KSP Math Library

User's Guide

V702

Kontakt 4/5

Robert D. Villwock

7-16-15

What's New In V702 since V6xx

- New converter for **epDlyTime**
- New conditional predicates for:
  pEQ, **pNEQ**, **pLT**, **pGT**, **pLTE**, **pGTE, pRange, and pRange0**

  New inline (one-line) functions:
  Clamp, Clamp0, Min, Max
- New exponential functions with Variable Scaling:
  ExpeVS, Exp2VS, and **Exp10VS**
- New logarithm functions with Variable Scaling:
  LogeVS, Log2VS, and **Log10VS**
- New Inverse Trig Functions: **ACos**, **ASin**, and **ATan**
- Rectangular to Polar and Polar to Rectangular Conversion
- Fast Math options for all the new functions
- Four choices for the Angular Unit used in trig functions.

Deci-Grads	1000 divisions per right angle
Centi-Grads	10000 divisions per right angle
Deci-Degrees	900 divisions per right angle
Centi-Degrees	9000 divisions per right angle

- In addition to the **Ang90** trig constant, the library now also provides **Ang180** and **Ang360**. Moreover, the value of these constants reflect the angular unit you select.
- All the standard trig functions **Sin**, **Cos**, **SinCos**, and **Tan** now accept **any** input angle in the selected angular unit. (The former X versions of these functions have been retired)
- The library now **automatically** generates its required **pgs** callback handler when imported. If your application also requires the pgs callback, a new pseudo-callback directive **on_pgs_changed** is provided for easy coordination.
- New pseudo-directives — **on_post_init** and **on_pgs_changed**

Extremely Important
Please Read All Of This

Before you attempt to use the **KSP Math Library**, be absolutely sure that your **KScript Editor** is set up as indicated below. **DO NOT JUST ASSUME THAT YOUR OPTIONS ARE SET PROPERLY**. If you do not set the options as indicated below, **your compiled code may be very much larger than necessary**.

The **Math Library** is an import module that is compiled with your application script using Nils Liberg's **KScript Editor V1.52** or the equivalent **Sublime Text 3 KSP Plugin**. Throughout the rest of this guide, either of these editors will be referred to as the **KSE**.

Be Sure You Use These Settings

- **Optimize compiled code Enabled**
- **Extra syntax checks Enabled**
- **Compact output Enabled**

NOTE: If you disable 'Extra syntax checks', it disables 'Optimize compiled code' also regardless of whether it indicates it or not. So do not disable 'Extra syntax checks'.

If Your Application Uses the PGS Callback

To avoid conflicts, you should use the new pseudo callback directives **on_pgs_changed** and **end_on** instead of the normal KSP directives **on pgs_changed** and **end on**. Note that the only difference is that the word **on** is connected with an **underscore**. See **Pseudo Callback Directives** in the **Appendix** for more info.

Table Of Contents

Figures and Tables

Appendix

1.0 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. The **KSP Math Library** was written to bridge this gap by providing 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 as shown on **page 3**).

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 2.2 on how to take advantage of the new **on_post_init** callback.

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 V702, 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. You'll find a brief discussion of **Fast Math** in section 4.

There are two demo instruments provided along with the source code for the scripts used in these instruments. The first of these, the **XFadeDemo**, illustrates the use of the crossfade functions discussed in **section 3.4** and will allow you to quickly visualize the curve shapes that are possible with these functions. The second demo instrument and script, the **DB-Demo**, can be used like a **mini-tutorial** on using engine parameter converters (and is also discussed briefly in **section 5.0**).

V702 adds a number of new functions including a complete set of special exponential and logaritmic functions with Variable Scaling **ExpeVS, Exp2VS, Exp10VS, LogeVS, Log2VS,** and **Log10VS**. There are also some nice new conditional predicates **pRange** and **pRange0** and some other one-line functions **Clamp, Clamp0, Min,** and **Max** that you can use inline in your code without restriction. These functions can often avoid the traditional use of if-else clauses to accomplish these same commonly occuring tasks. All in all, **V702** is the most complete and easiest to use version of the KSP Math Library to date.

2.0 Using V700

Besides importing the library, you must also include the **SetMathMode** directive in your script's initialization callback (the ICB). Once you have included **SetMathMode** (specifying the options you want), 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 functions (such as SinCos), output values are returned as a part of the argument list. See KSE documentation for more information on the function types supported.

2.1 SetMathMode(fast_opts, ang_unit)

This directive **must be** invoked near the top of your **ICB** and before any other library references can be made. This directive requires that you specify two parameters: the fast mode option list and an angular unit. The **fast_opts** list is constructed by summing the desired **Fast Math** options. The options currently available and what they do are summarized in **Table 1** and further discussed in section 4. The options (in any combination) are simply strung together with connecting plus signs. If you want to run with the whole library in **Standard Mode**, just use **zero** for the **fast_opts** list.

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 standard trig functions: Sin, Cos, Tan	See section 4
FIT	Use Fast Math for inverse tangent: ATan	See section 4
FIS	Use Fast Math for inverse sine: ASin, ACos	See section 4
ALL	Use all Fast Math options	See section 4

Table 1

The angular unit parameter allows you to specify the angle unit/resolution that will be used by all the trig functions and must be one of the four library options **DG**, **CG**, **DD**, or **CD** as illustrated in Table 2.

Option	Description	Resolution
DG	Use the legacy deci-grads unit,: 4000 divisions per circle	0.1 grad
CG	Use centi-grads: 40000 divisions per circle	0.01 grads
DD	Use deci-degrees: 3600 divisions per circle	0.1 degree
CD	Use centi-degrees: 36000 divisions per circle	0.01 degrees

Table 2

2.2 The post_init Callback

Besides the normal **on init** callback (the **ICB**), the Math Library also provides a special post init callback directive: **on_post_init**. The post init callback runs immediately *after* the ICB exits and, like the ICB, it only runs once. Thus the post init callback acts simply as an extension of the normal ICB.

You might reasonably wonder why you would want to extend the ICB with a separate callback when you could just put the post init code at the end of the normal ICB? The answer is that the ICB has several restrictions as to what kind of code can be written for it whereas the post init callback lifts most of these restrictions. For example, the post init callback will allow you to **call** user functions whereas the ICB does **not** allow this. And, it often happens that some rather lengthy routines (used by other callbacks) have to be executed during initialization also. By moving such initialization sequences to the post init callback, you can simply **call** the common functions without having to repeat their code inline (as you would if you put them in the normal ICB).

For code-space efficiency, the Math Library also makes use of *callable* user functions and thus many library routines cannot be invoked directly from the ICB. However, 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 the post init callback instead. That way, you are free to use KSP user function calls and/or library references as you wish and yet these actions will still be performed with initialization.

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:

```
on_post_init
   read_persistent_var(MyKnob)    { Interestingly enough, this is not needed, see NOTE below }
   RootKnob := Root2(MyKnob)    { Root2 cannot be invoked from the ICB, but it can be here }
end_on
```

While the **Root2** library routine cannot be invoked from the ICB, it can be invoked from the post init callback as shown above. Of course the **on_post_init** callback is *synthesized* by the Math Library and is not a normal part of the KSP language. But, for all practical purposes you can use it just as if it were a normal KSP callback directive. However, please note the word '**on**' in both directives must be connected with an **underscore** as shown.

NOTE: The read_persistent_var (which would be necessary if this code were actually executed in the ICB), is **not** needed when this code is put in the post init callback. Since all persistence vars are loaded automatically after the normal ICB exits, you no longer need to *explicitly* load them. This can often be exploited to avoid a number of **read_persistent_var** statements (such as the one above) which would be required in the ICB.

The post init callback is triggered immediately after the ICB exits. However, if you use any **wait** statements in your post init callback, it is possible that some other callbacks could be triggered and run **before** all of your initialization code finishes. If those *interrupting* callbacks *rely* on your initialization being finished, you may want to test the Boolean conditional **post_init_active** and proceed accordingly. **post_init_active** is **true** all the while post init is running and becomes **false** only after post init finishes. For more info on this, please see the section titled **Pseudo Callback Directives** in the Appendix.

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 need to reference these 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

SetMathMode(fast_opts, ang_unit)	Compiler switches
on_pgs_changed ..... end_on	pseudo callback directive (see Appendix)
on_post_init ..... end_on	pseudo callback directive (see section 2.1)
post_init_active	Read-only Boolean condition

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 re-settable random number generator. Also, at the end of this section, you will find a list of useful library constants that are available for your use.

ACos(X) ASin(X)

ArcCosine and **ArcSine** functions. These functions accept as input **X** (the scaled (10^4) **cos** or **sin** of some angle) and return the principal angle **A** in the current angular unit specified by **SetMathMode**. For **ACos**, the angle will be in the range **0 ≤ A ≤ Ang180**. For **ASin**, it will be in the range from **-Ang90 ≤ A ≤ Ang90**. Input **X** will normally be in the range of **0 ≤ |X| ≤ 1.0000**

ATan(X)

ArcTangent function. This function accepts an input **X** (the scaled (10^4) tangent of some angle) and returns the principal angle **A** in the range: **-Ang90 ≤ A ≤ Ang90**. Input **X** can be any signed integer in the range: **MinInt ≤ X ≤ MaxInt**.

Boolean(X)

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

Clamp(X,a,b) Clamp0(X,m)

Clamp returns the value of **X** limited to the range from **a** to **b** while **Clamp0** returns the value of **X** limited to the range from **0** to **m**. These are one-line functions that can be used inline anywhere in your code.

Cos(A)

This function returns the **Cosine** of angle **A**, where **A** is any angle in the current unit specified by **SetMathMode**. Like all trig functions, **Cos** returns the actual value scaled by 10^4. See also **SinCos**.

Expe(X) ExpeVS(X,d)

The natural (base e) anti-log function. When $\mathbf{X = Z \cdot 10^6}$, **Expe** returns the value of $\mathbf{e^Z}$
ExpeVS returns $\mathbf{10^d \cdot e^Z}$**.** Example: **ExpeVS**(500000,4) = 16487 (interpreted as $e^{0.5}$ =1.6487)

Exp2(X) Exp2VS(X,d)

The binary anti-log function. When $\mathbf{X = Z \cdot 10^6}$, this function returns the value of $\mathbf{2^Z}$
Exp2VS returns $\mathbf{10^d \cdot 2^Z}$**.** Example: **Exp2VS**(-2000000,3) = 250 (interpreted as $2^{-2} = 0.250$)

Exp10(X) Exp10VS(X,d)

The common anti-log function. When $\mathbf{X = Z \cdot 10^6}$, this function returns the value of $\mathbf{10^Z}$
Exp10VS returns $\mathbf{10^d \cdot 10^Z}$**.** Example: **Exp10VS(1650000,2) = 4467** (interpreted as $\mathbf{10^{1.65} = 44.67}$)

Loge(X) LogeVS(X,d)

This function returns the natural logarithm **Ln** (base e) of **X** scaled by $\mathbf{10^6}$. ie $\mathbf{10^6 \cdot Ln(X)}$
LogeVS returns $\mathbf{10^6 \cdot Ln(X \cdot 10^{-d})}$**.** Ex: **LogeVS(250,3)VS = -1386294** (interpreted as **Ln(0.250) = -1.386294**)

Log2(X) Log2VS(X,d)

This function returns the binary logarithm **Lg** (base 2) of **X** scaled by $\mathbf{10^6}$. ie $\mathbf{10^6 \cdot Lg(X)}$
Log2VS returns $\mathbf{10^6 \cdot Lg(X \cdot 10^{-d})}$**.** Ex: **Log2VS(250,3) = -4000000** (interpreted as **Lg(0.250) = -4.000000**)

Log10(X) Log10VS(X,d)

This function returns the common logarithm **Log** (base 10) of **X** scaled by $\mathbf{10^6}$. ie $\mathbf{10^6 \cdot Log(X)}$
Log10VS returns $\mathbf{10^6 \cdot Log(X \cdot 10^{-d})}$**.** Ex: **Log10VS(250,3) = -602060** (interpreted as **Log(0.250) = -0.602060**

Max(a,b) Min(a,b)

Max returns the larger of **a** or **b** while **Min** returns the smaller of **a** or **b**. These are one-line functions that can be used inline anywhere in your code.

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 in an attempt to avoid arithmetic overflow.

pGT(a,b), pGTE(a,b), pEQ(a,b), pLT(a,b), pLTE(a,b), pNEQ(a,b) pRange(X,a,b), pRange0(X,m)

These conditional predictes return 1 when the condition is true and 0 when it's false. For example, if $\mathbf{a > b}$, **pGT(a,b)** will return 1 and when $\mathbf{a \le b}$, **pGT(a,b)** will return 0. **GT** stands for greater than, **GTE** stands for greater-than or equal. **LT** stands for less than and **LTE** stands for less than or equal. **EQ** stands for equal and **NEQ** stands for not equal. The **pRange** predicate returns 1 if **X** is in the range from **a** to **b** while **pRange0** returns 1 if **X** is in the range from **0** to **m**. All these can be used inline anywhere in your code.

P2R(R,A,X,Y) R2P(X,Y,R,A)

Polar to Rectangular and Rectangular to Polar conversion. **R**, the radius vector, can be any integer up to ± 210,000 and **A**, the vector angle, can be any integer value in the current angular unit. When using **R2P**, choose **X** and **Y** so as not to yield an **R** value in excess of 210,000. Since **R** and **X,Y** are always in the same unit/scale, it is often desirable to scale both **X** and **Y** by the same factor before invoking **R2P** since better relative accuracy will generally be obtained with larger values of **X,Y**.

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, the **Power** 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** fast_opts 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 re-settable 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)

These 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**. These one-line functions, can be referenced in-line anywhere in your code.

Sin(A)

This function returns the **Sine** of angle **A**, where **A** is any angle in the current unit specified by **SetMathMode**. Like all trig functions, **Sin** returns the actual value scaled by 10^4. See also **SinCos**.

SinCos(A,sin,cos)

SinCos outputs both the sine and cosine of angle **A** in the current unit specified by **SetMathMode**. **SinCos** is **not** a return-value function. The dual outputs are passed to the variables you stipulate as part of the parameter list. Like all trig functions, **SinCos** returns the actual values scaled by 10^4. See also **Sin** and **Cos**.

Tan(A)

This function returns the tangent of any angle A in the current unit specified by **SetMathMode**. Like all trig functions, **Tan** returns the actual value scaled by 10^4.

Library Constants

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

Ang90

This constant is a right angle in the current angular unit. For example, if you want to use a MIDI CC to cover the range from 0..90 degrees, you could use: angle := CC*Ang90/127.
This expression will remain correct even if you change the angular unit option.

Ang180 Ang360

These constants are a straight angle and full circle respectively in the the current angular unit.

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.

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
FL, FE, FT, FIT, FIS, ALL	Fast math options.
DG, CG, DD, CD	Angular unit options

3.4 Special Functions

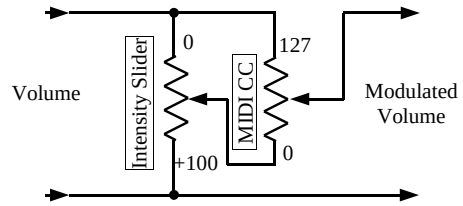
This section is where to look for audio-taper volume control or various morphing crossfade functions.

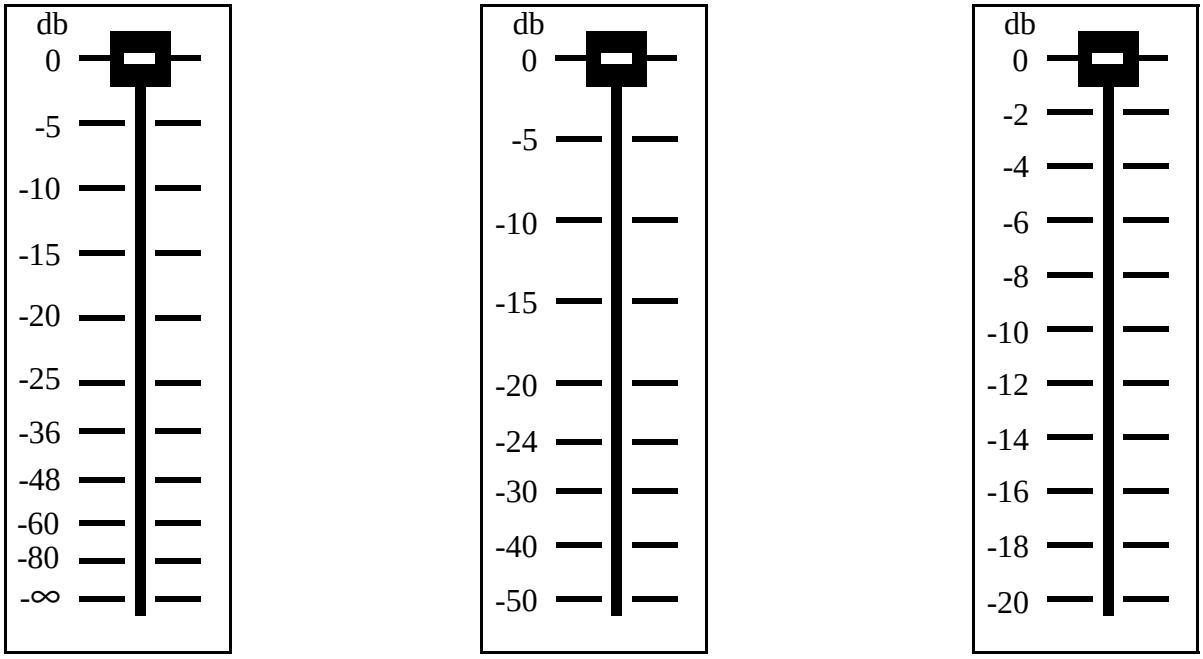
ATFade(cv,rng)

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 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**.

epDlyTime(V2E,T) epDlyTime(E2V,ep)

This routine converts send FX delay time **T** in the range from 5.00 to 2900.00 msec (scaled by 100) 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 cutoff 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 $-1000000 \le P \le +1000000$. 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 type of core routine, 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 the standard trig functions will run faster. If you use **FIT** or **FIS**, the inverse tangent or inverse sine and cosine functions (respectively) will run faster. You can check the header comments for each routine or derivative routine you are using for info on which compiler options are needed to compile that function in fast mode.

Once you have determined which option codes you need, just add them to your **SetMathMode** fast_opts 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 of the five core types used in fast mode will add about 20 lines of code to your compiled source (to initialize the corresponding lookup table). So, if you use fast_opts=**ALL,** it will add about 100 (5x20) lines of code to initialize all the fast mode tables. In addition, each fast core type allocates a block of RAM for its lookup table (32K for **FL**, 20K for **FE**, and 10K each for **FT**, **FIT**, and **FIS** respectively). 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?

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. When in doubt, you can easily try it both ways.

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 be able to apply the same set of techniques to other **EP Converters** you might want to use.

Appendix

A.1 Pseudo Callback Directives

The Math Library requires the use of the pgs callback in order to perform its initialization so, when you import the library, it will *automatically* create a pgs callback with the needed initialization code. This all happens *behind the scenes* and normally you won't have to concern yourself with it. However, there can only be one pgs callback per script so, to allow for the possibility that your application might also require the pgs callback, the library provides the pseudo callback directive **on_pgs_changed**.

Any code you write between **on_pgs_changed** and **end_on** will function just as if you had created a normal pgs callback for your application. However, when you use this pseudo directive, your pgs code will be correctly integrated with the library's pgs callback so they will not interfere with each other. Once post-initialization finishes, the library stops using the pgs callback so it will be devoted entirely to your application if you are using it.

The pgs callback is automatically triggered by the library after the normal ICB exits. In the pgs callback, there is a section of code that will run only one time (just like the normal ICB). This post initialization phase is also available for your application with the pseudo callback directive **on_post_init**. Any code you write between **on_post_init** and **end_on** will be executed one time, right after the normal ICB exits. And, because this code is actually executing in the pgs callback, there are fewer restrictions on what kind of code you can use (compared with the normal ICB). For example, you can use *callable* functions and even *wait* statements if you want.

However, if your post initialization code uses one or more wait statements, keep in mind that it is possible for some other callback to get triggered during such waits. If these *other* callbacks rely on your initialization being completed, you may want to test for this condition and proceed accordingly. For this purpose, the library provides a read-only Boolean named **post_init_active**. This conditional is *true* all during post init but becomes *false* as soon as post init finishes. Therefore, you can use **post_init_active** in an **if** or **while** statement to determine if post init has finished. For example, if you have some waits in your post_init callback and you don't want to allow any played notes to be processed until your post_init callback finishes, you could include some code in the note callback like this:

```
on note
  if  post_init_active
    { post_init not finished yet, ignore note and dismiss callback }
    ignore_event(EVENT_ID)
    exit
  end if
 { proceed with normal callback processing }
end on
```

A.2 Using Import As

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

```
import "KSPMathV700.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 "KSPMathV700.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 "KSPMathV700.ksp" as Math

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

Math.on_post_init
  { your post init code }
Math.end_on

{ ... }
```

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).

Or, if your application uses the pgs callback, you will have to write it this way:

```
Math.on_pgs_changed
  { your pgs callback code }
Math.end_on
```

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

Another complication occurs if you use a library macro to declare some named object. If you do this, you won't be able to reference it with the name you give it. For example, suppose you have the following code in your ICB that declares and references a 2-dimensional array.

```
import "KSPMathV700.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 "KSPMathV700.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.