of **SPe3/SPe** is about **2.991**. Thus, we know that $\textbf{err}_1$ must be in the range of $\textbf{-0.991·SPe} < \textbf{err}_1 < \textbf{1.01·SPe}$ and thus $\textbf{err}_1\textbf{·err}_1$ is at most **1.02·SPe·SPe**. Thus, we can now express a second scaling limitation in the form of the inequality of expression (2-24).

$$\textbf{(2-24) 1.02·SPe·SPe} < \textbf{2}^{31} \textbf{ or SPe} < \textbf{1.4·2}^{15} = \textbf{45,875}$$

Comparing (2-23) and (2-24), we note that (2-23) dictates a lower scaling limit than (2-24). Therefore our scaling limit will have to be 32,767 as stated by equation (2-23). Now, the desired real value for **Pe = 0.014830** is well-approximated by the integer ratio of **16,313/1,100,000**. And, since **16,313** is less than half of **32,768** we can make **SPe = 2*16,313 = 32,626** without arithmetic overflow. Therefore, we define the scaled **Pe** constants **SPe = 32,626** and **SPe3 = 97,584** (**2.991*SPe** to the closest integer). Furthermore, since **2*1100000 = 220*LogScale**, after we calculate **err** using **SPe** and **SPe3**, we can reduce the scale to **LogScale** by dividing **err** by **220** which we define as another constant **PeScale = 220**.

Now locate the first code lines after the line that calculates **Lm** (the lines beginning with **err :=**). The first line calculates $\textbf{err}_1$ but rather than defining another variable, we just use **err**. This code line multiplies **SPe** by **X** and then right shifts the product by **Lc –1** bits. From this, **SPe3** is subtracted and thus the value of **err** is now that of $\textbf{err}_1$ in (2-21). The next line of code essentially finishes the calculation for scaled error as shown in (2-22). At this point the value in **err** is the error value scaled by **PeScale*LogScale = 2,200,000**.

Remember, on the other hand, that **Lm** is scaled by **2*LogScale** so, before **err** can be combined with **Lm**, we have to rescale it. Therefore, the next line of code rescales **err** by first left-shifting it by one bit (ie multiplying by 2) and then dividing that value by **PeScale**. The actual line of code executed is shown in equation (2-25). Thus the scaling is converted from **PeScale*LogScale** to **2*LogScale** which makes the scaling of **err** the same as the scaling of **Lm**.

$$\textbf{(2-25) err := (sh\_left(err,1) + PeRound)/PeScale}$$

Now, since we are multiplying the previously scaled error (where the scaling was 2,200,000) by an additional factor of 2 before we divided by **PeScale**, we need to verify that this does not cause an arithmetic overflow. From equation (2-12) and looking at **Figure 2-5**, we can see that **err** in real numbers (not scaled) ranges from **0 to +Pe**. Therefore, the max value of **err** is about **Pe** and **2,200,000*Pe = 32626**. This, when multiplied by 2 is thus a max of 65,252 before adding **PeRound** and dividing by **PeScale**. **PeRound** is merely half of **PeScale** and thus equals 110 (which clearly will not produce an arithmetic overflow).

So, now that **err** and **Lm** are scaled the same, we can combine them and then further reduce their scale from **LogScale*2** to **LogScale** only. At the same time, (**Lm - err**)***LogScale** can then be combined with **Lc*LogScale** to produce the desired output of **lgx = (Lc + Lm - err)*LogScale**. This last executable line of code for **Log2** is shown as equation (2-26).

$$\textbf{(2-26) lgx := Lc*LogScale + sh\_right(Lm - err + 1,1)}$$

Analyzing equation (2-26), **err** is first subtracted from **Lm** (remember at this point they are both scaled by **LogScale*2**) and then their difference (plus one for rounding) is right shifted by 1 bit effectively dividing by 2. Thus, **Lm - err** is now scaled only by **LogScale** so it can be added to **Lc*LogScale** to obtain **lgx**.

2.9 Log2 Error Data

A Pascal emulation of this KSP algorithm was written and various error data was collected. The stats, shown in **Table 2-2**, were taken for an **X** range from 3 to 65535 (X = 1 and 2 are excluded because their logs are 0 and 1 respectively and trivial to calculate without error). Note that the error data in **Table 2-2** generally comes pretty close to that of the error-less arithmetic given in **Table 2-1**, in fact the **Avg |Error|** is the same. The only significant difference is that the maximum errors are a little larger for the KSP version. This indicates that our scaling and operation ordering has accomplished its objectives.

Top 5 +/-	Absolute Error		Relative Error	
#1 +	+0.000353	@Lg(45825)	+0.015%	@Lg(3)
#2 +	+0.000352	@Lg(45069)	+0.00919%	@Lg(6)
#3 +	+0.000352	@Lg(45965)	+0.00776%	@Lg(11)
#4 +	+0.000352	@Lg(45559)	+0.0074%	@Lg(5)
#5 +	+0.000351	@Lg(699)	+0.00662%	@Lg(12)
#1 –	-0.00036	@Lg(33721)	-0.00643%	@Lg(17)
#2 –	-0.000357	@Lg(16787))	-0.00583%	@Lg(33)
#3 –	-0.000357	@Lg(33574)	-0.00552%	@Lg(7)
#4 –	-0.000354	@Lg(1041)	-0.00517%	@Lg(34)
#5 –	-0.000354	@Lg(2082)	-0.00488%	@Lg(15)
Error Summary Data				
Avg Error:	+1.81E-5		+0.000125%	
Avg |Error|:	**+0.000162**		**+0.00113%**	

Table 2-2
Log2 Error Stats
TopBit = 15
Input **X** range: 3 to 65535

2.10 Natural Logarithm Function

The function **Ln(X,lgx)** returns the Natural (or base **e**) logarithm, **lgx**, of the input integer **X**. The return value **lgx** is an integer scaled by **10,000** ie **lgx = Ln(X)·10000**. The formula used is given in equation (2-27) where **Ln(2) = 0.693147183** which, expressed as a ratio of integers, can be nicely approximated by **7050/10171** with 9 digit accuracy.

$$(2\text{-}27) \quad \mathbf{Ln(X) = Log_2(X)·Ln(2)}$$

2.11 Common Logarithm Function

The function **Log10(X,lgx)** returns the Common (or base **10**) logarithm, **lgx**, of the input integer **X**. The return value **lgx** is an integer scaled by **10,000** ie **lgx = Log₁₀(X)·10000**. The formula used is given in equation (2-28) where **Log₁₀(2) = 0.301029996** which, expressed as a ratio of integers, can be approximated (with 9-digit accuracy) by **12655/42039**.

$$(2\text{-}28) \quad \mathbf{Log_{10}(X) = Log_2(X)·Log_{10}(2)}$$

3.0 Extended Log and Exponential Functions

3.1 Introduction

The KSP Math Library's fast log functions are based on the clever **DeVos' Approximation** which provides good accuracy and fast execution (about 2 μsec, see Appendix A). However, for all its virtues, it does have its shortcomings.

1. There isn't any reasonable way to extend its accuracy.
2. Computational headroom requirements preclude an input argument range above 65,536.
3. The algorithm cannot be easily reversed to provide the antilog function.

So, for situations where any or all of the above are needed, the **KSP Math Library** now has a full set of **extended log functions** together with their **corresponding exponential/antilog counterparts**. As already discussed with the trig functions, the **CORDIC** algorithm has a number of attributes which makes it almost ideally suited for the KSP. And, while the original Cordic was developed to solve vector rotation problems, the key ideas and benefits of the Cordic have since been applied to many other kinds of problems; including the calculation of logarithms and exponentials. Since Cordic methods are so compatible with the capabilities of the KSP, the Extended Log and Exponential routines were implemented using a set of Cordic-like algorithms.

As implemented (20-bit precision), the average execution time of these functions is on the order of 10 μsec (again see Appendix A). While slower than the **DeVos Approximation**, overall accuracy is better than 6 decimal digits over the entire positive integer range (1 to 2,147,483,647). Moreover, the algorithm's accuracy can easily be extended further, to as high as 30 bits (at a time cost of about 0.6 usec per additional bit). Of course Cordic methods always require a small lookup table (one entry per bit of precision) but, the same lookup table supports both logs and the corresponding exponential (antilog) functions.

3.2 Extended Log Functions

Cordic techniques can be adapted to compute **Logarithms** rather easily and the specifics of how it was done for the KSP Math Library will now be described. Throughout the following discussion, we will use the shorthand notation **Lg(X)** to mean $\log_2(X)$, ie the base 2 logarithm of **X**. First we note that any integer **X > 0**, can be *normalized* to the range of **1.0 ≤ nx < 2.0** as shown in equation (3-1). Therefore we can easily reduce the domain of our algorithm to that of **nx**, as shown in equation (3-2).

$$\textbf{(3-1)} \ \ \mathbf{X = nx \cdot 2^C}$$

$$\textbf{(3-2)} \ \ \mathbf{Lg(X) = Lg(nx \cdot 2^C) = Lg(nx) + Lg(2^C) = C + Lg(nx)}$$

In equation (3-2), the term **C** is the integer part of **Lg(X)** (the characteristic) and, for 32-bit signed integers, is in the range **0 ≤ C < 31**. The term **Lg(nx)** is the fractional part of **Lg(X)** (the manitissa) and is confined to the range of **0.0 ≤ Lg(nx) < 1.0**. Thus our 'inner' algorithm need only operate within the limited range of **nx**.

Now, since we know that the log of a product is the same as the sum of the logs of its factors, we can write the identity shown in equation (3-3). Or, for easier subsequent discussion and manipulation, we can restate this identity with the more usual compact notation of equation (3-4).

$$\textbf{(3-3)} \ \ \mathbf{Lg[nx \cdot (1 + b_1 \cdot 2^{-1}) \cdot (1 + b_2 \cdot 2^{-2}) \cdot \ ... \ (1 + b_N \cdot 2^{-N})] \ =}$$
$$\mathbf{Lg(nx) + Lg(1 + b_1 \cdot 2^{-1}) + Lg(1 + b_2 \cdot 2^{-2}) + \ ... \ Lg(1 + b_N \cdot 2^{-N})}$$

$$\text{(3-4)} \quad \text{Lg}[nx \cdot \prod_{n=1}^{N}(1 + b_n \cdot 2^{-n})] = \text{Lg}(nx) + \sum_{n=1}^{N}\text{Lg}(1 + b_n \cdot 2^{-n})$$

In this identity, b_n is either **0 or 1** and will be determined for each inner loop iteration based on a comparison to be defined subsequently. By examining the **right side** of equation (3-4), we note that when $b_n = 0$, the summation term will become **Lg(1) = 0** and when $b_n = 1$, the same term becomes $\text{Lg}(1 + 2^{-n})$. Therefore, we can reposition b_n (without changing the identity) as shown in equation (3-5).

$$\text{(3-5)} \quad \text{Lg}[nx \cdot \prod_{n=1}^{N}(1 + b_n \cdot 2^{-n})] = \text{Lg}(nx) + \sum_{n=1}^{N}b_n \cdot \text{Lg}(1 + 2^{-n})$$

On the left side of (3-5) we note that **nx** is multiplied by **N** factors that are either $1 + 2^{-n}$ (when $b_n = 1$) or **1.0** (when $b_n = 0$). We are going to design our algorithm to choose the b_n series such that the product of **nx** by these **N** factors will be driven to **2.0**. And, as the left-side product approaches **2.0**, equation (3-5) can be rewritten as equation (3-6) from which, after solving for **Lg(nx)**, we obtain equation (3-7).

$$\text{(3-6)} \quad \text{Lg}(2.0) = \text{Lg}(nx) + \sum_{n=1}^{N}b_n \cdot \text{Lg}(1 + 2^{-n})$$

$$\boxed{\text{(3-7)} \quad \text{Lg}(nx) = 1.0 - \sum_{n=1}^{N}b_n \cdot \text{Lg}(1 + 2^{-n})}$$

Figure 3-1 illustrates the iteration-loop logic needed to accomplish the job. To drive the cumulative **nx** product to **2.0**, each loop pass determines if multiplying by the next factor, $(1 + 2^{-n})$, will move **nx** closer to **2.0** without exceeding it. If so, b_n is considered one and **nx** is multiplied by $(1 + 2^{-n})$, ie **nx** is increased by **sx**. Meanwhile, the desired mantissa is accumulated in **m** which is initialized to **1.0** outside of the loop and then each time **nx** is increased, $\text{Lg}(1 + 2^{-n})$ is subtracted from **m**. The **N** values for $\text{Lg}(1 + 2^{-n})$ are pre-computed and stored in a **Logs** array. Each element of the **Logs** array is computed per equation (3-8).

$$\text{(3-8)} \quad \textbf{Logs[n]} = \textbf{Lg}(1 + 2^{-n})$$

Note that if adding **sx** to **nx** would cause it to exceed **2.0**, b_n is essentially considered to be zero and the right, null branch is taken. Thus the value of **nx** is effectively multiplied by **1.0** and accordingly, **zero** is subtracted from the manitissa **m**, as dictated by equation (3-5). Note also that calculation of **sx** involves only a right shift of **nx** by **n** bits.

In order for this algorithm to work, it must always be possible to force the **nx** product to approach **2.0** within the required accuracy. Note that the series of factors, $(1 + 2^{-n})$ follow a decreasing geometric progression, namely **1.50, 1.25, 1.125,** etc. For **N = 20**, the maximum product of these **N** factors is somewhat greater than **2.38** (when all $b_n = 1$). Therefore, as long as **nx** is in the **domain** from **0.84** (ie 2/2.38) to **2.0**, the **nx** product can always be forced to approach **2.0** closely. How closely depends on the value of **nx** itself but the product can always be driven to within ± one-half of the LSB's value (if we bias the mantissa seed properly to balance the positive and negative errors, this will be discussed later).

With **N < 20**, the domain of **nx** is slightly reduced. For example, with an **N** of only 6, the maximum product is > 2.34. This translates to a convergence domain from **0.86** to **2.0**. But since the normalized **nx** always exceeds **1.0**, we will never have a problem with convergence. **Figure 3-2** depicts the complete algorithm for **XLog2** and its core support routine **XLg.Core** is shown in **Figure 3-3**. These charts will be discussed in the next section.

$$1.0 < nx < 2.0$$

$$m = 1.0$$
$$n = 1$$

$$n \leq N$$

$$sx = nx \cdot 2^{-n}$$

$$(2.0 - nx) \geq sx$$

$$nx = nx + sx$$
$$m = m - \text{Logs}[n]$$

$$n = n + 1$$
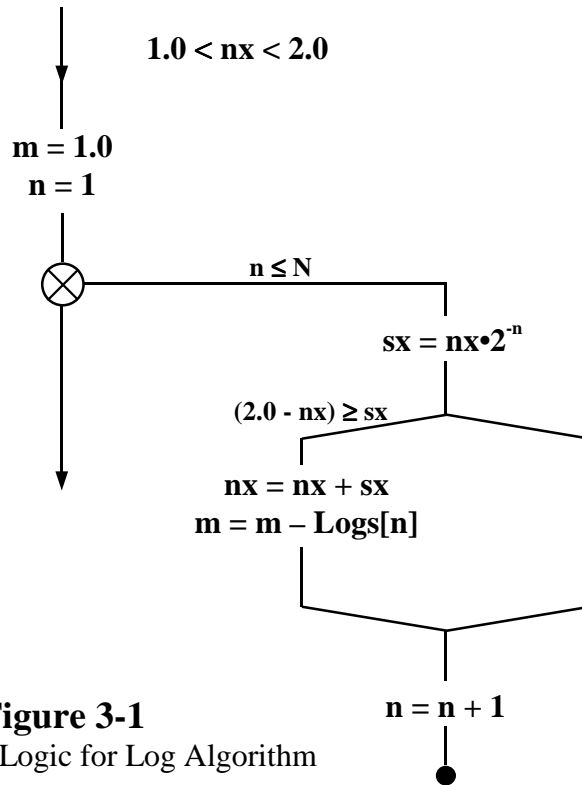
Figure 3-1
Iteration Loop Logic for Log Algorithm

XLog2
(X, lgx)

$$lgx = \text{Lg}(X) \cdot 10^6$$

$$X \leq 0$$

$$lgx = \text{NAL}$$

XLg.Core
(X, C,M)

$$lgx = 10^6 \cdot C + M$$

Exit
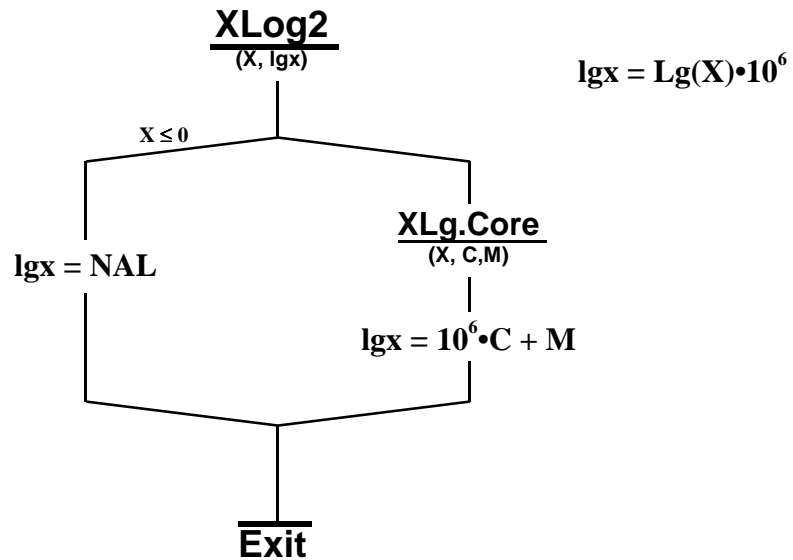
Figure 3-2
Extended Binary Log Function

XLg.Core
(X,C,m)

$$Lg(X) = C + m \cdot 10^{-6}$$
$$0 \le C \le 30$$
$$0 \le m < 10^6$$

nx = X
C = 30

⊗

nx = 2•nx
C = C – 1

{ 1.0 ≤ nx < 2.0 }

nx = 1.0

m = 1.0 - Logs[N]/2
{ Scaled by 10^9 }

m = 0
{ Even Power of 2}

n = 1

⊗   (n ≤ N) and (nx ≠ 2.0)

$sx = nx \cdot 2^{-n}$

(2.0 – nx) ≥ sx

NOTE: Logs[n] Table
is scaled by 10^9

nx = nx + sx
m = m – Logs[n]

m = (m + 500)/1000
{ Downscale m to 10^6 }

n = n + 1

Exit

Figure 3-3
Binary Log Core Function

3.2.1 XLog2 Algorithm

Figure 3-2 shows the 'outer' shell of the **Extended Binary Log** function. The routine begins by testing the input value **X**. If **X** is zero or less, it's logarithm is undefined (either it's infinite or imaginary) so the token value for **Not A Log**, is output (**NAL** is currently defined as $-100 \cdot 10^6$). However, if **X** is a positive integer greater than zero, the core support function **XLg.Core** is invoked. This routine accepts any **X > 0** and returns **Lg(X)** in the form of an integer characteristic **C**, plus a mantissa **m** (scaled by a factor of 10^6). Finally, the scaled output result, **lgx**, is computed by multiplying **C** by 10^6 and adding it to the already scaled manitissa, **m**. Thus the output result is **Lg(X)** scaled by 10^6. Obviou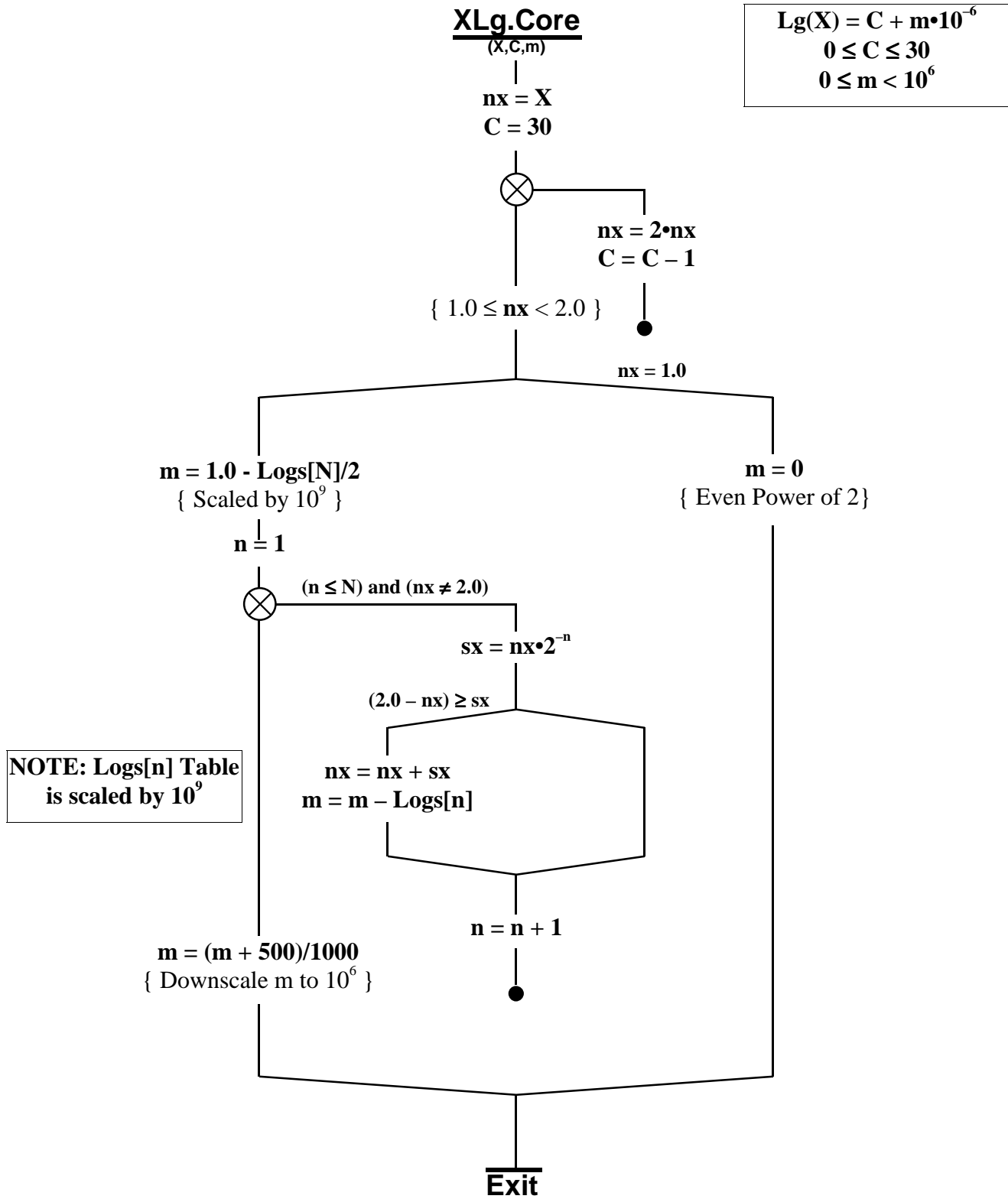sly, most of the work is done by the **XLg.Core** routine but this routine is factored out of **XLog2** because it is also used by **XLn** and **XLog10**.

A flow chart for **XLg.Core** is shown in **Figure 3-3**. Remember that on entry to this function **X** is always guaranteed to be greater than zero. Since **X** is an integer, it has an implied binary point just to the right of bit 0 and it has an implied exponent of 2^0. The variable **nx**, on the other hand, is to be interpreted as though it has a binary point just to the right of bit 30. Therefore, when **X** is copied into **nx**, it has the effect of reducing its value by a factor of 2^{30}. By initially setting **C = 30**, and viewing **nx** with its binary point between bits 29 and 30, we can say that $\mathbf{nx} \cdot 2^C = \mathbf{X}$. Next, we normalize **nx** by testing bit 30 and if it's zero, we left shift **nx** until bit 30 becomes one. We also decrement the exponent **C** for each left shift of **nx**. Thus, when the normalizing loop exits, $\mathbf{nx} \cdot 2^C$ still has the same value as **X**, but now **nx** is normalized to the range $\mathbf{1.0 \le nx < 2.0}$ and **C** is not only the **nx** exponent it is now also the integer characteristic of **Lg(X)**.

After normalization, **nx** is tested to see if only bit 30 is on (ie all other bits are off). If so, it means that **X** is an integral power of 2 and the mantissa will be zero (so the right-hand branch is taken). However, if **nx** ≠ **1.0**, the left branch is taken to compute the mantissa. Before entering the iteration loop, **m** is initialized to **1.0** (minus a rounding bias) and **n** is set to one for the first iteration. The reason for the rounding bias is as follows. When the iteration loop runs it will try to drive $\mathbf{nx} \to \mathbf{2.0}$. However, since **nx** is never allowed to exceed **2.0**, the final value for **nx** will always be **2.0 or less, but never more**. Unlike the Cordic algorithm used to compute trig functions, this variant uses a one-sided series instead of an alternating series. Thus, the mantissa which is reduced for each increment of **nx**, will either be at or above the correct result **but never below it**. Since the maximum mantissa error that can occur is equal to the **Logs** value for the least significant bit, ie **Logs[N]**, the error 'noise' will be confined to the zone from zero to **Logs[N]**. By initially subtracting **Logs[N]/2** from the mantissa, we 'balance' the errors by moving the error 'noise' into the zone from –**Logs[N]/2 to +Logs[N]/2**. It also should be pointed out that, for maximum precision, the mantissa is scaled up by a factor of 10^9 (as are all the **Logs[n]** table entries).

The main logic of the iteration loop has already been discussed in conjunction with **Figure 3-1**, but now we need to discuss a subtle detail. Since we are trying to 'drive' **nx** to **2.0**, and since **nx** has its binary point just to the right of bit 30, if we do manage to drive **nx** precisely to **2.0**, bit 31 (the sign bit) will be turned on. If the KSP had an unsigned integer type this wouldn't be a problem (since we don't need any negative values for this application). But, unfortunately, the KSP has only a signed integer data type available for us to use. However, the internal representation employs 2's complement coding for negative values. Therefore, if all we do is add and subtract, we can use KSP integers as though they were unsigned and allow the use of bit 31 to represent **2.0**. For example, suppose the current value of **nx = 0x7FFFFFFF**, and we add one to it. It will very nicely become **0x80000000** which is the same thing an unsigned integer would do. Problems only arise if we try to multiply, divide, compare, right shift (or display) such a value because in the KSP, all such operations treat **0x80000000** as though it was a big **negative** number. The long and the short of it is that we can treat **nx** as though it were an unsigned, 32-bit integer as long as we are very careful about what kind of operations we subject it to and how we interpret what happens. In the iteration loop of the **Xlg.Core** routine, there are two related areas where we need to be aware of this potential sign bit problem.

As we iterate and 'push' the value of **nx** toward **2.0**, we do so from the low side, ie we never allow **nx** to exceed **2.0**. However, if during some iteration, **nx** becomes precisely **2.0**, we can quit the iteration process because the mantissa will be as accurate as its going to get. Now take a look at **Figure 3-3** and the **while-loop's** condition for continuing. The two conditions imposed are that the loop index **n** must not exceed **N** and **nx** must not be precisely **2.0**. Therefore, on any loop pass, if **nx** should become precisely **2.0** (ie 0x80000000), the iteration loop will terminate. This allows us to save the time it would take to perform the remaining (**N** – **n**) iterations but, that is **not** the primary reason we need to do this. **The main reason for taking this early exit is to protect us from the sign-bit problem.** To see why this is so, we need to look at the loop's comparison test (for whether or not we can add **sx** to **nx** without exceeding **2.0**).

Again, refering to **Figure 3-3**, we note that the left branch of the iteration loop requires that we compare the **difference** between **nx** and **2.0** to see if it is equal to or greater than the proposed **sx** increment we want to add to **nx**. As long as **nx** < **2.0**, we can subtract **nx** from **2.0** and the difference will be a positive integer. Remember that **2.0** is actually **0x80000000** and if we subtract from that any **nx** value in the range **1.0** < **nx** < **2.0**, it will result in a value less than 0x80000000 which the KSP will see as a positive value that can then be compared against the positive value of **sx** and come up with the correct decision. However, if **nx** = **2.0** (ie 0x80000000), when we try to compute **sx** by right shifting **nx**, we will run afoul of the sign-bit problem right off. This is because the KSP only has an arithmetic right shift (ie the sign bit is always propagated). So, for example, if we want **sx** to be one-half of **nx** (viewed as a positive value), when we right shift it we would want 0x80000000 to become 0x40000000. But, the KSP will instead produce 0xC0000000). Therefore, when we compare (**2.0** – **nx**) against **sx, sx** (which is now negative) will always appear to be less than (**2.0** – **nx**). By taking an early exit of the iteration loop when **nx** = **2.0**, we can avoid this problem.

After exiting the iteration loop, the manissa is rescaled down from 10^9 to 10^6 before exiting the **XLg.Core** routine. Remember that 10^9 is the scaling used by the **Logs** table to obtain as much precision as possible with a simple power of ten. Another factor of 2 could be used without exceeding the posiitve integer range but 10^9 will be more than adequate for our current, 6-digit output resolution.

I should mention that if you examine the coding for the mantissa calculation, you will see a few differences. For one thing the total number of iterations **N** is replaced by the constant named **LogBits** and the rounding bias of **Logs[N]/2** is replaced by the constant named **LSB2**. You might also notice that the **while-loop** test for **nx** = **2.0** is actually coded as **nx ≥ 0**. This is because **2.0** is actually **0x80000000** which the KSP sees as a negaitve number. So as long as **nx ≥ 0**, it hasn't yet reached **2.0**. Finally, the comparision of (**2.0** – **nx**) against **sx**, which could have been coded as:

if (0x80000000 – nx) >= sx

is coded a little more compactly by utilizing some old-fashioned, bit-twiddling tricks. The actual coding used is:

if (-nx - sx) < 0

Proof that this construct yields the same result as the former, will be left as an exercise for the intrepid. ;-)

3.2.2 XLog2 Error Data

Absolute and Relative errors are shown for two **X** ranges in **Tables 3-1 and 3-2**. **Table 3-1** was run for all integer input from **3 to 10,000,000** while **Table 3-2** was run over the entire positive integer range from **3 to 2,147,843,647** (as usual I left out **X = 1 and X =2** because their log calculation is trivial). The first thing to note is that there is very little difference in the two tables which indicates uniformity over the entire integer range (ie larger input arguments do not incur larger errors). This is typical of a good-quality log function.

Table 3-1
XLog2 Error Data
X = 3 to 10,000,000

Top 5 +/-	Absolute Error		Relative Error	
#1 +	+1.19E-6	@Lg(6747585)	+1.48E-5%	@Lg(181)
#2 +	+1.18E-6	@Lg(3664835)	+1.34E-5%	@Lg(249)
#3 +	+1.18E-6	@Lg(7329670)	+1.33E-5%	@Lg(67)
#4 +	+1.18E-6	@Lg(8025027)	+1.31E-5%	@Lg(362)
#5 +	+1.18E-6	@Lg(9874757)	+1.27E-5%	@Lg(193)
#1 –	-1.19E-6	@Lg(7021419)	-4.72E-5%	@Lg(5)
#2 –	-1.19E-6	@Lg(2893151)	-3.3E-5%	@Lg(10)
#3 –	-1.19E-6	@Lg(5786302)	-3.28E-5%	@Lg(7)
#4 –	-1.19E-6	@Lg(4900791)	-3.16E-5%	@Lg(3)
#5 –	-1.19E-6	@Lg(9801582)	-2.53E-5%	@Lg(20)
Error Summary Data				
Avg Error:	-3.94E-9		-1.86E-8%	
Avg |Error|:	**+4.05E-7**		**+1.86E-6%**	

Table 3-2
XLog2 Error Data
X = 3 to 2,147,483,647

Top 5 +/-	Absolute Error		Relative Error	
#1 +	+1.19E-6	@Lg(362754385)	+1.48E-5%	@Lg(181)
#2 +	+1.19E-6	@Lg(725508770)	+1.34E-5%	@Lg(249)
#3 +	+1.19E-6	@Lg(1451017540)	+1.33E-5%	@Lg(67)
#4 +	+1.19E-6	@Lg(1897458009)	+1.31E-5%	@Lg(362)
#5 +	+1.19E-6	@Lg(111726627)	+1.27E-5%	@Lg(193)
#1 –	-1.2E-6	@Lg(1698874571)	-4.72E-5%	@Lg(5)
#2 –	-1.2E-6	@Lg(1311428671)	-3.3E-5%	@Lg(10)
#3 –	-1.19E-6	@Lg(1311871435)	-3.28E-5%	@Lg(7)
#4 –	-1.19E-6	@Lg(1179306189)	-3.16E-5%	@Lg(3)
#5 –	-1.19E-6	@Lg(1675314951)	-2.53E-5%	@Lg(20)
Error Summary Data				
Avg Error:	-3.94E-9		-1.33E-8%	
Avg |Error|:	**+4.04E-7**		**+1.37E-6%**	

On the Absolute side of the ledger, the max errors are no more than 3 times the average variance (ie the average error magnitude, **Avg |Err|**). And, the +/- errors are well balanced since the **Avg Err** is about 2 orders of magnitude below the **Avg |Err|.** Overall the error stats are fairly impressive for a output scaling of 10^6. On the Relative side of the ledger, the worst case (low-valued argument) error is less than 0.0001%.

3.2.3 XLn and XLog10 Functions

The natural, **XLn**, and common, **XLog10**, functions are derived from **XLog2** by multiplying by the constants **Ln(2)** and **log(2)** respectively. However, this process is not quite as simple as it was for the **Ln** and **Log10** functions. The extended log functions have 100 times greater precision and produce output values more than 100 times greater than their DeVos counterparts which leaves less 'headroom' for the multiplication. The maximum output value of the **XLog2** function is nearly $31 \cdot 10^6$ which leaves only a factor of 69 or so left for headroom. To illustrate this dilemma, for **XLog10** we need to multiply **XLog2** by a ratio of integers which will approximate **log(2) = 0.3010299957** to a sufficient degree of accuracy to maintain most of **XLog2**'s precision. If the numerator of this ratio is restricted to be no higher than 69, the best ratio available is: **59/196** which has a value of **0.301020408** which has too much error compared to **XLog2** itself.

There are two basic techniques we can use to 'punch our way out of this box'. For one thing, we can keep the integer part of **XLog(X) = C** separate from the fractional part **M**, and operate on them separately. This will 'open up' the headroom, especially for **C** where it is needed most. This is the reason that the core support routine outputs **M** and **C** values separately. Another thing we can do is try to find a suitable ratio for **M** that has a lot of prime factors. If we can come up with such a ratio, we can alternate several multiplies and divides and get at least some of the precision of a larger numeratior without exceeding our headroom. However, in order to be able to recombine **M** and **C** when we're done, **C** must be scaled by $10^6 \cdot \log(2)$ because **M** is already scaled by 10^6.

The needed **C** factor of $10^6 \cdot \log(2) = 301029.9957$ (to 10 significant decimal digits) can be very closely approximated with the ratio **70139989/233**. Since **C** is at most 30, the multiplier of 70139989 can be used without exceeding the positive integer limit. For the needed **M** factor, the ratio **105244/349613 = 0.301029996** has about 9 digit accuracy. But, the maximum value of **M** is 10^6, so we can't simply multiply by 105244 without exceeding the positive integer limit. However $105244 = 2^2 \cdot 83 \cdot 317$ and $349613 = 11 \cdot 37 \cdot 859$. Therefore, we can multiply **M** by this **log(2)** approximation with **(M•1208/407•83 + 400)/859**, where the +400 is for rounding adjustment. Note that 400 was determined empirically to provide slightly better error balance than the normal half-adjust value of 429.

Table 3-3 presents the error data for **XLog10**. When comparing **Table 3-3** with **Table 3-2**, at first it might appear that **XLog10** has less error than **XLog2** on which it is based! For example, the average absolute variance for **XLog2** is $4.04 \cdot 10^{-7}$ whereas the average absolute variance for **XLog10** is only $2.74 \cdot 10^{-7}$. Yet we know intuitively that the base conversion we performed will have to introduce additional error over that of the **XLog2** function itself. The reason for this apparent contridiction is that for any given **X**, **XLog10(X)** is less than one third the value of **XLog2(X)**. If the multiplication by **log(2)** was errorless, the resulting error in **XLog10** should be around 3 times less than the error in **XLog2**. The fact that the error is not that much smaller clearly demonstrates that we did introduce additional error in converting the log base from 2 to 10. This can also be seen by comparing the **Relative** average variance for **XLog2** and **XLog10** which are $1.37 \cdot 10^{-6}\%$ and $3.09 \cdot 10^{-6}\%$ respectively. So, far from getting a 'free lunch' we have worsened the precision by a factor of about **2.25** or so but it's still quite respectable.

Converting **XLog2(X)** to **XLn(X)** is performed in a similar way except that we don't need to use the prime factor trick. For **XLn** we multiply **C** by **49906597/72** and **M** by **1588/2291**. **Table 3-4** presents the error stats for the **XLn** function. Comparing the **Relative** variance of **XLn** with **XLog2**, you can see that we actually fared a little bit better than for **XLog10** in that error has only been increased by about **1.7** or so.

Table 3-3
XLog10 Error Data
X = 3 to 2,147,483,647

Top 5 +/-	Absolute Error		Relative Error	
#1 +	+9.46E-7	@Log(1668902840)	+3.12E-5%	@Log(19)
#2 +	+9.46E-7	@Log(1399880790)	+3.02E-5%	@Log(193)
#3 +	+9.46E-7	@Log(1668902841)	+3.02E-5%	@Log(121)
#4 +	+9.46E-7	@Log(1668902842)	+2.72E-5%	@Log(249)
#5 +	+9.46E-7	@Log(1399880791)	+2.69E-5%	@Log(386)
#1 –	-8.96E-7	@Log(15887)	-6.58E-5%	@Log(11)
#2 –	-8.93E-7	@Log(8923)	-5.34E-5%	@Log(3)
#3 –	-8.91E-7	@Log(31774)	-5.34E-5%	@Log(9)
#4 –	-8.89E-7	@Log(17846)	-5.07E-5%	@Log(22)
#5 –	-8.87E-7	@Log(63548)	-4.12E-5%	@Log(44)
Error Summary Data				
Avg Error:	+4.28E-8		+4.79E-7%	
Avg |Error|:	**+2.74E-7**		**+3.09E-6%**	

Table 3-4
XLn Error Data
X = 3 to 2,147,483,647

Top 5 +/-	Absolute Error		Relative Error	
#1 +	+1.79E-6	@Ln(34114266)	+3.47E-5%	@Ln(19)
#2 +	+1.78E-6	@Ln(33726560)	+2.31E-5%	@Ln(38)
#3 +	+1.78E-6	@Ln(34467321)	+2.2E-5%	@Ln(8)
#4 +	+1.78E-6	@Ln(38593106)	+2.18E-5%	@Ln(772)
#5 +	+1.78E-6	@Ln(34894585)	+2.17E-5%	@Ln(563)
#1 –	-1.82E-6	@Ln(2140567513)	-5.9E-5%	@Ln(7)
#2 –	-1.82E-6	@Ln(2140567512)	-5.67E-5%	@Ln(5)
#3 –	-1.82E-6	@Ln(2140567511)	-2.99E-5%	@Ln(35)
#4 –	-1.82E-6	@Ln(2140567510)	-2.92E-5%	@Ln(70)
#5 –	-1.82E-6	@Ln(2140567509)	-2.88E-5%	@Ln(140)
Error Summary Data				
Avg Error:	-3.01E-7		-1.43E-6%	
Avg |Error|:	**+4.72E-7**		**+2.3E-6%**	

3.3 Exponential (Antilog) Functions

In addition to the Extended Log functions, the KSP Math Library provides a complete set of complementary exponential functions. **Exp2**, **Exp10**, and **exp** provide the inverse functions of **XLog2**, **XLog10**, **XLn** respectively. For example, **Exp2(lgx,X)** accepts an input value **lgx** (scaled by 10^6) in the range from **0.0 to 31.0** and returns a positive integer result **X** in the range from 1 to 2,147,483,647. Or, mathematically, $X = 2^{lgx/1000000}$. The basic exponential algorithm is implemented by 'reversing' the Cordic techniques used for the extended log function. In fact, the same **Logs** array is used so no additional lookup table is required. Also, like the extended log functions, the natural and common exponential routines (**exp** and **Exp10**) are both based on the binary exponential core routine.

3.3.1 Exp2 Algorithm

To compute the value of $X = 2^Z$, for any real value of **Z**, we can reduce the domain of the exponent by noting that we can always replace **Z** with **C + M**, where **C** is an integer and **M** is in the range of **0.0 ≤ M < 1.0**. Therefore, **X** can be expressed as shown in equation (3-9) and computing $X = 2^Z$ can be reduced to computing 2^M where $1.0 \le 2^M < 2.0$ and then left shifting the result by **C** bits.

$$(3\text{-}9) \quad X = 2^{C+M} = 2^M \cdot 2^C$$

To develop the necessary Cordic relationships, we begin with the identity of equation (3-10), or more compactly as shown in equation (3-11).

$$(3\text{-}10) \quad f_1 \cdot f_2 \cdot f_3 \ldots f_N = 2^Q \qquad \text{where: } Q = Lg(f_1) + Lg(f_2) + Lg(f_3) + \ldots Lg(f_N)$$

$$(3\text{-}11) \quad \prod_{n=1}^{N} f_n = 2^Q \qquad \text{where: } Q = \sum_{n=1}^{N} Lg(f_n)$$

These equations merely state that the product of any set of factors is equal to the antilog of the sum of the logs of those same factors. This of course is a fundamental property of logs and antilogs. Now, multiplying both sides of equation (3-11) by 2^M and then solving for 2^M, we obtain equation (3-12).

$$(3\text{-}12) \quad 2^M = 2^M / 2^Q \cdot \prod_{n=1}^{N} f_n = 2^{M-Q} \cdot \prod_{n=1}^{N} f_n$$

Now, if we choose the factors $f_1, f_2, \ldots f_N$ to be as shown in equation (3-13), we can rewrite equation (3-12) as shown in equation (3-14). Like for the **XLog2** algorithm, b_n will always be **zero** or **one** based on a test that we will perform in the iteration loop. And, we will construct the iteration loop logic such that **M – Q** will be driven to **zero** and therefore $2^{M-Q} \rightarrow 1$. Under these conditions, equation (3-15) can be used to calculate the value of 2^M.

$$(3\text{-}13) \quad f_n = 1 + b_n \cdot 2^{-n}$$

$$(3\text{-}14) \quad 2^M = 2^{M-Q} \cdot \prod_{n=1}^{N} (1 + b_n \cdot 2^{-n}) \qquad \text{where: } Q = \sum_{n=1}^{N} b_n \cdot Lg(1 + 2^{-n})$$

$$\boxed{(3\text{-}15) \quad 2^M = \prod_{n=1}^{N} (1 + b_n \cdot 2^{-n})}$$

The flow chart segment in **Figure 3-4** depicts the needed logic for the iteration loop. Before entering the loop, the value **M** (the fractional part of the input **Z**) is scaled up from 10^6 to 10^9 to align its scaling with that of the **Logs** table. Remember that **M** is confined to the range: $0.0 < M < 1.0$ (but scaled by 10^9). As the loop iterates, a series of **Logs** will be subtracted from **M** in order to drive the value of $M \rightarrow 0.0$. This series of **Logs** represents **Q** in equation (3-14). The value of b_n is determined by testing the current value of **M** to see if is large enough to allow **Logs[n]** to be subtracted from it without going below zero. If so, b_n can be considered to be one and **SX**, the cumulative scaled product, will also be updated. On the other hand, if subtracting **Logs[n]** from **M** would cause **M** to become less than 0.0, b_n can be considered to be zero and **SX** will not be updated.

Before entering the iteration loop, **SX** is initialized to 1.0 (but scaled by 2^{30} for precision). Thus as the loop iterates and $M - \Sigma\, Lg[n]$ approaches zero, **SX** will approach $2^M \cdot 2^{30}$. Note for each loop pass with $b_n = 1$, **SX** will be multiplied by $(1 + 2^{-n})$, per equation (3-15). Also, note that this multiplication can easily be performed with a simple right shift and add operation, the hallmark of Cordic iteration.

Similar to the Cordic Log function, the exponential iteration loop series is *unipolar* so that **M** always approaches 0.0 from the positive side and **M** is never allowed to become negative. Therefore, the error band in driving **M** to zero has a peak to peak range of **Logs[N]**, and is centered at **Logs[N]/2**. Thus, the corresponding value of **SX** will either be correct or low, but never high. As for the Cordic Log function, this error band can be balanced by simply adding a small bias (equal to **Logs[N]/2**) to **M**.

Figure 3-5 shows the outer shell for **Exp2**. The input value **lgx** is assumed to be scaled by 10^6 (just as it would be if it was the output of **XLog2**). Of course it is not necessary that the input value originate from **XLog2**, but input scaling by 10^6 is needed (to provide reasonable precision when computing 2^{lgx}). If $lgx \geq 31.0$, the output **X** is clamped to the maximum positive integer, $MaxInt = 2^{31} - 1$. For all other input values, **lgx** is passed as input to the core support routine named **XP2.Core** shown in **Figure 3-6** and will be discussed next.

$$0.0 < M < 1.0$$
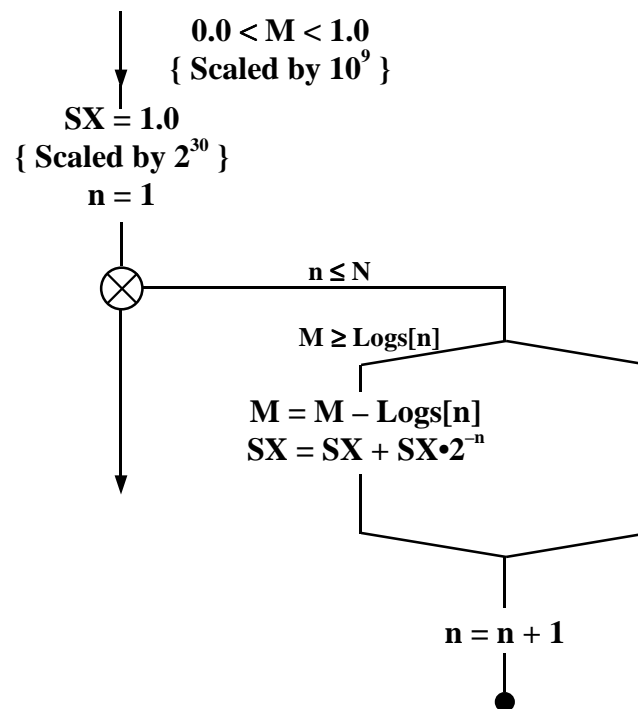$$\{ \text{Scaled by } 10^9 \}$$

$$SX = 1.0$$
$$\{ \text{Scaled by } 2^{30} \}$$
$$n = 1$$

$$n \leq N$$

$$M \geq Logs[n]$$

$$M = M - Logs[n]$$
$$SX = SX + SX \cdot 2^{-n}$$

$$n = n + 1$$

Figure 3-4
Iteration Loop Logic for Exponential Algorithm

Exp2
(lgx,X)

$$X = 2^{lgx/1000000}$$

lgx ≥ 31.0

X = MaxInt

XP2.Core
(lgx, X)

Exit

Figure 3-5
Binary Exponential Function

XP2.Core
(Z, X)

$$X = 2^{Z/1000000}$$
$$0.0 \le Z < 31.0$$

Z < 0

$C = Z/10^6$
$M = Z \bmod 10^6$
$SX = 1.0$
{Scaled by 2^{30} }

M = 0.0

$M = M \cdot 1000 + Logs[N]/2$
{ M now scaled by 10^9 }
n = 1

X = 0

n ≤ N

M ≥ Logs[n]

$M = M – Logs[n]$
$SX = SX + shr(SX,$

C < 30

X = shr(SX, 30 - C)
{ Rounded }

X = SX

n = n + 1

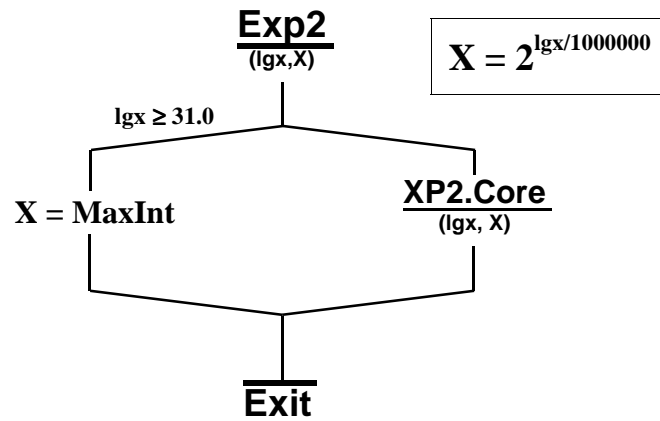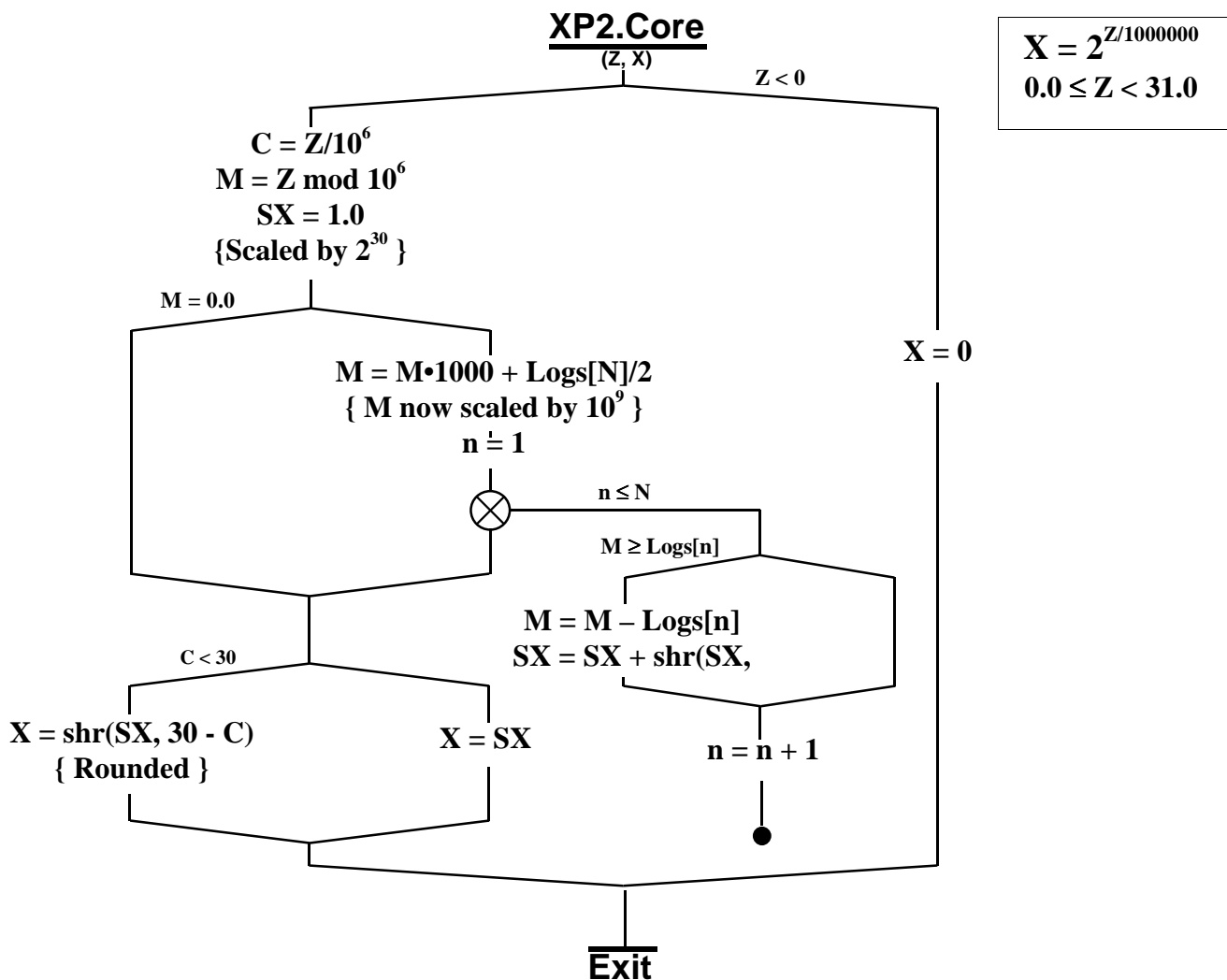**Exit**

Figure 3-6
Binary Exponential Core Function

Since the output **X** will be an unscaled integer, if the input **Z** is negative, **X** is set to 0 since there are no fractions in the integer domain. For all other input $0.0 \le Z < 31.0$, **XP2.Core** splits the scaled input value **Z** into an integer characteristic **C** and a fractional, scaled mantissa **M**. Note that in the actual code, the output variable **X** is used as a temporary for **SX** (the scaled product accumulator) to avoid declaring another local variable. However, in **Figure 3-6**, **SX** is shown as a distinctly separate variable for clarity of discussion. Thus **X** (as **SX**) is initialized to **1.0** (but scaled by 2^{30} for precision). Effectively, **SX** can thus be considered to have its binary point between bits 29 and 30. If **M = 0.0**, the iteration loop can be bypassed (the final output will be an integral power of 2). But if **M ≠ 0.0**, then **M** is in the range of $0.0 < M < 1.0$ and therefore $SX = 2^M$ will be confined to the range $1.0 < SX < 2.0.$ Since **SX** is always less than **2.0**, we don't need to worry about turning on its sign bit as we calculate the chained product $SX \cdot \Pi (1 + 2^{-n})$. As it originates, the scaling of **M** is 10^6 but the **Logs** table has a scaling of 10^9, therefore before entering the iteration loop, **M** is scaled up by another factor of 1000 and then a rounding bias is added. As previously discussed, this bias of **Logs[N]/2** provides balancing of positive and negative errors.

The iteration loop itself was already discussed in conjunction with **Figure 3-4** but, note that after the iteration loop exits, $SX = 2^M \cdot 2^{30}$ and thus $2^M = SX \cdot 2^{-30}$. Also, since we want $X = 2^Z = 2^M \cdot 2^C$, it follows that the desired output value $X = SX \cdot 2^{C-30}$. Further, since $C \le 30$, we can better express **X** as shown in equation (3-16). Note that if **C = 30**, **X = SX** (and since we're using **X** as a temp for **SX,** the actual code needs to do nothing more).

$$\textbf{(3-16)} \quad X = SX/2^{30-C}$$

However, if **C < 30**, in order to convert **SX** to the final **X** value we need to divide **SX** by 2^{30-C} which can be done by simply right shifting **SX** by **30–C** bits. However, if we just right shift, we will obtain truncated results which will add another 'one-sided error' (especially for smaller values of **X**). But, such truncation error can effectively be counterbalanced by using a *rounded* right shift. To round a right shift, we need to inspect the bit that 'falls off the right' as we make the last shift. If it is a one, we need to add 1 to the result. The way this is done in the code is to first right shift **SX** by **29–C** bits, then increment it and finally right shift it one more time.*******

If you compare the actual coding to the flow chart you will notice a few differences. The maximum iteration value **N** is replaced by the symbolic constant named **LogBits**. The log rounding bias **Logs[N]/2** is replaced by the symbolic constant named **LSB2**. These two constants are defined within the data function **BuildLogTable**.

*** NOTE: There is a potential sign bit problem associated with the rounded right shift as described above. If it were possible for **SX** to be **0x7FFFFFFF** (prior to shifting) and **C = 29**, first we would shift right **SX** by **29–C = 0** bits (ie no shift). Then we would increment **SX** before the final right shift and in this case when we increment **SX**, it would become **0x80000000**. The final right shift would then make **SX = 0xC0000000** (which is a big negative number). This problem of course results from the fact that the increment of **SX** turns on the sign bit and the KSP's **sh_right** function propagates the sign (as mentioned previously). However, I don't think that there is any way that this can actually happen because when **M** is the largest it can be, ie **M = 999999**, the value of **SX** that the algorithm computes is **0x7FFFF7FC** which of course is shy of **0x7FFFFFFF** so incrementing it won't turn on the sign bit. However, to be absolutely safe from this potential problem, I simply masked off the sign bit after the last right shift. It's probably not necessary but it's cheap insurance, so why not use it?

3.3.2 Exp2 Error Data

An error analysis was run for the **Exp2** algorithm with two input value ranges and the results are shown in **Table 3-5** and **Table 3-6**. **Table 3-5** resulted from an input **lgx** range from **0 to 16.000000** while **Table 3-6** resulted from the remaining input range from **16.000000 to 31.000000**.

Table 3-5
Exp2 Error Data
lgx = 0 to 16,000,000

Top 5 +/-	Absolute Error		Relative Error	
#1 +	+0.529	@Exp2(15940485)	+33.3%	@Exp2(584962)
#2 +	+0.528	@Exp2(15868014)	+33.3%	@Exp2(584963)
#3 +	+0.528	@Exp2(15956954)	+33.3%	@Exp2(584964)
#4 +	+0.528	@Exp2(15938304)	+33.3%	@Exp2(584965)
#5 +	+0.527	@Exp2(15986297)	+33.3%	@Exp2(584966)
#1 –	-0.53	@Exp2(15993503)	-33.3%	@Exp2(584961)
#2 –	-0.528	@Exp2(15996772)	-33.3%	@Exp2(584960)
#3 –	-0.528	@Exp2(15898850)	-33.3%	@Exp2(584959)
#4 –	-0.528	@Exp2(15915705)	-33.3%	@Exp2(584958)
#5 –	-0.528	@Exp2(15917431)	-33.3%	@Exp2(584957)
Error Summary Data				
Avg Error:	-0.0036		-0.329%	
Avg |Error|:	**+0.25**		**+2.18%**	

Table 3-6
Exp2 Error Data
lgx = 16,000,000
to 31,000,000

Top 5 +/-	Absolute Error		Relative Error	
#1 +	+1020	@Fn(30999316)	+0.0008%	@Exp2(16006950)
#2 +	+1020	@Exp2(30997010)	+0.000798%	@Exp2(16014750)
#3 +	+1020	@Exp2(30995500)	+0.000798%	@Exp2(16008439)
#4 +	+1020	@Exp2(30998509)	+0.000796%	@Exp2(16005635)
#5 +	+1020	@Exp2(30998022)	+0.000794%	@Exp2(16006972)
#1 –	-1030	@Exp2(30999532)	-0.000808%	@Exp2(16002563)
#2 –	-1030	@Exp2(30999792)	-0.000806%	@Exp2(16002541)
#3 –	-1030	@Exp2(30997835)	-0.000805%	@Exp2(16004671)
#4 –	-1020	@Exp2(30998642)	-0.000802%	@Exp2(16005965)
#5 –	-1020	@Exp2(30999129)	-0.0008%	@Exp2(16008659)
Error Summary Data				
Avg Error:	-0.239		-6.39E-8%	
Avg |Error|:	**+49.3**		**+5.3E-5%**	

Before trying to relate the error data to the quality of our algorithm, we need to first understand what the data would look like if the algorithm itself was error free. Even if the function itself was perfect, there will be errors due to the limitations imposed by the input and output number formats. For example, for the input argument range from **lgx = 0** to **lgx = 1,000,000**, the output of a perfect algorithm would be the closest integer value for $2^{lgx/1000000}$. But, consider that **Exp2(0) = 1** and **Exp2(1000000) = 2**. This means that for all the input values between 0 and 1,000,000, **Exp2** can only output **either 1 or 2**. With real (floating point) numbers, **Exp2(0.584963) = 1.5** and **Exp2(1.321928) = 2.5** but with only integer output, we can only output 1, 2, or 3 as the closest integer. Therefore, a perfect algorithm (with our input/output format limitations) would probably output **1** for **lgx <584963** and **2** for **584963 < lgx < 1321928**.

Therefore, for small output values, there are far more input values and therefore, there must be many input numbers that produce the same output! For example, there are more than a half-million input values that will produce an output of 1 and another half-million input values that will produce an output of 2, etc. Therefore, compared with perfect, floating point input and output, **Relative** errors will be large for small numbers.

At the other extreme when **lgx i**s near 31,000,000, the opposite problem exists. With only a 6-digit fraction for the mantissa, there are lots of output integers for which there is no **lgx** value that will produce them. For example, again with a near-perfect algorithm, **Exp2(30999998) = 2,147,480,671** and **Exp2(30999999) = 2,147,482,159**. Note that more than **1400** output integers are 'skipped over' when we make the minimum change we can make in **lgx!** Therefore, we can expect to see high **Absolute** errors for large numbers.

Therefore much of the error that we see in **Tables 3-5 and 3-6** is due mostly to the input/output number format limitations rather than to imperfections in the algorithm itself. And, this makes it rather difficult to assess the algorithm's contribution to the error with any degree of accuracy. In an effort to reduce this 'masking effect' of the input/output error, a 'hybrid test' was performed. For this test, the **XLog2** routine was cascaded with the **Exp2** routine. With this configuration, if everything was perfect, the output integer from **Exp2** should always be the same as the input integer to **XLog2**. When they differ, it will mostly result from algorithmic errors in one or both of the routines.

Of course this kind of test is still not totally independent of the input/output number resolution when the numbers get large. The **XLog2** routine, because it only has a 6-digit manitssa resolution, will by necessity have to generate the same log value for multiple input values. At this level, the **XLog2** routine effectively is **compressing** the input data and the **Exp2** routine tries to uncompress it. But because there is not sufficient precision for the mantissa, the compression is not lossless and the **Exp2** routine therefore cannot make a perfect expansion. However, in spite of these deficiencies, the **log/antilog hybrid** test does provide a somewhat clearer picture of algorithm accuracy. The data taken for this **Log/Antilog Hybrid Test** is presented in **Table 3-7**.

In **Table 3-7**, the first input integer range tested was from **0 to 400,000** and for that range, the output was exactly the same as the input (ie errorless). Input number ranges above **400,000** contained some errors. the absolute value of which increased as the numbers increased. For example, for the number range from **0 to 1,000,000**, the maximum error between input and output was ±**1** with an average absolute error of **0.092**. This means that only about **10%** of the number range from **400,000 to 1,000,000** had an error of ±**1** and the rest were error free. Of course as the number ranges get higher, both the maximum absolute variance and the average absolute variance continue to get larger. However, note that the **Relative Variance, in parts per million**, remains pretty uniform throughout the upper number ranges at an impressive **0.33 ppm**. Moreover this low relative variance includes the **combined** errors of the **XLog2** and **XExp2** routines, which is a pretty convincing indication of the quality of these two algorithms.

Input Integer Range	Max Output Variance ±	Absolute Avg Variance	Relative Avg Variance
0 to 400,000	0	None	None
to 1,000,000	1	0.092	0.112 ppm
to 2,000,000	2	0.31	0.221 ppm
to 5,000,000	6	1.44	0.32 ppm
to 10,000,000	11	2.8	0.33 ppm
to 20,000,000	25	4.9	0.33 ppm
to 50,000,000	64	11.4	0.33 ppm
to 100,000,000	128	24.6	0.33 ppm
to 500,000,000	625	154	0.34 ppm
to 1,000,000,000	1200	250	0.33 ppm
to 2,000,000,000	2500	500	0.33 ppm

Table 3-7
Log/Antilog Hybrid Test Data
Input Integer → XLog2 → Exp2 → Output Integer

3.3.3 exp and Exp10 Functions

The natural **exp** and common **Exp10** antilog functions are derived from **Exp2** by first converting the base of the input argument to the binary base. The fundamental relationship needed to 're-base' an exponent from **base 10** to **base 2** is given by equation (3-17). Similarly, converting an exponent from **base e** to **base 2** is given in (3-18).

$$(3\text{-}17)\ \ 10^X = 10^{Log(2)/Log(2)\bullet X} = (10^{Log(2)})^{X/Log(2)} = 2^{X/Log(2)}$$

$$(3\text{-}18)\ \ e^X = 2^{X/Ln(2)}$$

From equations (3-17) and (3-18), we see that we can re-base the input value by simply dividing it by an appropriate constant. For **Exp10** we divide **lgx** by the constant **Log(2)** and for **exp** we divide **lgx** by the constant **Ln(2)**. As was discussed in **section 3.2.3**, the biggest challenge with re-basing is finding suitable integer ratios that meet the conflicting requirements of staying within the available headroom and also preserving enough precision. The best technique I know of to accomplish this is to split **lgx** into its integer and fractional components, rebase them individually, and then recombine them. Because the integer characteristic can easily be extracted from **lgx** exactly, we can capitalize on the increased headroom and utilize a more accurate ratio for **1/Log(2)** or **1/Ln(2)**.

Basically, the outer shell routine for **Exp10** clamps the output to **MaxInt** if **lgx > 9.331929**. If **lgx** is less than this, it is rebased as **rbx = lgx/Log(2)** and passed to the **XP2.Core** routine. Similarly, the outer shell routine for **exp** clamps the output to **MaxInt** if **lgx > 21.487562** and otherwise rebases it to **rbx = lgx/Ln(2)** before passing it to the **XP2.Core** routine Note that for either **Exp10** or **exp**, the input **lgx** is rebased to the allowable binary exponent range **0.0 ≤ rbx < 31.0**. before **XP2.Core** is invoked. Details of how the exponent is rebased for **Exp10** and **exp** will be discussed next.

For **Exp10**, the characteristic of **lgx** is multiplied by the ratio **106301699/32 = 3,321,928.094** which approximates $1/Log(2) \cdot 10^6$ to 10 digit accuracy. Since the unscaled charactistic of **lgx**, is always less than 10, the minimum headroom available for base conversion is about $238 \cdot 10^6$ which is more than adequate for the above ratio. The scaled mantissa component of **lgx** is always less than 10^6 so the minimum headroom for rebasing is about **2147**. The best ratio within this headroom is **2136/643** which approximates **1/Log(2)** with about 7-digit accuracy. Since the mantissa only has 6-digit output precision, this should be adequate for the scaled mantissa.

For **exp** the characteristic of **lgx** is multiplied by the ratio **70692057/49 = 1,442695.041** which approximates $1/Ln(2) \cdot 10^6$ to 10 digit accuracy. Since the unscaled charactistic of **lgx**, is always less than 22, the minimum headroom available for rebasing is about $102 \cdot 10^6$ which again is sufficient for the above ratio. The scaled mantissa component of **lgx** is always less than 10^6 so the minimum headroom for rebasing is about **2147**. The best ratio within this headroom is **1649/1143** which approximates **1/Ln(2)** with about 7-digit accuracy.

4.0 The Root3 Function

4.1 Introduction

The **Root3(X,r)** function accepts any signed integer **X** over the full, 32-bit range, $-2^{31} \leq X < 2^{31}$ and returns its cube root **r**, scaled by 10^5. Mathematically, the **Root3(X,r)** routine implements the function of **equation (4-1)**.

$$(4\text{-}1) \quad r = 10^5 \cdot X^{1/3}$$

4.2 Algorithm Overview

This algorithm extracts the cube root using 3 phases. Phase 1 handles input normalization and special values. Phase 2 determines the integer part of the cube root and phase 3 approximates the fractional part of the root. A top-level flow chart for the cube root function is shown in **Figure 4-1**.
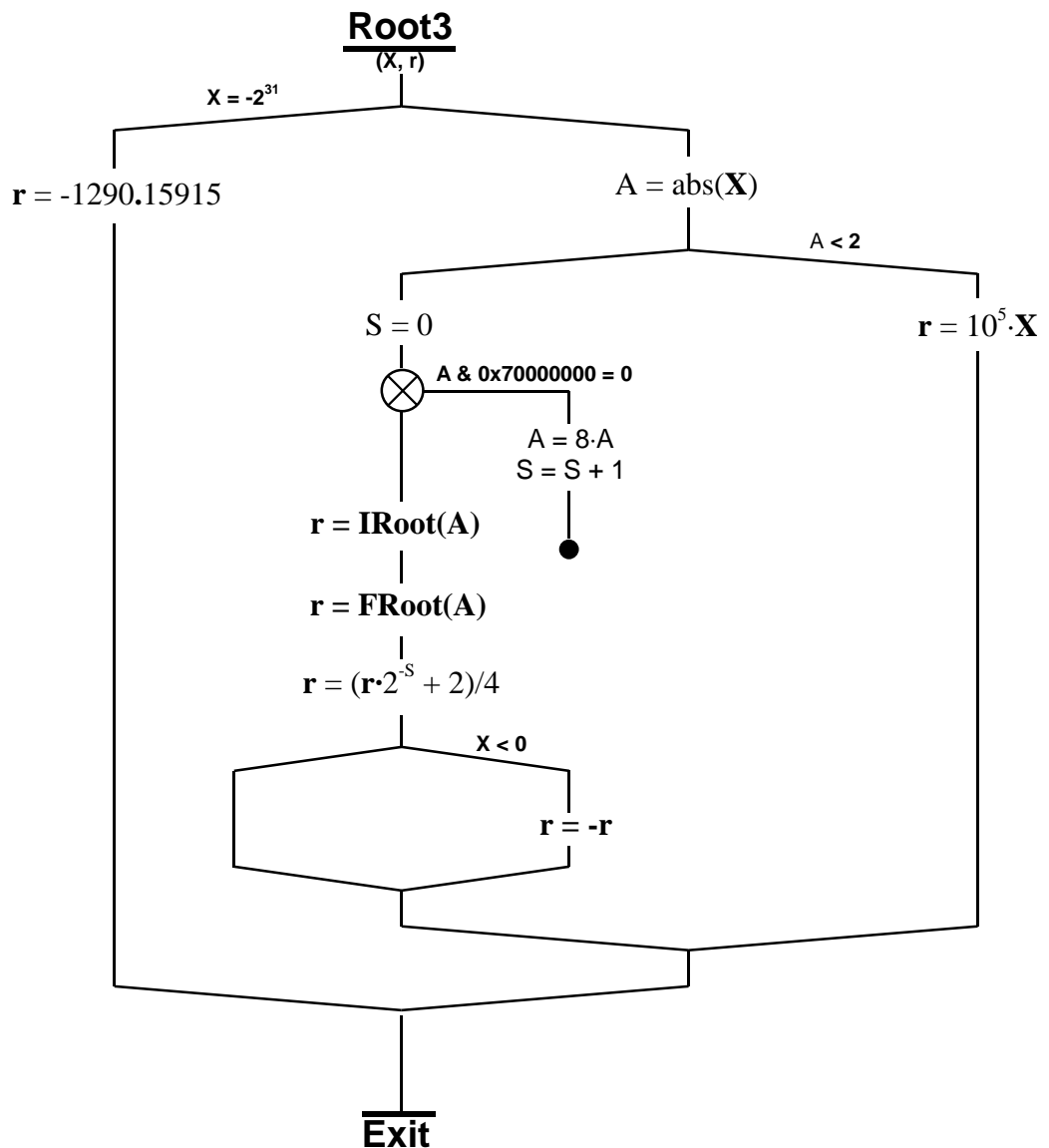
Root3
(X, r)

$X = -2^{31}$

$r = -1290.15915$

$A = abs(\mathbf{X})$

$A < 2$

$S = 0$

$r = 10^5 \cdot \mathbf{X}$

$A \,\&\, 0x70000000 = 0$

$A = 8 \cdot A$
$S = S + 1$

$r = \mathbf{IRoot(A)}$

$r = \mathbf{FRoot(A)}$

$r = (r \cdot 2^{-S} + 2)/4$

$X < 0$

$r = -r$

Exit

Figure 4-1
Root3 Algorithm

Starting at the top of **Figure 4-1**, the first decision tests for the biggest possible negative integer, **-2,147,483,648**. If **X** is that value, the left branch is taken and the pre-calculated cube root of **-1290.15915** is returned for **r**. Of course the actual value returned is the integer **-129015915** and the decimal point is implied (because the output is presumed to be scaled up by 10^5). This left branch test is necessary because **there is no positive counterpart for the biggest negative value**. If **X** is **not** the biggest negative value, its absolute value is placed in the local variable **A**. Thus, for negative values of **X**, the cube root of **abs(X)** is found and then the final result is negated.

After **A** is set to the absolute value of **X**, **A** is tested against 2. If **A < 2**, it means that **X = 0** or **X = ±1**. For any of these 3 values, the cube root of **X** is the value of **X** itself (but of course it must be scaled up by 10^5). Thus **r** is set to $10^5 \cdot X$ in the rightmost branch. For all remaining integer values of **X**, ie

$$-2{,}147{,}483{,}647 \leq X \leq 2{,}147{,}483{,}647$$

the main body of the algorithm is executed. In the main body, the value of **A** is first normalized such that it becomes a value in the hexadecimal range: **0x10000000 ≤ A ≤ 0x7FFFFFFF**. This is accomplished by testing the highest 4 bits and left shifting by 3-bit positions at a time until the highest 4 bits are non-zero. The variable **S** keeps track of how many 3-bit shifts are required to achieve normalization. This value will be used later to denormalize the final cube root value.

Once **A** is normalized, two subfunctions are used to compute the integer and fractional parts for the cube root of **A**. These subfunctions are shown in **Figure 4-1** as **IRoot3** and **FRoot3** respectively. The details of these subfunctions will be discussed subsequently. However, as coded in the actual **Root3** routine, on exit from the **FRoot3** procedure, the value of **r** will be the cube root of **A**, scaled by $4 \cdot 10^5$. The additional factor of 4 is used to preserve more interim precision prior to denormalization and rounding. Denormalization requires that the cube root be right-shifted by **S** bit positions and rounding is performed by half adjusting before the final division by 4 (actually performed by an additional 2-bit right shift). After denormalization and rounding, the orignal value of **X** is examined and if it is negative, the value of **r** is negated (since the cube root of a negative number must also be negative).

4.3 The IRoot3 Subfunction

Over the normalized range of **A**, the integer part **I** of the cube root of **A** will be a value in the range of **645 ≤ I ≤ 1290**. And, for every such root there exists a pair of consecutive integers, I_1 and I_2, that are related as shown in the equations and inequalities of (4-2) through (4-4).

$$(4\text{-}2) \quad I_2 = I_1 + 1$$
$$(4\text{-}3) \quad I_1 \leq r$$
$$(4\text{-}4) \quad I_2 > r$$

The purpose of the subfunction **IRoot3** is to find one of these bracketing integers I_1 or I_2. One way of doing that is depicted in the flowchart of **Figure 4-2**. This sub-function starts with an initial estimate or 'guess' for the cube root and then uses Newton's method to iteratively 'refine' the initial guess until either I_1 or I_2 is found. The criterion for loop exit is to stop iterating when two successive loop passes yield values for **r** that differ from each other by less than two (ie by zero or one). While a possible **exception** to this will be discussed in **Section 4.3.1**, for now we will assume that when this condition is achieved, **r** will be either I_1 or I_2 as desired.

IRoot3
(A,r)

r = Guess(A)
r1 = 0

abs(r - r1) < 2

r1 = r
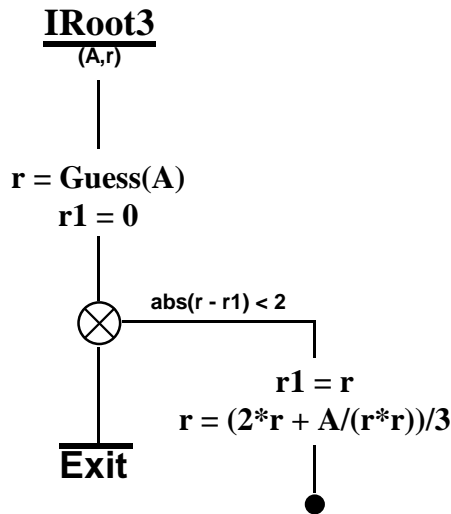r = (2*r + A/(r*r))/3

Exit

Figure 4-2
IRoot3 Sub-Function

One rather crude way to make our initial guess for **r** is to start with the cube root of the normalized range mid-point. Since the median normalized radicand is **A = 0x48000000 = 1,207,959,552** we can use an initial guess or **r = 1065**. The subfunction of **Figure 4-2** can be put in a loop and tested for every integer value over the normalized range to determine the maximum number of Newton iterations needed. If we always start with a guess of **r = 1065**, the maximum number of iterations required is 4. To reduce this number we could either use a faster-converging iteration method (such as Halley's method) or we could start with a more accurate guess. While using Halley's method has the potential to reduce the number of iterations, it also adds more calculations per loop pass. So if we can find a fast way to make better 'guesses' for **r**, it may be preferable.

One way we can make a better initial guess (without a lot of additional calculations to slow us down) is to use a table of starting guesses and choose the starting value for **r** based on the value of **A**. For example, we could divide the normalized number range of **A** into 7 equal zones and use the cube root of each zone's median value as our starting estimate for **r**. This strategy is depicted in **Table 4-1**.

Zone	Normalized Input Range	Median	Guess
1	0x10000000 to 0x1FFFFFFF	0x18000000	738
2	0x20000000 to 0x2FFFFFFF	0x28000000	876
3	0x30000000 to 0x3FFFFFFF	0x38000000	979
4	0x40000000 to 0x4FFFFFFF	0x48000000	1065
5	0x50000000 to 0x5FFFFFFF	0x58000000	1139
6	0x60000000 to 0x6FFFFFFF	0x68000000	1204
7	0x70000000 to 0x7FFFFFFF	0x78000000	1263

Table 4-1
7-Zone Initial Guess

To access such a table of guesses is simple because all we have to do is use the top four bits of **A** as an index into the table. Only 3 of these bits are significant (since the sign bit is always zero) so when we shift **A** to the right by 28 bit positions, we will obtain a value from 1 to 7. And, if we want the table to be zero-based (as requrired by the KSP), all that would be necessary to get our starting estimate for **r** would be the following simple line of code (assuming that our table array is named **R3Tbl**).

$$r \; := \; R3Tbl[sh\_right(A,28) - 1]$$

Using **Table 4-1** to obtain our initial guess for **r** reduces the maximum iterations from 4 to 3 but more interestingly, zones 7 and 6 never exceed 2 iterations. From this information, we can conclude that the lower normalized numbers converge slower than the higher numbers and this suggests that we might benefit from sub-dividing the lower zones. By splitting zones 3, 4, and 5 into two sub-zones each and by splitting zones 1 and 2 into four sub-zones each, no more than two Newton iterations are required for any input integer. And, since the iteration loop exit criterion is that **r** doesn't change by more than one for the last two passes, **only a single Newton iteration is necessary** to obtain I_1 or I_2.

The minimum number of zones and corresponding initial guesses required to achieve this is 16 as shown in **Table 4-2**. However, to access such a compact table containing single, dual, and quad zone sub-divisions requires more code with several compares and shifts involved. This would both increase the number of code lines and the execution time as well. However, we can easily avoid this complication by simply using a larger table with some redundant entries.

Zone	Normalized Input Range	Median	Guess
1a	0x10000000 to 0x13FFFFFF	0x12000000	671
1b	0x14000000 to 0x17FFFFFF	0x16000000	717
1c	0x18000000 to 0x1BFFFFFF	0x1A000000	758
1d	0x1C000000 to 0x1FFFFFFF	0x1E000000	795
2a	0x20000000 to 0x23FFFFFF	0x22000000	829
2b	0x24000000 to 0x27FFFFFF	0x26000000	861
2c	0x28000000 to 0x2BFFFFFF	0x2A000000	890
2d	0x2C000000 to 0x2FFFFFFF	0x2E000000	917
3a	0x30000000 to 0x37FFFFFF	0x34000000	956
3b	0x38000000 to 0x3FFFFFFF	0x3C000000	1002
4a	0x40000000 to 0x47FFFFFF	0x44000000	1045
4b	0x48000000 to 0x4FFFFFFF	0x4C000000	1084
5a	0x50000000 to 0x57FFFFFF	0x54000000	1121
5b	0x58000000 to 0x5FFFFFFF	0x5C000000	1156
6	0x60000000 to 0x6FFFFFFF	0x68000000	1204
7	0x70000000 to 0x7FFFFFFF	0x78000000	1263

Table 4-2
16-Zone Initial Guess

This is the approach used for the library implementation of **Root3**, since I felt execution time and code compactness were more important than a modest table size increase. By simply sub-dividing the orignal 7 zones into quarters (which then requires a 28-element table) we can easily obtain our initial guess as a function of **A** with the following single line of code.

$$r := R3Tbl[sh\_right(A,26) - 4]$$

By using Halley's method, which converges faster than Newton's, the number of starting guesses can be reduced from 16 to 8 and thus our table size can easily be reduced by a factor of 2. However, Halley requires more calculations and, because of integer arithmetic limitations, a lot more scaling and ordering 'tricks' are also required to maintain enough precision to make it work. The end result is that several more code lines are needed and the execution time increases quite a bit. Therefore, it was decided to stick with the simpler Newton iteration because its only penalty is that a somewhat larger table is needed to support it.

4.3.1 Exit Criterion Revisited

As mentioned at the beginning of **Section 4.3**, there is a possible complication associated with the Newton iteration criterion. Remember that empirical tests of all normalized input values revealed that (as long as we use the initial guesses from **Table 4-2**) no more than two Newton iterations are required by any zones. And, up until now we have assumed that when two successive iterations yield values that are only one apart, the two values represent I_1 and I_2 as defined by equations (4-2), (4-3), and (4-4). However, it seems at least intuitively possible that for some input values, the first and second iteration might both be above or both be below the root (rather than straddling it). If that should be the case, then, while the 2nd iteration would still produce either I_1 or I_2, the first iteration **would not**. While there might be some sound mathematical reason why this could never happen, rather than straining my poor old brain with it, I decided instead to prove it by further empirical testing. Therefore, I ran all possible integers through a single Newton iteration and verfied that no input value ever caused the **IRoot** process to output anything but I_1 or I_2.

4.4 The FRoot3 Subfunction

There are several techniques that can be used to approximate the fractional component of the cube root (after we have obtained the integer part from the **IRoot3**) sub-function. One of the easiest ways is to utilize the simple binomial approximation given in equation (4-5). To see how this approximation can be utilized, let's denote the integer root from **IRoot3** by the letter **I** and the desired fractional component by the letter **F** (for the actual library implementation the variable **r** is used for both the integer and final root value but here we'll use **I** and **F** for clarity).

Equation (4-6) expresses the 'remainder' (or error if you like) after subtracting the cube of the integer root from the radicand **A**. Since **IRoot3** provides either I_1 or I_2, I^3 could be less or more than **A** so **Rem** (as well as **F**) are signed values. Equation (4-7) simply cubes the composite root and equates it to **A** as our starting point. The next step in (4-7) factors out I^3, to get the binomial into the proper form of $(1 + b)^3$. Now, as long as **F/I** is much smaller than 1, the approximation of (4-5) can be used as shown in the last two steps of (4-7). And, solving for **F**, we obtain equation (4-8). Finally, we can combine **I** and **F** after scaling by 10^5 to obtain the final formula for the scaled **root** as shown in equation (4-9). Note that if **Rem < 0**, it means **I > r** so that **F** will be a negative fractional value.

(4-5) $(1 \pm b)^n \approx 1 \pm nb$ | if b << 1

(4-6) $Rem = A - I^3$

(4-7) $A = (I + F)^3 = I^3(1 + F/I)^3 \approx I^3(1 + 3{\cdot}F/I) = I^3 + 3{\cdot}I^2{\cdot}F$

(4-8) $F = (A - I^3)/(3{\cdot}I^2) = Rem/(3{\cdot}I^2)$

(4-9) $root = 10^5{\cdot}I + 10^5{\cdot}Rem/(3{\cdot}I^2)$

The accuracy of equation 4-9 depends on several factors, one of which is how small **F/I** really is compared to one. Note that **IRoot3** provides either $I = I_1$ or $I = I_2$, so in general **abs(F)** could almost be as high as 1.0. However, if we modify **IRoot3** so it will always provide the **closest** integer to the actual root, then the maximum absolute value of **F** can be cut in half and thus **abs(F)** $\leq$ **0.5** which will definitely enhance precision. Moreover, if we keep **F** smaller, we can use more scaling during the interim calculations to minimize divsion truncation errors. By embellishing **IRoot3** to provide the closest integer to the root, the binary approximation is overall, remarkably accurate.

However, if we modify **IRoot3** to find the closest integer to the root, it adds quite a few code lines and several otherwise unnecessary multiplications which slows down the overall execution time for the routine. Yet without finding the closest root, the precsion of the fractional approximation is seriously degraded. Obviously, it would be nice if we could avoid the necessity of finding the closest root but also not suffer any material loss of precision. A careful inspection of equation (4-9) reveals that it is mathematically identical to what we would get if we scaled **I** by 10^5 and then performed another **Newton** iteration. This suggests that we might be able to improve things by scaling **I** and then performing a **Halley** iteration instead of a **Newton**. Since Halley converges more rapidly we might be able to get more precision (under the same conditions) or about the same precsion but over a wider domain. This latter posibility leads us to the hopeful expectation that we might be able to eliminate the need to find the closest integer and still maintain sufficient precision. And indeed, this does turn out to be the case.

Now, I'd like to say that it was the foregoing 'train of thought' that led to using this technique for the **Root3** library routine. However, credit for the idea of using a Halley iteration to derive the fractional part of the root has to go to **Joris** on the **VI Forum**. Only **after** he suggested this idea did I tumble to the fact that my binomial approximation was really a Newton iteration in disguise. But thanks to Joris, the final version of **Root3** is now faster and more compact and yet has nearly identical precision with the prior binomial version. In fact, if one is willing to accept a modest speed reduction (on the order of 10% slower), the Halley approximation can be coaxed to provide even superior precision to that of the closest-integer binomial algorithm. However, the current library version of **Root3** was chosen for speed and compactness while still maintaining more than adequate overall accuracy (as can be seen by referring to **Section 4-6**).

4.5 Library Implementation

Refer to the top-level flow chart of **Figure 4-1** and we will now discuss the actual library implementation of the **IRoot3** and **FRoot3** subfunctions. Each of these subfunctions is implemented with two lines of code as follows:

$$r := \text{R3Tbl[sh\_right(A,26)} - 4]$$
$$r := (2*r + A/(r*r))/3$$
IRoot3

$$M := A/r - r*r$$
$$r := 400000*r + 400000*M/(3*r + M/r)$$
FRoot3

The first line obtains an appropriate 'guess' for the integer root from the 16-value table in the array named **R3Tbl**. This process has already been discussed in **Section 4.3**. The second line refines this 'guess' in **r** by performing a single Newton iteration. No special scaling was required to accomplish the desired mission of obtaining $r = I_1$ or $r = I_2$ as discussed in **Section 4.3**. Therefore, after these first two lines of code execute, the variable **r** contains the integer part of the root (ie either I_1 or I_2).

The last two lines scale **r** by $4 \cdot 10^5$ and derive the Halley approximation for the fractional part of the root. Note that this scaling is 4 times the required 10^5 final output scaling. This extra scaling as well as other details of the Halley iteration will be discussed next.

One way of expressing the formula for a Halley iteration is given in equation (4-10). In this equation, r_n represents the current value of r (the integer root) and r_{n+1} represents the 'final' value of the root r (including the fractional part). First note that the term $A - r_n^3$ is the remainder (or error in the integer cube). Thus, if we denote this remainder by **Rem**, we can re-write equation (4-10) as shown in (4-11). Now if we factor r_n from the denominator we can express equation (4-11) as in (4-12).

$$(4\text{-}10) \qquad r_{n+1} = r_n + (A - r_n^3)/[3 \cdot r_n^2 + (A - r_n^3)/r_n]$$

$$(4\text{-}11) \qquad r_{n+1} = r_n + \text{Rem}/[3 \cdot r_n^2 + \text{Rem}/r_n]$$

$$(4\text{-}12) \qquad r_{n+1} = r_n + (\text{Rem}/r_n)/[3 \cdot r_n + \text{Rem}/r_n^2]$$

$$(4\text{-}13) \qquad M = \text{Rem}/r_n$$

$$(4\text{-}14) \qquad r_{n+1} = r_n + M/(3 \cdot r_n + M/r_n)$$

Next, we define a new variable M as shown in equation 4-13 and after substituting this into equation (4-12) we obtain equation (4-14). Except for scaling considerations, equation (4-14) is of the form we want for our implementation. By comparing equation (4-14) with code lines 3 and 4 shown at the beginning of Section 4.5 we can see that both terms of (4-14) are merely multiplied by 400000. Comparing (4-11) with (4-9) we see that the only difference between the Halley and Newton iteration formulae is that Halley has an additional **Rem/r_n** term in the denominator

To preserve reasonable accuracy in the result, we need to minimize the truncation error in the first-order division of the numerator M. The denominator division of M/r is a second-order term and much less critical, so we won't need to scale it. But we would like to multiply the numerator M by as big a value as we can before dividing it by $(3 \cdot r + M/r)$. Of course the current integer value of r must be scaled by the same amount so we can add it to the scaled fraction. Now, the question arises, how big can we make our scale factor? What we must insure is that the multiplication of our scaling factor by M must not exceed the maximum integer value or arithmetic overflow will result.

Therefore to determine the maximum value of our scaling constant we need to know the maximum value of M. There are various ways of arriving at a ballpark maximum for M but again, since the integer number set is finite (even though it is rather large at 2+ billion), I thought it best to once again resort to empirical methods. By running all the numbers through the **IRoot3** subfunction and then computing each corresponding value of M the maximum range for M was determined and is shown as expression (4-15).

$$(4\text{-}15) \qquad -2858 \leq M \leq +3789$$

From (4-15) and knowing the maximum integer value we can compute that our scaling factor can be no higher than **566,767**. Since we want our final output scaling to be 10^5, we should pick a convenient integer multiple of this (preferably a power of two so it can be removed by shifting instead of with a division). The highest value that satisfies these conditions is $2^2 \cdot 10^5$ which is what is used for the Halley calculation scaling.

Therefore, after executing the two lines of code for the Halley iteration, r contains the cube root of A scaled by $4 \cdot 10^5$. Refering back again to **Figure 4-1**, we can now understand the denormalization and rounding process. Since after normalization, $A = |X| \cdot 2^{3S}$, and since $r = 4 \cdot 10^5 \cdot A^{1/3} = 4 \cdot 10^5 \cdot |X|^{1/3} \cdot 2^S$, to finish up we must first multiply r by 2^{-S} (which is the same as shifting it right by S bit positions). Finally, we add 2 and then shift right 2 more times (effectively dividing by 4 with half-adjust rounding). After doing that we will have the rounded result we want, namely $r = 10^5 \cdot |X|^{1/3}$. The final step of course is to negate r if the input value of X is negative.

4.6 Root3 Error Data

The completed algorithm was run for the entire (non-trivial) positive number range from 2 to 2,147,483,647 and our usual set of statistics was collected. **Table 4-3** shows the tabulated error data. The data pretty much speaks for itself. The average absolute error magnitude indicates that the 5 fractional digits are typically correct out through the 4th digit with maximum absolute errors around 0.0005. The average Relative error magnitude is about one part in 10 million with a maximum around 2 parts per million.

For approximately a 10% speed penalty, these error statistics can be improved by roughly a factor of two by simply replacing the two lines of code for the Halley iteration with the following two lines.

$$M := 1000*(A/r - r*r) + 1000*(A \bmod r)/r$$

$$r := 400000*r + 400*M/(3*r + M/r/1000)$$

With the above code in place, the average absolute error magnitude drops to about 0.00006 as opposed to 0.00012 for the current routine. The average relative error magnitude drops to about 7 parts per 100 million (as opposed to about 10 parts per 100 million for the current implementation).

Top 5 +/-	Absolute Error		Relative Error	
#1 +	+0.000498	@X = 300760323	+0.000237%	@X = 5
#2 +	+0.000497	@X = 300760324	+0.000137%	@X = 38
#3 +	+0.000496	@X = 300760325	+0.000117%	@X = 49
#4 +	+0.000496	@X = 300760326	+0.000114%	@X = 70
#5 +	+0.000495	@X = 300757647	+0.000113%	@X = 53
#1 –	-0.000519	@X = 269589365	-0.000218%	@X = 10
#2 –	-0.000519	@X = 269592595	-0.000205%	@X = 26
#3 –	-0.000519	@X = 269595825	-0.000199%	@X = 13
#4 –	-0.000519	@X = 269599055	-0.000184%	@X = 9
#5 –	-0.000519	@X = 270856197	-0.000151%	@X = 21
	Error Summary Data			
Avg Error:	-0.0000921		-0.0000102%	
Avg |Error|:	**+0.000117**		**+0.0000130%**	

Table 4-3
Root3 Error Stats
**X = 2 to 2,147,483,647**

5.0 THE ATFADE FUNCTION

5.1 Introduction

The **ATFade** function is intended to be incorporated into a host script that requires overall volume control to be handled by a MIDI CC acting as an Audio-Taper Fader. This function is called with 3 parameters as follows: **ATFade(cv,rng,atn)**. **'cv'** is an input value, **0..127** from the desired MIDI CC. **'rng'** is an input value, **0..100%** from a user adjustable Range control and **'atn'** is the output audio-contoured attenuation in mdb.

The electronic equivalent of K2's MIDI CC control is shown in **Figure 5-1**. The **ATFade** function operates in a similar way except that with **ATFade**, the contouring of the MIDI CC is more like that of a traditional 'audio' fader whereas in K2, the contour of the potentiometers is simply linear. So, when you assign a CC to modulate the amplifier in K2, some of the most-needed volume levels are cramped together in a small region of the CC's travel. **ATFade** on the other hand, spreads the most-needed attenuation range of **0 to -25 db** across the top 60% of the CC's travel (just like a real mixing-board fader). In addition, by setting a suitable value for the Range parameter (equivalent to K2's Intensity slider), you can narrow the total range of the CC by bringing up the bottom end. This is equivalent to removing the lower end of the fader scale and then stretching the remainder back to the full size (see **Figure 5-2**). With such tapers, you can easily control volume expressively.
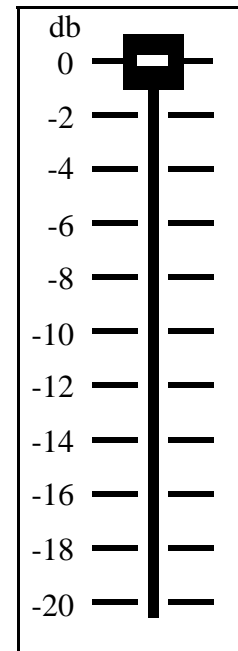
Figure 5-1

Electronic Equivalent For K2's Positive Modulation
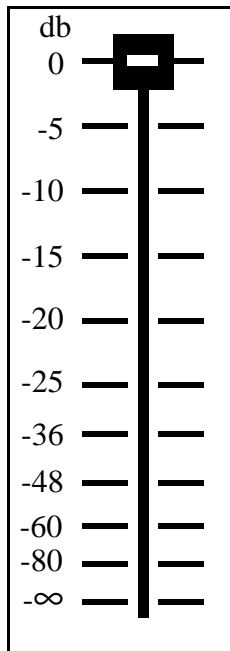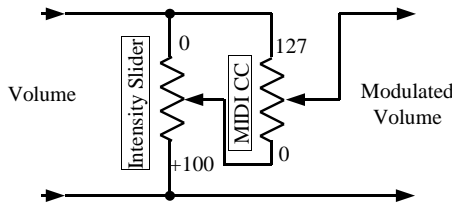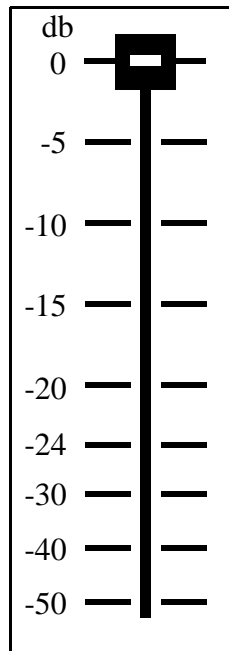
Figure 5-2

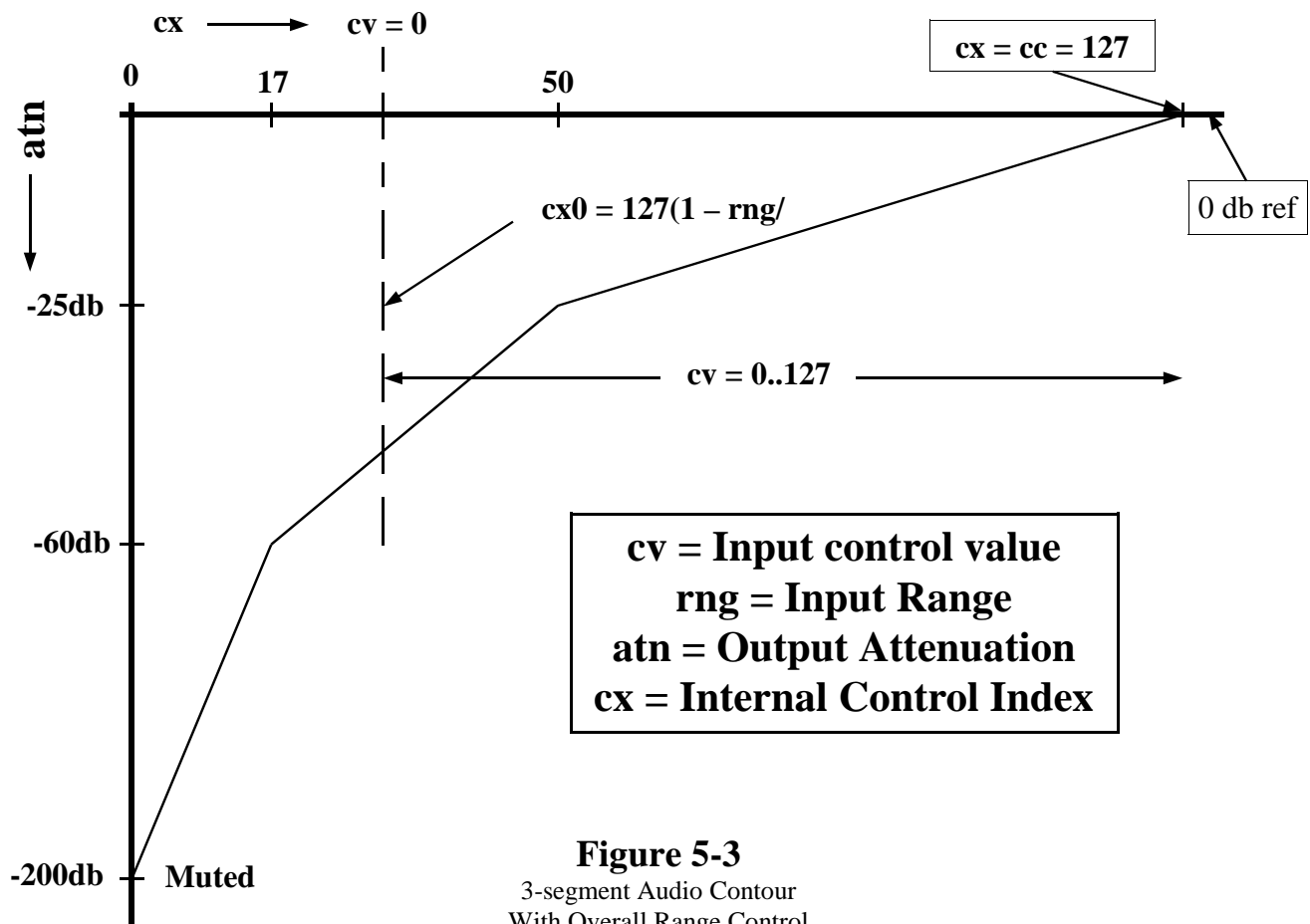Audio Contours
at various 'Range' settings

5.2 ATFade Math

When the **rng** input is is **set to 100%**, the attenuation output versus the **cv input** is a 3-segment, linear-db curve as depicted in **Figure 5-3**. The normal 'working zone' from **0 to -25db**, is controlled by the **cv = 127 to 50** range. This is about 61% of the full 'fader' travel. The quiet or 'soft zone' from **-25db to -60db** is controlled by the range of **cv = 50 to 17**, which is about 26% of the full 'fader' travel. And, the 'fade-out' zone from **-60db to -200db** (where -200db is for all practical purposes is **Muted**), is covered by the lowest 13% of the full 'fader'travel of **cv = 17 to 0**. Thus, with **Range = 100%**, the fader's audio control characteristic is as depicted by the scale shown in the leftmost drawing of **Figure 5-2**.

Internally, the output attenuation is controlled by a local control variable named **cx**. Specifically, the relationship between **cx** and the input **cv** is given by equation (5-1). Note that when **rng = 100%**, **cx = cv** and when **rng = 0%** that **cx = 127**.

$$(5\text{-}1)\quad cx = 127 + (cv - 127)\cdot rng/100$$

Effectively, the value of **rng**, restricts the full range of attenuation controlled by **cv** by removing some of the bottom end. This is depicted graphically in **Figure 5-3** by the dashed vertical line labeled on top as **cv = 0**. This **cx0** line (which is a function of the **rng** value) determines the value of **cx** (when **cv = 0**).

cx ⟶ cv = 0

cx = cc = 127

0   17   50

atn

0 db ref

cx0 = 127(1 – rng/

cv = 0..127

-25db

-60db

cv = Input control value
rng = Input Range
atn = Output Attenuation
cx = Internal Control Index

-200db   Muted

Figure 5-3
3-segment Audio Contour
With Overall Range Control

The attenuation formulae are given in equations (5-2), (5-3), and (5-4) for the 3 segments of the curve. Equation (5-2) gives the attenuation for the normal 'working' zone, equation (5-3) gives the attenuation for the 'soft zone' and equation (55-4) gives the attenuation for the 'fadeout' zone.

$$(5\text{-}2) \quad \mathbf{atn = 25 \cdot (cx - 127)/77} \qquad \mathbf{50 < cx < 127}$$
$$(5\text{-}3) \quad \mathbf{atn = \text{-}60 + 35 \cdot (cx - 17)/33} \qquad \mathbf{17 < cx < 50}$$
$$(5\text{-}4) \quad \mathbf{atn = \text{-}200 + 140 \cdot cx\ /17} \qquad \mathbf{0 < cx < 17}$$

Equations (5-2), (5-3), and (5-4) give the attenuations in db. However, for the actual KSP implementation we want the attenuation output in mdb and we should also round the divisions for a smoother response. Since all the divisors are odd, we need to double the numerator and denominator of each expression so we can use an integer half-adjust value for rounding. Thus, the KSP equations for **atn** are as given in (5-5), (5-6), and (5-7).

$$(5\text{-}5) \quad \mathbf{atn = [50000 \cdot (cx - 127) - 72]/154} \qquad \mathbf{50 < cx < 127}$$
$$(5\text{-}6) \quad \mathbf{atn = \text{-}60000 + [70000 \cdot (cx - 17) + 33]/66} \quad \mathbf{17 < cx < 50}$$
$$(5\text{-}7) \quad \mathbf{atn = \text{-}200000 + (280000 \cdot cx\ + 17)/34} \qquad \mathbf{0 < cx < 17}$$

Take a look now at the code for **ATFade**. The routine begins by computing **cx** as shown in equation (5-1). Then a 3-way case statement is used to select which zone **cx** is in. Finally, **atn** in mdb is calculated using the appropriate equation selected from (5-5), (5-6), or (5-7) with **-200000** in (5-7) being replaced by the constant named **Muted**. Note that this constant is also made available to the host script if either **ATFade**, **Get_db,** or **VR_to_mdb** is called by the host (just as the constant named **Ang90** is made available to the host if any of the Trig functions are called). If the host script uses **Muted** instead of **-200000 mdb** the host will then be self correcting if the library value for **Muted** is ever changed.

6.0 Format Conversion Routines

6.1 Introduction

These utility routines are provided to convert between different units or between engine parameters and their corresponding actual parameter values. There is nothing you can do with these routines that you can't do with the basic Math Library Functions already discussed in sections 1 to 5. However, optimum scaling, ordering of operations, and other precision-preserving details can sometimes require considerable effort to code. Therefore, it should save a lot repeated effort if I simply include optimized versions for the most commonly needed conversion functions by making them a standard part of the library.

Each format conversion routine is based upon some mathematical formula that relates the input to the desired output. In some cases, such as conversion of Volume Ratio (to its equivalent in mdb), the mathematical function is well known. However, many of the non-linear relationships between engine parameters and what they control have not been disclosed by NI so most of these conversion formulae had to be 'reverse engineered' by observation, measurement, and 'curve-fitting' techniques. The resultant functions track Kontakt's behavior quite accurately and most of the deviation you might encounter will be due to errors introduced by the all-integer arithmetic limitations of the KSP. But, keep in mind that the reverse-engineered formulae may also differ from what Kontakt is actually using internally; only NI knows for sure ;-)

6.2 Envelope Time

These routines are used to convert envelope attack, decay, or release time to their equivalent engine parameters and vice versa. **Envelope Attack** time ranges from 0 to 25,000 ms while **Envelope Decay** and **Release** time both range from 0 to 15,000 ms. **AtkTime_to_ep** can be used to convert from the desired attack time in msec to the corresponding engine parameter from 0 to 1,000,000. The routine named **DRTime_to_ep** can be used to convert from the desired **Decay** or **Release** time in msec to the corresponding engine parameter. The inverse functions are named **ep_to_AtkTime** and **ep_to_DRTime** respectively. These functions convert from the engine parameter to its equivalent time in msec. The reverse-engineered formulae used to relate envelope time, **T**, in **msec** to the corresponding engine parameter, **N**, are:

$$\textbf{(6-1)} \quad N = K \cdot [Lg(T + 2) - 1]$$

$$\textbf{(6-2)} \quad T = 2 \cdot [2^{(N/K)} - 1]$$

where K = 77683 for Attack Time or K = 73477 for Decay/Release Time

Both **AtkTime_to_ep** and **DRTime_to_ep** call on the support routine named **Time2ep** while the inverse functions call on the support routine named **ep2Time**. For **eq (6-1)**, the Library routine **Log2** is used but since it returns the log scaled by 10000, **K·Log2** has to be divided by 10000 as shown in **eq 6-3**. Note that the division is performed in two stages first dividing by 5 and then by 2000. This ordering of the operations is necessary to prevent overflow when **T** is at max. Even as it is, for **AtkTime = 15000** msec, **Log2(15002) = 138729** which when multiplied by **K/5** yields **2,155,376,982** which exceeds the positive integer limit of 2,147,483,647. However, it doesn't exceed the 'unsigned' 32-bit integer limit and the next arithmetic operation will be to subtract **2000·K** from it. This will reduce the value below 31 bits once again before division by 2000. However, if we didn't first divide **K** by **5**, the **K·Log2** product would exceed even the unsigned integer limit by a factor of more than 2 and this could not be contained in 32 bits without overflow. This of course would not be corrected by a subsequent subtraction. The rightmost form of **equation 6-3** is the way **Time2ep** is coded (but with the addition of half-adjust rounding).

$$\textbf{(6-3)} \quad N = [K \cdot Log2(T + 2)]/10000 - K = [K/5 \cdot Log2(T + 2) - 2000 \cdot K]/2000$$

The inverse function of **equation (6-2)** is implemented by **ep2Time** using the Library routine **Exp2**. However, **Exp2** requires that its argument be scaled by **1,000,000** so we have to rewrite **eq (6-2)** as shown in **eq (6-4)**. Note that we have to split the multiplication by **1000000** into two factors of **1000** each to avoid arithmetic overflow. The rightmost form of **eq (6-4)** is the way **ep2Time** is coded.

$$\textbf{(6-4)} \quad \textbf{T} = \textbf{2·Exp2(1000000·N/K) - 2} = \textbf{2·Exp2(N·1000/K·1000) - 2}$$

6.3 EQ Parameters

In a multi-channel EQ, there are three parameters for each channel (Frequency, Bandwidth, and Gain) that can be controlled from a corresponding set of engine parameters. The main library routines for this purpose are named **EqFreq_to_ep**, **EqBW_to_ep**, and **EqGain_to_ep** while the inverse functons are named **ep_to_EqFreq**, **ep_to_EqBW**, and **ep_to_EqGain** respectively. The 'reverse-engineered' formulae that relate EQ Frequency, **F**, in Hz to the corresponding engine parameter, **N**, are given in **equations (6-5)** and **(6-6)**. In these equations, the frequency range is from 20Hz to 20,000Hz.

$$\textbf{(6-5)} \quad \textbf{N = K·Lg(F/20)} \quad \text{where } \textbf{K = 100343}$$

$$\textbf{(6-6)} \quad \textbf{F} = \textbf{20·2}^{\textbf{(N/K)}} = \textbf{2}^{\textbf{(N/K) + Lg(20)}} = \textbf{2}^{\textbf{(N/K) + 4.321928}}$$

Equation (6-5) is implemented using the Library's **Log2** function which returns the desired log but scaled up by 10,000. Therefore, we can re-write **eq (6-5)** as shown in **eq (6-7)**.

$$\textbf{(6-7)} \quad \textbf{K/10000·Log2(F) - K·Lg(20) = [2338·Log2(F)]/233 - 433675}$$

Since **Log2(20000)** = **142877**, we cannot multiply by **K** before dividing by **10000** without arithmetic overflow for the higher **F** values. However, since **K/10000 = 10.03430**, we can replace it with **2238/233 = 10.03433476** which is accurate to 6 significant figures. This allows us to first multiply **Log2(F)** by **2238** (which is safely within the positive integer range even at $\textbf{F}_{\textbf{max}}$). The rightmost part of **equation (6-7)** is the way the **EqFreq_to_ep** function is coded (except for the addition of half-adjust rounding).

Equation (6-6) is implemented using the **Exp2** Library function which requires its argument to be scaled up by a factor of 1,000,000. Therefore, we can re-write **eq (6-6)** as shown in **eq (6-8)**

$$\textbf{(6-8)} \quad \textbf{F} = \textbf{Exp2(10}^{\textbf{6}}\textbf{·N/K + 4321928) = Exp2(1166·N/117 + 4321928)}$$

Since $\textbf{10}^{\textbf{6}}\textbf{·N}$ can be as much as $\textbf{10}^{\textbf{12}}$, we can't multiply by $\textbf{10}^{\textbf{6}}$ without risking arithmetic overflow. However, $\textbf{10}^{\textbf{6}}\textbf{/K} = \textbf{9.965817247}$ can be replaced by the ratio **1166/117 = 9.965811966** which is accurate to six significant figures, and, **1166·N** will not cause arithmetic overflow (even with $\textbf{N} = \textbf{10}^{\textbf{6}}$). Therefore, the rightmost part of **equation (6-8)** is the way the **ep_to_EqFreq** is coded (with the addition of half-adjust rounding).

For **EQ Bandwidth** and **Gain**, Kontakt seems to use a simple linear relationship with the corresponding engine parameters as shown in **equations (6-9) to (6-12)**. In these equations, **B** is the **Bandwidth** in octaves scaled up by 10 (**0.3 ≤ B ≤ 3.0**) and **G** is the **Gain** in db scaled by 10 (**-18.0db ≤ G ≤ +18.0db**).

$$\textbf{(6-9)} \quad \textbf{N} = \textbf{10}^{\textbf{6}}\textbf{·(B - 3)/27 = (2·10}^{\textbf{6}}\textbf{·B - 6·10}^{\textbf{6}}\textbf{)/54}$$

$$\textbf{(6-10)} \quad \textbf{B = 27·N/10}^{\textbf{6}} + \textbf{3}$$

$$\textbf{(6-11)} \quad \textbf{N = 5·10}^{\textbf{5}}\textbf{·G/180 + 5·10}^{\textbf{5}} = \textbf{5·10}^{\textbf{4}}\textbf{·G/18 + 5·10}^{\textbf{5}}$$

$$\textbf{(6-12)} \quad \textbf{G = 180·(N - 500000)/500000 = (9·N - 4500000)/25000}$$

Note that the right member of **eq (6-9)** is multiplied above and below by 2 to obtain an even divisor of 54. This facilitates the half-adjust rounding which is used by **EqBW_to_ep**. The routine **ep_to_EqBW** is a direct implementation of **eq (6-10)** with half-adjust rounding added. **EqGain_to_ep** and **ep_to_EqGain** are coded the same as the rightmost members of **equations (6-11)** and **(6-12)** respectively with signed half-adjust rounding added. It should be mentioned that these linear functions track Kontakt quite well **except** that Kontakt seems to use some odd form of rounding that doesn't seem to fall in quite the right place.

6.4 Volume Control Functions

Two routines are provided to convert from a simple volume ratio to either **mdb** (for use with the **change_vol()** function) or to an engine parameter (for group or instrument volume control). These routines are named **VR_to_mdb** and **VR_to_ep** respectively. The library also provides a routine to convert a volume ratio in **mdb** to its corresponding engine parameter, **mdb_to_ep**, and the inverse function to convert from the engine parameter to a volume ratio in mdb, **ep_to_mdb**. The legacy routine named **Get_db** (which performs a more restricted form of **VR_to_mdb**) is provided **for backward compatibility only**. However, new scripts should use **VR_to_mdb** because it is more efficient as well as handling the additional positive range from 0 to +12db.

The routine **VR_to_mdb** uses the well-known db formula shown in **equation (6-13)** (middle) and since the library routine **Log10** returns the log scaled by **10,000**, this is equivalent to the rightmost form shown in **eq (6-13)**.

$$(6\text{-}13)\ \ mdb = 20{,}000 \cdot Log(V/V_0) = 2 \cdot Log10(V/V_0)$$

Since the KSP only provides an integer arithmetic package, the actual input volume ratio **VR** is determined by scaling V/V_0 by **10000**. Therefore, in terms of $VR = 10{,}000 \cdot V/V_0$, **eq (6-13)** becomes **eq (6-14)**.

$$(6\text{-}14)\ \ mdb = 2 \cdot Log10(VR \cdot 10^{-4}) = 2 \cdot [Log10(VR) - 40000] = 2 \cdot Log10(VR) - 80{,}000$$

The rightmost member of **eq (6-14)** is how the **VR_to_mdb** routine is coded. The **Get_db** legacy routine performs the same function but clamps the output to **0 mdb** when **VR > 0.0000** for backward compatibility.

The routine **VR_to_ep** uses the reverse-engineered formula given in **equation (6-15)**, where **VR** is again given by $10{,}000 \cdot V/V_0$. Since the cube root is extracted with the Library routine named **Root3**, which returns the root scaled by 10^5, we can recast **eq (6-15)** as **eq (6-16)**. This is the form used by **VR_to_ep** except that half-adjust rounding is also included.

$$(6\text{-}15)\ \ N = 10^6 \cdot (0.25 \cdot V/V0)^{1/3} = 1000 \cdot (25000 \cdot VR)^{1/3}$$

$$(6\text{-}16)\ \ N = Root3(25000 \cdot VR)/100$$

The final pair of Volume routines are **mdb_to_ep** and its inverse **ep_to_mdb**. These routines are based on the reverse-engineered formulae given in **equations (6-17)** and **(6-18)**. In these formulae, **Vol** is the volume in mdb and **N** is the corresponding engine parameter.

$$(6\text{-}17)\ \ N = 10^6 \cdot 10^{(Vol - 12000)/60000} = 10^6 \cdot 10^{-212000/60000} \cdot 10^{(Vol + 200000)/60000}$$
$$= 10^{37/15} \cdot 10^{(Vol + 200000)/60000} = 292.8644567 \cdot 10(Vol + 200000)/60000$$

$$(6\text{-}18)\ \ Vol = 20000 \cdot Log(N/10^6)^3 + 12000 = 60000 \cdot Log(N) - 348000$$

The second form of **eq (6-17)** factors out $10^{-212000}$ so that the variable exponent will always be positive over the entire range of (**-200000 mdb** ≤ **Vol** ≤ **12000 mdb**). The third and forth forms of **eq (6-17)** combines the constant exponents and reduces them to the factor $10^{37/15} \approx$ **292.8644567**. To implement **equation (6-17)**, the Library function **Exp10** is used. But since **Exp10** requires its argument to be scaled up by 10^6, we can rewrite **eq (6-17)** as **(6-19)**. For **eq (6-19)**, the exponential division by 6 is rounded by adding **200003** instead of just **200000**. Since the maximum value of the **Exp10** function will be less than **3500** even at max **Vol = 12000**, it can be multiplied by **292864** without arithmetic overflow. The **mdb_to_ep** routine also rounds the result by half-adjusting the final division by **1000**.

$$\textbf{(6-19)} \ \ N = 292864 \cdot Exp10[100 \cdot (Vol + 200003)/6]/1000$$

To implement the inverse function of **equation (6-18)**, the Library function **XLog10** is used but, since **XLog10** returns the desired log scaled up by 10^6, we will rewrite **eq (6-18)** as **eq (6-20)**. The routine **ep_to_mdb** implements **eq (6-20)** with the addition of half-adjusting (rounding) for the division by 50.

$$\textbf{(6-20)} \ \ Vol = 3 \cdot Xlog10(N)/50 - 348000$$

6.5 Pitch Control Functions

The Pitch format conversion routines are named **Pitch_to_ep** and **ep_to_Pitch**. **Pitch_to_ep** accepts a Pitch value in cents, **-1200** ≤ **P** ≤ **+1200** and returns the equivalent engine parameter. The inverse function accepts the engine parameter and returns its corresponding Pitch value in cents. The reverse-engineered formulae used for these routines are given in **equations (6-21)** and **(6-22)**.

$$\textbf{(6-21)} \ \ N = 500000 \cdot (P/1200)^{1/3} + 500000$$

$$\textbf{(6-22)} \ \ P = 1200 \cdot X^3 \qquad \text{where } X = (N - 500000)/500000$$

To implement **eq (6-21)**, the **Root3** Library routine is used. and it returns the cube root of the radicand, scaled by 10^5. Also, for increased precision of the result, we want to make the radicand as large as possible, which can be accomplished by rearranging the first term of **eq (6-21)** as shown in **eq (6-23)**. This then allows us to rewrite **eq (6-21)** as shown in **eq(6-24)**. Note that **Root3** returns $10^5 \cdot (833333 \cdot P)^{1/3}$ so it must be divided by **200** to get the correct scaling of 500 (ie $10^5/200 = 500$). The actual routine also applies signed half-adjust rounding to the output value given by **eq (6-24)**.

$$\textbf{(6-23)} \ \ 500000 \cdot (P/1200)^{1/3} = 500 \cdot (10^9 \cdot P/1200)^{1/3} \approx 500 \cdot (833333 \cdot P)^{1/3}$$

$$\textbf{(6-24)} \ \ N = Root3(833333 \cdot P)/200 + 500000$$

To implement **eq (6-22)** we need to cube **X**, where $X = (N - 500000)/500000$. But since the value of **X** covers only the small range of **-1.0 to +1.0**, we need to do some clever scaling in order to maintain good precision (without running into arithmetic overflow). We begin by scaling **X** up by 10^4 which we assign to the variable **u**. (ie $u = N/50 - 10000$). We then square **u** to obtain $X^2 \cdot 10^8$ and then divide this by 1000 (with rounding) to obtain $X^2 \cdot 10^5$. Note that we can use a positive half-adjust (rather than a signed round) because the square of **X** must always be positive. We can now multiply by **u** again without overflow (because $|u| < 10^4$) and thus, we obtain $X^3 \cdot 10^9$. However, we want a final result of $1200 \cdot X^3$, so we must **multiply** the 10^9 by a factor of $1.2 \cdot 10^{-6}$ or **6/5000000**. However, $10^9 \cdot X^3$ can be as high as 10^9 and therefore we cannot multiply by **6** without a possible overflow. Therefore, we first divide by **5** then multiply by **6** (which is safe because the value can reach at most $1.2 \cdot 10^9$) and then finally divide by **1000000**. This final division by one million is also rounded but, in this case, we have to use a signed half-adjust value of ±**500000** (depending on the sign of **u**).

6.6 Speed Control Functions

This pair of routines is for conversion of the Source Module's Speed parameter (when operating in the Tone, Time, or Beat Machine modes). **Speed_to_ep** converts a **Speed** value (from **0% to 800%**) to its corresponding engine parameter. The inverse routine **ep_to_Speed** converts the engine parameter to its equivalent **Speed** value. The 'reverse-engineered' formulae used are given in **equations (6-25) and (6-26)**.

$$(6\text{-}25)\quad S = 2^{K \cdot Fn + A} - 50/3 \qquad \text{where } Fn = X^3 + 1$$

$$(6\text{-}26)\quad X = \{[Lg(3 \cdot S + 50)]/K - B\}^{1/3}$$

In the above equations, $X = (N - 500000)/500000$ and **K, A**, and **B** are constants defined as $K = 2.807354922$, $A = 4.058893690$, and $B = 3.010382137$.

To implement **equation 6-25**, **ep_to_speed** begins with cubing **X** (using a similar scaling technique as that discussed for **ep_to_pitch**). The first two lines of code (in **ep_to_Speed**) thus computes $u = 10^6 \cdot X^3$. And, since the **Exp2** routine requires its argument to be scaled by 10^6, the 3rd line then computes the required exponent as shown in **equation (6-27)**.

$$(6\text{-}27)\quad u = 10^6 \cdot [K \cdot (X^3 + 1) + A]$$

For this calculation, **K** is approximated by the ratio $991/353 = 2.807365439$ (which is accurate to nearly 1 part per 3 million) and the half-adjusted value of **u** is then given to **Exp10** to obtain **S'**, the exponential part of **equation (6-25)**. Finally, the last line of **ep_to_Speed** effectively subtracts **50/3** from **S'** (with rounding) to obtain the desired output value for **S**.

To compute the value of **N**, the inverse function **Speed_to_ep** implements **equation 6-26**. The first two lines of code (for **Speed_to_ep**) computes $N' = 10000 \cdot Lg(3 \cdot S + 50)$. Note that **equation (6-26)** can be rewritten as shown in **equation (6-28)**. With this in mind, the 3rd line of code computes the scaled radicand as shown in **equation (6-29)**.

$$(6\text{-}28)\quad X = 1/K^{1/3} \cdot [Lg(3 \cdot S + 50) - B \cdot K]^{1/3}$$

$$(6\text{-}29)\quad u = 10^6 \cdot [Lg(3 \cdot S + 50) - B \cdot K]$$

Since the **Root3** routine returns the desired cube root scaled by 10^5, the next value thus obtained for **N'** is given by **equation (6-30)**.

$$(6\text{-}30)\quad N' = 10^5 \cdot 10^2 \cdot [Lg(3 \cdot S + 50) - B \cdot K]^{1/3}$$

Comparing **eq (6-30)** with **eq (6-28)**, to obtain the value for **X**, we need simply divide **eq (6-30)** by $10^7 \cdot K^{1/3}$ to obtain **eq (6-28)**. And, since $N = 500000 \cdot X + 500000$, we can express **N** in terms of **N'** as shown in **equation (6-31)**.

$$(6\text{-}31)\quad N = 500000/(10^7 \cdot K^{1/3}) \cdot N' + 500000 = K' \cdot N' + 500000$$
$$\text{where } K' = 0.03544357852$$

Since the maximum value of $N' < 1.5 \cdot 10^7$, the most we can multiply it by is 143. The closest approximation for **K'** as a ratio of integers (with a numerator less than 143) is given by $131/3696 = 0.0354437229$, which is accurate to one part per 3 million or so. Thus, the last line of **Speed_to_ep** implements the rightmost member of **eq (6-31)** as shown in **eq (6-32)** with half-adjust rounding.

$$(6\text{-}32)\quad N = 131 \cdot N'/3696 + 500000$$

Appendix A

Library Routine Benchmarks

The execution time for each basic routine was measured using a **Intel Core 2 Duo, 2.53GHz PC with 4 GB of Ram**. Execution time was measured for both **K2** and **K3** (since K3 is noticeably slower at running scripts) and the data is tabulated in **Table A-1**. When execution time is dependent on the input value, the times shown in the table is that for the worst case input data sets. Thus for an average mixture of input values, execution time will be somewhat faster than shown.

Input data values outside the normal range, as well as other special values that the library functions find 'trivial' to compute are of course avoided. For example, the cube-root of 0 or 1 is computed very quickly without invoking the normal body of the algorithm. Using such input values for testing would produce deceptively fast results and are therefore excluded from the benchmark tests. Overall, the tablulated data tends to be more pessimistic than optimistic.

But, keep in mind that this data was taken with Kontakt running standalone and no other applications running (other than the usual background services). If you have a faster or slower computer or if you are running many other applications, you may experience better or worse performance. However, the relative performance of the routines should be fairly consistent. For example, **Log10** will generally run about six times faster than **XLog10** and any of these routines will run faster with **K2** than with **K3**.

Library Routine	K2	K3
ep_to_mdb	9.0 µs	10.8 µs
Exp2	5.2 µs	6.7 µs
Exp10	5.5 µs	6.9 µs
exp	5.6 µs	7.0 µs
Get_db	1.9 µs	2.5 µs
Log2	1.2 µs	1.5 µs
Log10	1.4 µs	1.8 µs
Ln	1.4 µs	1.8 µs
mdb_to ep	6.3 µs	7.1 µs
Pitch_to_ep	1.6 µs	2.0 µs
Root3	1.4 µs	1.9 µs
SinCos	8.2 µs	9.5 µs
Tangent	8.3 µs	9.7 µs
VR_to_ep	1.9 µs	2.3 µs
VR_to_mdb	1.3 µs	1.6 µs
XLog2	8.4 µs	10.0 µs
XLog10	8.6 µs	10.4 µs
XLn	8.5 µs	10.4 µs

<u>Table A-1</u>
Execution Times