KSP Math Library

User's Guide

V610

Kontakt 4/5

Robert D. Villwock

1-22-14

Table Of Contents

1.0 KSP Math Library Overview 4

2.0 Using V610 5
- 2.1 SetMathMode 5
- 2.2 do_post_init 6
- 2.3 About Post-Initialization 6
- 2.4 Using wait_for_post 6

3.0 The Library Routines 7
- 3.1 System Directives 7
  - AppInit 7
  - do_post_init, on_pgs_do_post 7
  - SetMathMode 7
  - wait_for_post 7
- 3.2 Advanced Data Types 7
  - 2dArray, 3dArray 7
  - PackedArray 7
- 3.3 Basic Functions 8
  - Boolean 8
  - Cabs 8
  - Cos 8
  - Expe, Exp2, Exp10 8,9
  - Loge, Log2, Log10 9
  - MulDiv64 9
  - Power 9
  - Rand, ResetRand 9
  - Root2, Root3 9
  - RoundDiv 10
  - Sign, Sign3 10
  - Sin, SinCos 10
  - Tan 10
  - XOR 10
  - XCos, XSin 10
  - XSinCos 10
  - XTan 10
  - Library Constants 11
- 3.4 Special Functions 12
  - ATFade 12
  - EP_XFade 13
  - S1_XFade 13
  - S2_XFade 13
  - S3_XFade 13

3.5 Format Converters 14
CVHex 14
DFmtVal 14
*** Bi-Directional EP Converters ***
epARFreq 14
epAtkTime 14
epDAFTFreq 14
epDRTime 14
epEqBW 14
epEqFreq 14
epEqGain 14
epHBFreq 15
epHP2Freq 15
epLFOFreq 15
epLPFreq 15
epProFreq 15
epSGEqBW 15
epSGEqFreq 15
epSGEqGain 15
epSpeed 15
epSVFreq 15
epSVMFreq 16
epTune 16
epV3x2Freq 16
epVolume 16
epVR 16
****
GetTrueEP 16
ModInt_to_ep 16
VR_to_mdb 16

4.0 Fast Math 17
4.1 Why Use Standard Mode 17

5.0 epVolume Mini Tutorial 17

Figures and Tables

Table 2-1 5
Figure 3-1 12
Figure 3-2 12

Appendix

Using Import As 18

1.0 KSP Math Library Overview

Most of Kontakt’s control functions are non-linear. For example, volume control has a logarithmic relationship. Similarly, time and frequency control is based on log/exp relationships. In addition, equal-power crossfading of two audio signals requires the use of sine/cosine contouring. Therefore, when writing scripts, there is often a need for some of these standard math functions (especially in the area of engine parameter control) but the KSP only provides basic arithmetic functions. So, the **KSP Math Library** bridges this gap with a large collection of the most needed math functions required by Kontakt scripters.

In addition to the standard functions such as log, exponential, trigonometric, powers, and roots, the library also contains an ever-expanding collection of **Engine Parameter Converters** using basic functions and other sophisticated techniques to model the control curves found in Kontakt. Even though most of these converters use functions like logs or exponentials, it is not necessary for you to understand such mathematical details to benefit from these routines. **The engine converters are very easy to interface with — even if your math skills are virtually non-existent.** You need not trouble yourself with **how** these routines work. You only need to know what kind of values to give them and what kind of values you can expect them to return. In fact, many scripters just use the math library for the engine converters. However, all the standard functions are also available for those that have sufficient math skills to benefit from them.

Even though the library is quite large, importing it will not 'bloat your code' because nothing will be added to your code except what is needed to support the functions you actually use. If you don't reference any of the Library's functions, no code or data declarations will be added to your compiled code (provided that you configure the KScript Editor properly as outlined in the Box at the top of page 5).

The Math Library is very easy to import and use in your application scripts. How you do this is discussed in section 2 of this guide and an overview of all the library functions is presented in section 3. If you need to invoke library functions (or you would like to be able to call your own user functions) during initialization, be sure to read section 2.3 on how to use the **post-initialization feature** provided with the library.

In more demanding applications, heavy usage of some of the library routines like log, exp, or trig functions can be a bit taxing on your CPU in some cases. For these situations, you can take advantage of the library’s **Fast Math** mode. With V610, you can easily switch back and forth between **Standard** and **Fast Math** modes at any time with no more fuss than changing a compiler switch. Unlike prior versions of the library, there is no need to set up or connect to any NKA files. V610 automatically builds its own lookup tables as needed without any effort on your part. You’ll find a brief discussion of **Fast Math** in section 4.

Finally, you’ll find **a mini-tutorial** on using engine parameter converters in section 5.0. I’m often asked to provide coding examples of how to use some of the ep converters in a practical application. So, the library now includes a nice demo instrument and tutorial script that illustrates some of the ways you can have your script control volume using the library’s **epVolume** function. By studying the heavily-commented source code, you should easily be able to apply similar techniques to your particular application.

2.0 Using V610

IMPORTANT PLEASE NOTE

The Math Library is an 'import' module and requires that you use Nils Liberg's KScript Editor (KSE) V1.5.2 (or its equivalent Sublime Text 3 plugin) to compile your application. Your host script should include the directive **import "KSPMathV610.ksp"** ahead of your Initialization Callback (ICB). **You must use the new compiler** (not the optional old compiler) **and you must enable 'Optimize compiled code'** and **'Compact output**.' Failure to set these options will result in much more compiled code than is needed. You can also optionally use the Compact variables option is desired.

Besides importing the library, there are only two things you **must do** to use the library in your application script. You must include the **SetMathMode** directive in your script's ICB and you must invoke **do_post_init** from your script's pgs callback. See sections 2.1 and 2.2 for more information about these two **mandatory** steps.

Once you have included **SetMathMode** and **do_post_init** in your script, you can freely use any library routine. Generally, to use a library routine, you should consult the header comments in the source code for the specific routine of interest **before** attempting to use it. To work within the integer-only limitation of the KSP, most library functions scale input and output parameters in order to provide reasonable precision and dynamic range. The header comments will tell you what parameter scaling is used as well as what the function is supposed to do for you.

All library routines treat **input** parameters as **read-only** so you can pass variables or constants (symbolic or literal) if desired. The library uses the KSE return-value feature for all single-output-value functions but, for multi-output functons (such as SinCos), output values are returned as a part of the argument list. See KSE documentation for more information on function types supported.

2.1 SetMathMode(option_list)

This directive **must be** invoked near the top of your **ICB** and before any other library references can be made. This directive requires an **option_list** parameter which is constructed by summing the desired **Fast Math** options. The options currently available and what they do are summarized in **Table 2-1** and further discussed in section 4.

The options (in any combination) are simply strung together with connecting plus signs. However, **do not use more than one** of each option type in the list. For example, **SetMathMode**(**FL+FE**) will compile all log and exponential routines in their fast mode. If you want to run with the whole library in **Standard Mode**, just use **zero** for the option_list like this: **SetMathMode(0)**.

Option	Compilation Action	Reference
FL	Use Fast-Math for logarithms and derivative routines	See section 4
FE	Use Fast Math for exponentials and derivative routines	See section 4
FT	Use Fast Math for trig functions and derivative routines	See section 4

Table 2-1

2.2 do_post_init

This directive **must be** invoked from the head of your pgs callback. However, if your script doesn't use a pgs callback, you can instead invoke **on_pgs_do_post** right after your ICB. This will both create a pgs callback handler and embed a reference to **do_post_init** within it. **NOTE:** You **must** include **do_post_init** in your pgs callback handler **or use on_pgs_do_post,** regardless of whether or not you write an **AppInit** routine (section 2.3)**.** The library itself requires **do_post_init** to perform its own initialization properly.

2.3 About Post-Initialization

Callable user functions can avoid repeated code in your scripts but oftentimes there is a need to invoke these same actions during initialization. You may be tempted to just repeat this code in-line because Kontakt doesn't allow user function calls from the ICB. However, the Math Library provides an easy way to avoid such duplicated code by extending the ICB into a post-initialization phase.

The post-initialization phase is run within the pgs callback and therefore does not have the same restrictions as the ICB. To avail yourself of this feature, you merely write a KSE function named **AppInit** using the KSE **override** directive. **AppInit** will be executed immediately after the ICB finishes so it will be the same as if you had placed **AppInit** at the end of your ICB. However, you can include code in **AppInit** that you couldn't use if it was run in the ICB itself. For example, you can include things like user function calls or even **wait** statements if desired (see section 2.4).

Since the Math Library makes extensive use of callable user functions, most library routines cannot be invoked directly from the ICB either. However, if you need to use library functions during initialization, all you have to do is invoke them from your **AppInit** routine.

To illustrate this with a trivial (nonsense) example, suppose that during initialization, you want to set a variable named **RootKnob** to the square root of the persisted value of one of your panel knobs named **MyKnob**. You could code this as follows:

```
function AppInit override
   read_persistent_var(MyKnob)    { Interestingly enough, this is not needed, see NOTE below }
   RootKnob := Root2(MyKnob)
end function
```

As long as you can arrange it so that the actions you want executed during initialization can be placed at the *end* of the ICB, you can simply include these actions in **AppInit** instead. That way, you are free to use KSP user function calls and/or library references as you wish and yet these actions will be performed with initialization.

NOTE: The read_persistent_var (which would be necessary if **AppInit** were actually executed in the ICB), is **not** needed when **AppInit** is executed in the post-initialization phase. All persistence vars are loaded automatically after the ICB exits so you no longer need to *explicitly* load them. This can often be exploited to avoid a number of **read_persistent_var** statements in the ICB.

2.4 Using wait_for_post

AppInit will be invoked immediately after the ICB exits. However, if you use any **wait** statements in **AppInit**, it is possible that **AppInit** will not completely finish before some other callbacks are triggered. If those callbacks rely on your initialization being finished, you may want to use the **wait_for_post** directive at the head of those callbacks. If **AppInit** hasn't finished yet, **wait_for_post** will pause the callback until it does. Once **AppInit** finishes, **wait_for_post** does nothing.

CAUTION: Do not invoke **wait_for_post** within your **AppInit** routine itself.

3.0 The Library Routines

The source code of the KSP Math Library is divided into 5 main sections as follows:

System Directives
Advanced Data Types
Basic Functions
Special Functions
Format Converters

These five sections are followed by a number of internal support sections. Within each section, the routines are generally arranged in **alphabetical order** for convenience in locating them. Your application will ordinarily only reference the functions in the first five sections of the source code. You won't need to concern yourself with any of the support routines unless you intend to modify the library in some way.

3.1 System Directives

AppInit

This is discussed in section 2.3

do_post_init, on_pgs_do_post,

These are discussed in section 2.2

SetMathMode

This is discussed in section 2.1

wait_for_post

This is discussed in section 2.4

3.2 Advanced Data Types

2dArray(name,d1,d2) and **3dArray(name,d1,d2,d3)**

These macros can be used to create 2 and 3-dimensional arrays where d1, d2, d3 are compile-time constants. For example, you can create arrays named **My2D** and **My3D** as follows:

```
2dArray(My2D,6,10)         { This creates a 6 x 10 word array named My2D }
3dArray(My3D,40,50,10)     { This creates a 40 x 50 x 10 word array named My3D }
```

Once such arrays are declared they can be referenced with the standard, multi-dimensional array syntax. For example:

```
My2D[3,7] := My3D[x,y,5]
```

The indices can be either constants or variables but please note that no runtime range checking is performed so your application must ensure that the index values you use (when you reference the array) are within the bounds you specified when you declared the array. For example, if you declare **3dArray**[name,4,5,6] you may only use indices from 0..3, 0..4, and 0..5 respectively when you reference this array. Also, be aware that internally these arrays are implemented as one-dimensional KSP arrays and therefore, the product of the dimension constants used to declare the array must not exceed 32,768 (the current size limit for KSP arrays).

PackedArray(name,d1,d2,ft)

This macro can be used to create a pseudo, 2-dimensional array where d1 is the number of array words and d2 is the number of bit fields within each 32-bit word. You may sub-divide the 32 bits into from 2 to 32 fields of various widths (from 1 to 31 bits each) but the total number of field bits cannot exceed 32. The field format itself is specified with a template array, **ft**, that you declare separately.

To illustrate the usage of **PackedArray**, let's create two packed arrays named **MyPars** and **MyFlags**. **MyPars** will be an array of 200 words, each word containing 6 data fields with widths 2, 3, 4, 5, 6, and 7 bits respectively. **MyFlags** will be an array of 100 words — each containing 32 Boolean flags.

To accomplish this, we first declare the field template arrays for **MyPars** and **MyFlags**.

```
declare MyParsTemplate[6] := (2,3,4,5,6,7)
declare MyFlagsTemplate[32] := (1)          { equivalent to 1,1,1,1, ... 1  (32 times) }
```

Now we can define the packed arrays themselves with:

```
PackedArray(MyPars,200,6,MyParsTemplate)
PackedArray(MyFlags,100,32,MyFlagsTemplate)
```

We can then reference any field of **MyPars** with MyPars[ax,fx] where ax is the array word index (a value from 0 to 199) and fx is the data field index (a value from 0 to 5). For example:

```
MyPars[6,3] := MyPars[12,0] { copies the 2-bit field of word 12 to the 5-bit field of word 6 }
```

We can reference **MyFlags** with MyFlags[ax,fx] where ax is the array word index (a value from 0 to 99) and fx is the flag index (a value from 0 to 31). For example:

```
if MyFlags[6,23] = 1
   message('On')       { Flag 23 of word 6 is set }
else
   message('Off)       { Flag 23 of word 6 is zero }
end if
```

3.3 Basic Functions

This section is where to look for things like log or trig functions. Here you will find extended arithmetic and transcendental functions along with some miscellaneous functions such as conditional predicates and a resettable random number generator. Also, at the end of this section, you will find a list of library constants that are available for your use.

Boolean(X)

This is a conditional predicate. **Boolean(X)** returns 0 when X = 0 and 1 otherwise. **Since this is a one-line function, the KSP will allow it to be referenced in-line** (anywhere in your code).

Cabs(X)

Clamped absolute value. This function performs the same operation as **abs** but it also handles X = MinInt more intelligently. Since there is no positive counterpart of X = 0x80000000, if you use abs(X), the KSP will simply return 0x80000000 which of course isn't a positive value (as one would expect when taking the absolute value). Whereas **Cabs(X)** will *clamp* the return value to 0x7FFFFFFF which is the largest possible positive value. For all other input X values, **Cabs** and **abs** return the same value. **Since this is a one-line function, the KSP will allow it to be referenced in-line** (anywhere in your code).

Cos(X)

This function returns the cosine of angle X, where X is a first-quadrant angle in deci-grads (1000 dg = 90°). **Cos**, like all trig functions, returns the value scaled by 10000. See also **XCos, SinCos, and XSinCos.**

Expe(X)

The natural (base e) anti-log function. When X = 10^6*Z, this function returns the value of $\mathbf{e}^{Z}$

Exp2(X)

The binary anti-log function. When X = 10^6*Z, this function returns the value of $\mathbf{2^Z}$

Exp10(X)

The common anti-log function. When X = 10^6*Z, this function returns the value of $\mathbf{10^Z}$

Loge(X)

This function returns the natural logarithm (base e) of X scaled by 10^6.

Log2(X)

This function returns the binary logarithm (base 2) of X scaled by 10^6.

Log10(X)

This function returns the common logarithm (base 10) of X scaled by 10^6.

MulDiv64(X,Y,Z)

This routine computes X*Y / Z as a 32-bit result but it maintains the product of X*Y internally as a 64-bit value. Thus, as long as the final value of X*Y / Z can be contained in a 32-bit signed integer, no arithmetic overflow will occur.

This can be useful for those stubborn situations in which one needs to multiply by a large number before dividing it back down to a smaller value. **MulDiv64** can thus avoid the necessity of trying to factor Y into two values such that $Y = Y_1*Y_2$ and then doing something like $X*Y_1$ / $Z*Y_2$ to avoid arithmetic overflow.

Power(X,n,d,dd)

This general power function allows you to compute $X^{n/d}*10^{dd}$ where X, n, d and dd are positive integers. The routine uses Log2/Exp2 internally so the result is approximate to about 6 significant decimal digits.

Here are some examples of how this routine can be used:

Power(X,1,2,2)	{ computes the square root of X scaled by 100 }
Power(X,1,3,1)	{ computes the cube root of X scaled by 10 }
Power(X,3,1,0)	{ computes X^3 }
Power(X,7,10,3)	{ computes $X^{0.7}$ scaled by 1000 }

As can be seen, this routine is quite flexible but since it uses log/exp internally, you may experience a bit of CPU drag unless you use the fast modes for the log/exp routines by including **FL+FE** in your **SetMathMode** option_list.

Rand(a,b)   RandomSeed   ResetRand

Rand(a,b) returns a random value between a and b inclusive. This performs essentially the same function as the **KSP random** function but, **Rand** is resettable by invoking **ResetRand** whenever you want to start a new, ‘repeatable random’ series. You can also opt to ‘randomly’ initialize the generator by invoking **RandomSeed**.

Root2(X)   Root3(X)

Root2(X) returns the square root of X scaled by 10^3. **Root3(X)** returns the cube root of X scaled by 10^5. Please see the header comments of these routines for more detail.

RoundDiv(X,Y)

The divide operation in the KSP always results in a truncated quotient. However, it is often desirable to round the quotient to the 'nearest' integer. This is fairly easy to do when the quotient is positive by simply half-adjusting. But, when signed numbers are involved, the rounding process becomes trickier. **Round-Div(X,Y)** will always return the quotient of X / Y rounded to the closest integer for any signed integers X and Y. For example if X is -66 and Y is 10, X / Y returns the value -6 whereas RoundDiv(X,Y) returns -7. And, **since this is a one-line function, the KSE will allow it to be referenced in-line** (anywhere in your code).

NOTE: There are differences in the way the KSE and KSP do arithmetic and that can result in different results when both X and Y are constants (compared with variables containing the same values). Nils has informed me that this KSE issue has already been corrected and it's only a matter of time before an update will be available. However, in the meantime, as long as X or Y (or both) are variables, the problem doesn't exist and there is really no good reason to use **RoundDiv** when both X and Y are constants.

Sign(X), Sign3(X)

Thes are conditional predicates. **Sign(X)** returns +1 when X ≥ 0 and -1 when X < 0. **Sign3(X)** returns +1 when X > 0, -1 when X < 0, and 0 when X = 0. **Since these are one-line functions, the KSE will allow them to be referenced in-line** (anywhere in your code).

Sin(X)

This function returns the sine of angle X, where X is a first-quadrant angle in deci-grads (1000 dg = 90°). **Sin**, like all trig functions, returns the value scaled by 10000. See also **XSin, SinCos, and XSinCos.**

SinCos(X,sin,cos)

SinCos outputs both the sine and cosine of a first-quadrant angle X in deci-grads (1000 dg = 90°). **SinCos** is **not** a return-value function. The dual outputs are passed to the variables you stipulate as part of the parameter list. **SinCos**, like all trig functions, returns values scaled by 10000. See also **XSinCos**.

Tan(X)

This function returns the tangent of a first-quadrant angle X in deci-grads (1000 dg = 90°). **Tan**, like all trig functions, returns the value scaled by 10000. See also **XTan**.

XCos(X)

This function returns the cosine of angle X, where X can be any extended angle in deci-grads (1000 dg = 90°). **XCos**, like all trig functions, returns the value scaled by 10000. See also **Cos, SinCos, and XSinCos.**

XOR(X,Y) Returns the bit by bit **Exclusive OR** of X and Y. This one-line function can be used inline.

XSin(X)

This function returns the sine of angle X, where X can be any extended angle in deci-grads (1000 dg = 90°). **XSin**, like all trig functions, returns the value scaled by 10000. See also **Sin, SinCos, and XSinCos.**

XSinCos(X,sin,cos)

XSinCos outputs both the sine and cosine of any extended angle X in deci-grads (1000 dg = 90°). **XSinCos** is **not** a return-value function. The dual outputs are passed to the variables you stipulate as part of the parameter list. **XSinCos**, like all trig functions, returns the value scaled by 10000. See also **SinCos**.

XTan(X)

This function returns the tangent of angle X, where X can be any extended angle in deci-grads (1000 dg = 90°). **XTan**, like all trig functions, returns the value scaled by 10000. See also **Tan.**

Library Constants

The following constants are made available by the library for use in your scripts. The first four are generally useful values while the remaining constants are for mode control of various ep converters and such.

Ang90

This constant (currently = 1000 dg) is useful for angle conversion from arbitrary right-angle units. For example, if you want to use a MIDI CC to cover the range from 0..90 degrees, you can use:

```
angle := CC*Ang90/127.
```

MaxInt

This constant is the maximum (most-positive) integer value which is 2,147,483,647 or 0x7FFFFFFF.

MinInt

This constant is the minimum (most-negative) integer value which is -2,147,483,648 or 0x80000000.
NOTE: For V600M, **MinInt** is actually a variable so be careful not to alter it with your code.

Muted

This constant is currently -180,000 mdb in value

****

E2V, V2E	Bi-directional ep converter mode control
IMOD, TARG	Mode control for ModInt_to_ep function
LF,LMF,HMF,HF	Solid-G EQ-Band selection
MATH_OVERFLOW	For use in the **TCM** Debug Mode

3.4 Special Functions

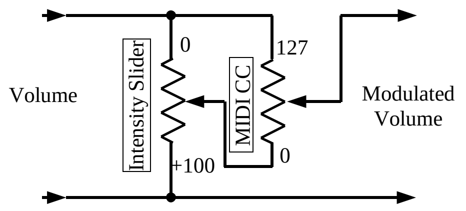
This section is where to look for audio-taper volume control or various morphing crossfade functions.

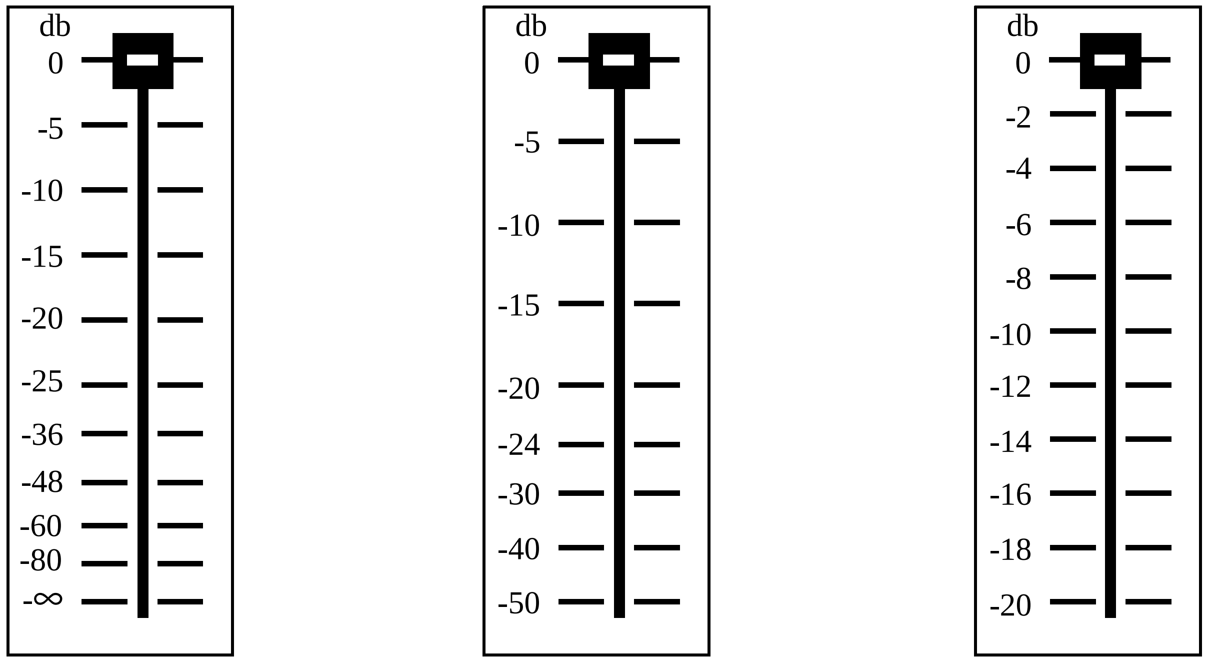
ATFade(cv,rng)

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 two parameters like this: **ATFade(cv,rng)**. **cv** is an input value (0..127) from a MIDI CC and **rng** is an input value from **0..100%** from a user adjustable Range control. **ATFade** returns an audio-contoured attenuation in mdb which can be used as input for the **change_vol** function or the **epVolume** function.

The electronic equivalent of Kontakt's MIDI CC control is shown in **Figure 3-1**. The **ATFade** function operates in a similar way except that with **ATFade**, the taper of the MIDI CC is more like that of a traditional audio fader whereas in Kontakt, the taper of the potentiometers is simply linear. So, when you assign a CC to modulate the amplifier in Kontakt, 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 Kontakt'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 down to the full size (see **Figure 3-2**). With such tapers, you can easily control volume more expressively.

Figure 3-1
Electronic Equivalent For Kontakt's Positive Modulation





Range = 100% **Range = 80%** **Range = 50%**

Figure 3-2
Audio Contours
at various 'Range' settings

EP_XFade S1_XFade S2_XFade S3_XFade

These are four morphing crossfade functions. **EP_XFade(xv,mv,VR1,VR2)** produces the familiar equal power crossfade outputs for VR1 and VR2 as the crossfade variable **xv** swings from 0 to 1000. While this alone can be done with the existing SinCos function, **EP_XFade** adds a new morphing variable, **mv**. When mv = 0, the outputs follow the traditional sine/cosine curve however, when mv = 1000, the outputs produce a purely linear crossfade. And, as mv swings from 0 to 1000, the curves will smoothly morph from sine/cosine to linear.

In addition to **EP_XFade**, three variants of S-shaped crossfade curves are provided with **S1_XFade**, **S2_XFade**, and **S3_XFade**. S1 is based on a 180-degree traversal of a cosine curve. S2 is based on an exponential/logarithmic curvature and S3 is based on a fully symmetrical quadratic curve. All three of these S-Curve functions also provide full morphing control as for EP_XFade.

The morphing feature may prove useful for stubborn musical situations where you just need something in-between a theoretically-good shape and a simple linear shape. The morphing variable will allow you to experiment to your heart's content until you hopefully find something that sounds just right ☺.

The S1 and S2 functions were suggested by Tuwa Sni and I added S3 to the hopper. I have no personal experience with any of these S-shapes but Tuwa seems to feel that S1 and S2 at least can provide musically pleasing crossfades.

So that you can visually see what these 4 crossfade curves look like in practice, the Math Library package includes a Demo instrument named **XFadeDemo.nki**. You can load this instrument into K4/K5 and take a look at the curves and the effect of the morphing control. To use the demo, click the drop-down menu to select one of the curves and then spin the Morph knob from -100 to +100% to watch its effect on the shape (this knob swings **mv** from -1000 to +1000). The graphical depiction of the curves is of course a little crude since the KSP Table component allows for only 128 bars to display both the up and down curves (only 64 bars each). However, the general shape can easily be seen. The real curve produced by these functions is of course much smoother. You can accurately read the pair of **VR** values for any given value of the crossfade variable by using the XFade knob. As you spin the XFade knob from 0 to 100% (this knob swings **xv** from 0 to 1000), the corresponding **VR** values are displayed in their decimal format where VR_{max} = 1.0000. Of course the actual outputs of the XFade functions are 10000 times these displayed values (to keep it in the integer domain). This scaled format is also the same as that required as input to the library functions **VR_to_mdb** and **epVR** so you can use the VR outputs with **VR_to_mdb** to crossfade notes via **change_vol** or you can use them with **epVR** to perform group volume crossfades via engine parameters.

Using A Negative Morphing Value

These routines were originally designed for an **mv** value range from 0 to +1000. However, as was pointed out to me in a PM from **Gabriel Rey-Goodlatte**, negative values of **mv** might be useful for certain special situations. The effect of using a negative **mv** value is to exaggerate the curvature. For example, Gabriel was using the **EP_XFade** function and wanted to *lift* the center crossover point above the normal 70.7%.

To let you see what the curves look like with negative **mv** values, the **Morph** knob in the demo instrument now covers the expanded range from -100% to +100% (an **mv** range from -1000 to +1000). However, if you use negative values for **mv**, keep in mind that as **xv** swings from 0 to 1000, there will be regions where the **VR** outputs will exceed the normal 0 to 10000 range (as you will be able to see on the numeric VR display for the Demo Script). Depending on how your application utilizes these VR outputs, you may have to *clamp* them before sending them on to some routines that are only designed to handle the normal 0 to 10000 range.

3.5 Format Converters

This is where to look if you need to convert from one unit or scale to another. For example, all the engine parameter converters are here along with several non-ep format converters.

CVHex(val)

CVHex returns an 8-digit, hexadecimal text string representation of the input integer value with the suffix ‘h’. For example, if **val** = 1234567890, the string returned will be 499602D2h. Note that no leading zero suppression is performed so that if **val** = 123456, the string returned will be 0001E240h.

DFmtVal(val,dd)

This function returns a text string representation of **val** formatted with **dd** decimal digits. For example, if the input value is **12345** and **dd = 2**, the output string will be **123.45**. Whereas if **dd = 6**, the output string will be **0.012345**.

***** Bi-Directional EP Converters *****

All the bi-directional converters allow you to convert from the data **value** to the corresponding **ep** and vice versa. The conversion direction is determined by the **mode** constant you supply in the parameter list. There are two mode constants named **V2E** and **E2V** which control the conversion direction as value_to_ep and ep_to_value respectively. For example, **epVolume(V2E,vol)** will return the equivalent **ep** for the input **vol** in mdb. Whereas **epVolume(E2V,ep)** will return the equivalent volume in mdb for the input **ep** value. Also please note that all bi-directional converter names begin with **ep**, such as **epVolume**, **epAtkTime**, etc.

epARFreq(V2E,F)   epARFreq(E2V,ep)

This routine converts cutoff frequency **F** for the AR filter in the range from 8.2 to 35500.0 Hz (scaled by 10) to/from the equivalent **ep**. This routine is usable with AR-LP2, AR-LP4, AR-LP2/4, AR-HP2, AR-HP4, AR-HP2/4, AR-B2, AR-B4, and AR-B2/4.

epAtkTime(V2E,T)   epAtkTime(E2V,ep)

This routine converts envelope attack time **T** in the range from 0.00 to 15000.00 msec (scaled by 100) to/from the equivalent **ep**.

epDAFTFreq(V2E,F)   epDAFTFreq(E2V,ep)

This routine converts cutoff frequency **F** for the DAFT filter in the range from 26.0 to 35500.0 Hz (scaled by 10) to/from the equivalent **ep**.

epDRTime(V2E,T)   epDRTime(E2V,ep)

This routine converts envelope **decay** or **release** time **T** in the range from 0.00 to 25000.00 msec (scaled by 100) to/from the equivalent **ep**.

epEqBW(V2E,B)   epEqBW(E2V,ep)

This routine converts the Bandwidth **B** (for the standard 1, 2, or 3 band EQ) in the range from 0.30 to 3.00 octaves (scaled by 100) to/from the equivalent **ep**.

epEqFreq(V2E,F)   epEqFreq(E2V,ep)

This routine converts Frequency **F** (for the standard 1, 2, or 3 band EQ) in the range from 20.0 to 20000.0 Hz (scaled by 10) to/from the equivalent **ep**.

epEqGain(V2E,G)   epEqGain(E2V,ep)

This routine converts Gain **G** (for the standard 1, 2, or 3 band EQ) in the range from -18.0 to +18.0 db (scaled by 10) to/from the equivalent ep.

epHBFreq(V2E,F) epHBFreq(E2V,ep)

This routine converts high-pass/band-pass cutoff frequency **F** in the range from 36.1 to 18100.0 Hz (scaled by 10) to/from the equivalent **ep**. This routine is usable with the **Legacy** filters HP1, HP4, BP2, BP4, and BR4. For **HP2** use **epHP2Freq**.

epHP2Freq(V2E,F) epHP2Freq(E2V,ep)

This routine converts high-pass cutoff frequency **F** in the range from 37.3 to 18700.0 Hz (scaled by 10) to/from the equivalent **ep**. This routine is for the **Legacy** HP2 filter.

epLFOFreq(V2E,F) epLFOFreq(E2V,ep)

This routine converts LFO frequency **F** in the range from 0.01 to 213.10 Hz (scaled by 100) to/from the equivalent **ep**. See also **GetTrueEP**.

epLPFreq(V2E,F) epLPFreq(E2V,ep)

This routine converts low-pass cutoff frequency **F** in the range from 43.6 to 21800.0 Hz (scaled by 10) to/from the equivalent **ep**. This routine is for the **Legacy** LP1, LP2, LP4, and LP6 filters.

epProFreq(V2E,F) epProFreq(E2V,ep)

This routine converts Pro filter cutoff frequency **F** in the range from 26.0 to 8400.0 Hz (scaled by 10) to/from the equivalent **ep**. This routine is for the **Pro53** and **Legacy** Ladder filters.

epSGEqBW(V2E,B) epSGEqBW(E2V,ep)

This routine converts LMF/HMF bandwidth **B** (Q) for the Solid-G EQ in the range from 0.70 to 2.50 (scaled by 100) to/from the equivalent **ep**.

epSGEqFreq(V2E,band,F) epSGEqFreq(E2V,band,ep)

This routine converts the LF/LMF/HMF/HF frequency **F** for the Solid-G EQ to/from the equivalent **ep**. The frequency **band** is specified with the library constants **LF**, **LMF**, **HMF**, or **HF**. The range of frequencies for each band is as follows.

LF band:	40.0 to 600.0 Hz (scaled by 10)
LMF band:	200.0 to 2500.0 Hz (scaled by 10)
HMF band:	600.0 to 7000.0 Hz (scaled by 10)
HF band:	1500.0 to 22000.0 Hz (scaled by 10)

epSGEqGain(V2E,G) epSGEqGain(E2V,G)

This routine converts Solid-G EQ Gain **G** (for LF/LMF/HMF/HF) in the range from -20.0 to +20.0 db (scaled by 10) to/from the equivalent **ep**.

epSpeed(V2E,S) epSpeed(E2V,ep)

This routine converts time/beat machine speed **S** in the range from 0.0 to 800.0% (scaled by 10) to/from the equivalent **ep**.

epSVFreq(V2E,F) epSVFreq(E2V,ep)

This routine converts **SV** and **Ladder** cutoff frequency **F** in the range from 26.0 to 19900.0 Hz (scaled by 10) to/from the equivalent **ep**. This routine is for the non-multi **SV** filters, LP1, LP2, LP4, HP1, HP2, HP4, BP2, BP4, and Notch4. This routine can also be used for the **Ladder** filters LP1, LP2, LP3, LP4, HP1, HP2, HP3, HP4, BP2, BP4, Peak, and Notch.

epSVMFreq(V2E,F)   epSVMFreq(E2V,ep)

This routine converts the **Multi SV** filter cufoff frequency **F** in the range from 2.6 to 84400.0 Hz (scaled by 10) to/from the equivalent **ep**. This routine can be used with the Par L/H, Par B/B, and Ser L/H filters.

epTune(V2E,T)   epTune(E2V,ep)

This routine converts instrument/source tuning **T** in the range from -3600 to +3600 cents to/from the equivalent **ep**.

epV3x2Freq(V2E,F)   epV3x2Freq(E2V,ep)

This routine converts the **Versatile 3x2** filter cutoff frequency **F** in the range from 26.0 to 15800.0 Hz (scaled by 10) to/from the equivalent **ep**.

epVolume(V2E,V)   epVolume(E2V,ep)

This routine converts instrument/group volume **V** in the range from Muted to +12000 mdb to/from the equivalent **ep**.

epVR(V2E,VR)   epVR(E2V,ep)

This routine converts scaled volume ratio **VR** ($10000*V/V_0$) in the range from 0.0000 to 4.0000 to/from the equivalent **ep** for group/instrument volume control.

****

GetTrueEP(get_ep,set_ep)

This routine returns the **ep** value that was last sent to it. It has been discovered that certain engine parameters do not always return the same value as written to them. In particular, the get_ep for LFO frequency often differs from the value sent with set_ep (but there may be others). This routine works by first reading the incorrect get_ep value and then finding the set_ep value required to produce that incorrect get_ep value. This servo-loop process returns the correct ep value within ±4. See this routine's header comments for more information on how to use it.

ModInt_to_ep(type,P)

This function will convert P = % modulation intensity (scaled by 10000) to the equivalent engine parameter. Thus this function can essentially linearize control of the modulation intensity slider. This routine works with either ENGINE_PARAMETER_INTMOD_INTENSITY or with MOD_TARGET_INTENSITY by specifying the **type** parameter with one of the library constants **IMOD** or **TARG** respectively. For use with **type = IMOD**, the control parameter **P** can be in the range of $-1000000 \leq$ **P** $\leq +1000000$. When **P** is negative, the modulator's **Invert** button will be activated. With **type = TARG**, only the absolute value of **P** will be used for control. Thus, when using ENGINE_PARAMETER_MOD_TARGET_INTENSITY, you must issue a separate engine parameter command to activate the **Invert** button.

VR_to_mdb(VR)

This routine returns the **mdb** equivalent of the scaled input volume ratio VR **=** $\mathbf{10^4*V/V_0}$. For example, if VR = 4.0000, 2.0000, 1.0000, 0.5000 or 0.2500 the value returned will be 12000, 6000, 0, -6000, and -12000 mdb respectively. This routine performs the same function as the deprecated routine named **Get_db** but with a wider **VR** range from 0.0000 to 4.0000 (versus 0.0000 to 1.0000 for **Get_db**).

4.0 Fast Math

Most library functions execute very fast and therefore put very little demand on your CPU. However, the core routines for log, exp, and trig functions are in a different class. Even though they are implemented using very efficient and compact algorithms, these core functions require more horsepower to accomplish their mission. And, since many of the engine converters utilize one or more of these core routines, heavy use of these converters can become a bit taxing for your CPU. But happily, the solution to this problem is just a compiler switch away.

The library includes special versions of the core log, exp, and trig functions that execute about 10 times faster than the standard versions. By using large lookup tables and sophisticated argument reduction techniques, these **Fast Math** core routines have far less CPU demand than their **Standard Mode** counter-parts. And of course by speeding up these core routines, all the library functions that depend on them run faster also.

For each of the three core routines, there is a compiler option that you can include in the **SetMathMode** directive (see section 2.1). If you include **FL**, all library routines that use logs will run faster. If you include **FE**, all library routines that use exponentials will run faster and, if you include **FT**, all library routines that use trig functions will run faster. You can check the header comments for each routine you are using for info on which compiler options are needed to compile the function in fast mode.

Once you have determined which options you need, just add them to your **SetMathMode** option list and recompile. There is nothing else that you need to do to enjoy the turbo performance.

4.1 Why Use Standard Mode

You might be wondering why you would ever want to use **Standard Mode** when the **Fast Math** modes are so much easier on your CPU. Well, there is some downside to using **Fast Math** and, there are many applications where the extra speed is not really needed.

Each core routine used in fast mode will add about 20 lines of code to your compiled source. So, if you use **FL+FE+FT** it will add about 60 lines of code. In addition, each fast core allocates a block of RAM for its lookup table (32K for **FL**, 20K for **FE**, and 1K for **FT**). This is a rather trivial price to pay if your CPU is gasping for air but, if it’s more or less loafing along, why pay the extra freight?

One other consideration is that fast logs are slightly less accurate (on the order of ±1 ppm additonal error). Ordinarily this will not be noticed but, if your application calls for something like computing the difference between two nearly equal logs, you might fare a little better in **Standard Mode**

However, you can change modes at any time by simply changing a compiler switch. So, choose the mode for each core routine that fits your situation best.

5.0 epVolume Mini Tutorial

This tutorial illustrates how you can use a panel knob, edit box, or slider to control Amplifier volume bi-directionally. If you load the **DB-Demo** instrument you can use any of the three panel controls to set some desired volume level. When you do, the other two panel controls will follow suit as will Kontakt’s corresponding Amplifier Volume knob. Similarly, if you move Kontakt’s Amplifier Volume knob, the three panel controls will all follow suit.

The source code for the script used in the demo instrument can be found in the **Demo Scripts** sub-folder under the **KSE Scripts** folder. The source code is heavily commented in a hopefully instructive way. If you study this script carefully until you understand what each line of code does, you should easily be able to apply the same set of techniques to other **EP Converters** that you might want to use in your application scripts.

Appendix
Using Import As

Most users import the Math Library with the simple import directive as follows:

```
import “KSPMathV610.ksp”
```

However, the Math Library uses a fair number of names both ‘public’ and ‘private’ and, since the KSP provides only a single namespace, it is always possible that you might accidentally use a name in your application script that is already in use by the library. This of course will be flagged when you try to compile your script but then it has to be dealt with.

This problem exists with any import library (not just the Math Library). Usually the easiest way to resolve this issue is for you to simply rename your duplicated-name variables and/or routines. However, if this turns out to be a major job, you might want to consider using the KSE’s ‘**import as**’ directive instead of the simple **import** directive.

For example, you could import the library as Math like this:

```
import “KSPMathV610.ksp” as Math
```

When you do this, the KSE essentially renames everything in the Math Library by affixing **Math** as a prefix. So, all your references to the Math Library now have to be prefixed with **Math.** (Math followed by a *period*).

```
import “KSPMathV610.ksp” as Math

on init
  message(‘’)
  Math.SetMathMode(0)
 { ..... }
end on

Math.on_pgs_do_post

{ ... }
```

Using ‘import as Math’, all you need do to avoid name collisions is not use names beginning with **Math.** (which translates to **Math__** for the KSP). So it should be a lot easier to avoid conflicts with library names. However, there are some drawbacks to this solution.

For one thing all your library references to variables and constants also need to be prefixed. So for example, if your application uses **epVolume**(**E2V**,Vol) you will have to change it to:

Math.epVolume(**Math.E2V**,Vol).

This of course does tend to make your source code a little less readable.

If you reference a library macro to declare some named object, you won’t be able to reference it with the name you gave it. For example, suppose you have the following code in your ICB that declares and references a 2-dimensional array.

```
import "KSPMathV610.ksp"

on init
  SetMathMode(0)
  2dArray(MyArray,6,10)
  message(MyArray[3,4])
end on
```

If instead you import the library as **Math**, here is what that changes to:

```
import "KSPMathV610.ksp" as Math
on init
  Math.SetMathMode(0)
  Math.2dArray(MyArray,6,10)
  message(Math.MyArray[3,4])
end on
```

Note that you name the array **MyArray** but when you reference it, you must use **Math.MyArray**. You may or may not find this a bit confusing.