KSP Math Library

User's Guide

V450

V451/V451M

Kontakt 4/5

Robert D. Villwock

6-25-13

Table Of Contents

Table Of Contents

1.0 Overview of V450

The V450 series of the Math Library is written as an 'import' module to be compiled with your host script using **Nils Liberg's KScript Editor (KSE) V1.5.2** or higher. However, beginning with **V451** the library will also be distributed with a companion **M** version. The **M** versions of the library can be compiled with **V1.5.1 of the KSE** and therefore will be of special interest to **Mac** users since this is the latest version of the **KSE** that is available for the **Mac**. Please see the V451 Readme file for other changes since V450.

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 provided that you enable the 'Optimize compiled code' option in the KSE. The source code for this Library module is heavily commented, so you will also want to enable the 'Compact output' option of the KSE to strip-out these comments (as well as unnecessary white space) from the compiled code.

V450 takes advantage of recent enhancements to the KSE (such as return-value functions and properties) to provide a friendlier and more consistent syntax. Most of the legacy routines have been re-coded more efficiently for both execution time and code space. The format of the engine parameter converters has been improved and many new converters have been added, including support for all the new K5 filters. In addition, there is now a routine to 'linearize' modulation intensity (see **ModInt_to_ep**).

The library now also provides support for 2 and 3-dimensional arrays as well as Packed arrays where each word can be partitioned into several data fields. These functions are supported by the new Property feature added to the KSE. Some new basic math routines have also been added such as a **Root2** function, a general **Power** function, and a new routine named **MulDiv64(x,y,z)** which computes x*y/z and maintains the x*y product as a 64-bit value to address those stubburn arithmetic overflow situations that often arise within the KSP's all-integer arithmetic limitation. A more thorough discussion of the actual library routines and how you can use them is presented in Section 5.

The Log, Exp, and Trig functions are the most CPU-intensive routines. So, if your application uses any of these routines (and/or their derivatives) and you find it a bit taxing for your CPU, you will want to take advantage of the library's Fast-Mode options. Taking advantage of the .nka file loading feature introduced with K4, the Math Library provides an optional **Fast-Mode** version for the Log, Exp, and Trig functions that execute considerably faster (ie have far less CPU demand) while retaining nearly as good an accuracy as their Standard-Mode counterparts. This is accomplished by means of three large arrays loaded from .nka files.

These support .nka files are easily generated or regenerated at any time with utilities provided with the Math Library and you will not need to laboriously type in or copy a long list of intialization values to populate any lookup tables; nor will you need to clutter up your source code with such lists. See sections 2 and 3 for more details about using the Math Library's Fast-Mode and recreating the .nka files.

Since V450 uses KSP-Callable functions for code efficiency, you cannot invoke most library routines from your application's ICB. However, the library provides a simple way to accomplish the same thing. So if you need to invoke library functions from your ICB, please read Section 4 for information on how to use the post-initialization options provided with the Math Library.

Finally, if you are using the new **Task Control Module** (recently added to the KSE) it should be mentioned that many of the Math routines for which overflow can occur, such as trying to compute the Log of X <= 0, can optionally report a Math Overflow TCM exception. For details on how to utilize this option, please see section 6.

2.0 Using V450

PLEASE NOTE

The Math Library is written as an 'import' module and requires that you use Nils Liberg's KScript Editor (KSE) V1.5.2 or higher to compile your application. Your host script should include the directive **import "KSPMathV450.txt"** ahead of your Initialization Callback, **ICB**. You should also enable the **Optimize compiled code** and the **Compact output** options of the KSE. Enabling compact variables is optional however, **you must use the new compiler** (not the optional old compiler).

ALSO PLEASE NOTE

With V450, you cannot invoke library functions directly from your script's ICB. However, the library provides an easy way for you to accomplish the same thing. If your initialization code needs to invoke library functions, please refer to **Section 4** for how to accomplish this.

There are basically two things you need to do before you can use **V450**. The first is to invoke the macro function **SetMathMode** in your script's ICB. The second, provided you want to use the fast mode for any routines, is to setup a resource **Data** folder for your hosting instrument and include the required .nka files in that **Data** folder. These steps will be discussed in **Sections 2.1** and **2.2** respectively.

2.1 SetMathMode

This function must be invoked in your ICB before any library references can be made. This macro function requires you to provide an **option_list** parameter which is used to control certain compile-time features of the Math Library and is constructed by summing the desired option codes. There are currently a total of 6 option codes available for your use as shown in **Table 2-1**.

Option Code	Compilation Feature
HX	Compile support data needed for the CVHex routine
F1	Compile the Fast-Mode routines for all Log functions referenced
F2	Compile the Fast-Mode routines for all Exp functions referenced
F3	Compile the Fast-Mode routines for all Trig functions referenced
RI	Randomly Initialize the seed used by the Rand function
PI	Trigger Post Initialization when your ICB first exits

Table 2-1

SetMathMode performs a number of conditional compilation and intialization steps among which are:

1. Handling **special** options, such as when you include **HX, RI** or **PI** in the option_list.
2. Declaring the associated data arrays for each fast-mode code specified.
3. Pre-loading each of these arrays from their corresponding .nka files in the resource **Data** folder.
4. Creating a mode-control, bit-flag constant that the library will use to determine whether to compile the standard or fast mode versions for library routines referenced by your application.

To further clarify all this, it will be convenient to use an example. Suppose that you have included **SetMathMode(F1+F3+PI)** in your script's ICB. If your script then references **SinCos**, the libary will compile the fast version of **SinCos** (because **you included F3** in the option_list). However, if your script references **Exp2**, the library will use the standard version because you did **not** include F2 in your option_list.

Finally, because you included **PI** in the **option_list**, a pgs callback will be triggered immediately upon exit from the ICB. Please refer to **Section 4.0** for a discussion of how to utilize this option to execute library functions as part of your script's initialization process. The purpose for the **HX** option is discussed in section 5.4.1 while a more detailed discussion of the Fast-Mode codes **F1, F2**, and **F3** will be presented in Sections 2.2 and 2.3.

It is important to realize that when constructing the summation of option codes for the **SetMath-Mode** parameter, you should **never use an option code more than once** in the summation. For example, even if your application references several trig functions, you should only include F3 one time in the **option_list** summation. Using something like **SetMathMode(F3+F3) will lead to erroneous behavior**. If your application doesn't require any of the special options and you are not using any fast-mode routines, you can simply use zero for the option_list, i.e. **SetMathMode(0)**. When you do this, all Log, Exp, and Trig functions will be compiled in their standard modes and no post-initialization code will be triggered.

2.2 The nka Files

The option_list codes F1, F2, and F3 (in addition to compiling the fast-mode versions for the Log, Exp, and Trig functions), also cause **SetMathMode** to create the supporting data arrays and load them from the .nka files expected to be found in your instrument's resource **Data** folder. So, in addition to including the option_list codes F1, F2, or F3, you also need to be sure the corresponding .nka files are in the host instrument's resource **Data** folder. The .nka file names that correspond with the option_list codes **F1**, **F2**, and **F3** are **FLog.nka**, **FExp.nka**, and **FSin.nka** respectively. All 3 of these .nka files are distributed with the library and can be found in its Data folder but you can easily recreate them yourself as discussed in Section 3.0.

If you aren't already familiar with how to create the resource **Data** folder for an instrument, the process is described on page 148 of the **KSP Reference Manual (for K4)**. Step by step instructions for how to create the .nka files are also presented in Section 3.4 but, assuming that you already have the .nka files, once the **Data** folder is created and your host instrument is pointing to it, you can simply copy the needed .nka files into the host instrument's resource **Data** folder. If you do not intend to use the library in fast mode at all, you can dispense with setting up the resource Data folder and of course, copying any .nka files to it.

2.3 Missing Or Corrupted nka files

In order for the fast-mode versions of the math routines to work properly, the support .nka file must be present in the hosting instrument's resource Data folder and the instrument must be pointing to it. Since arrays loaded from .nka files are inherently persistent, once they are properly loaded for a given instrument, the arrays will retain the correct data even if the .nka files are deleted. However, if you should do anything to clear the persistence buffer, it is essential that the .nka files be present the next time you run. If the .nka files are missing (or if the host instrument is not properly pointed to the resource Data folder), the fast math routines will not return correct results. If Kontakt cannot find one of the .nka files, it will *usually* (but not always) report an error with a status-line message. However, such messages are often unnoticed or unheeded. Therefore, it would be better if your host script watches out for this condition and at least displays a more prominent warning to the user and/or refuses to run until the nka file is present.

To accomplish this, your host script can read the library variable named **NKA_Status** which will contain a zero value when all needed nka files have been loaded successfully. However, if one or more of your needed nka files appear to be missing or corrupted, **NKA_Status** will return a non-zero value. Thus, your script can test **NKA_Status** and display an error message or take other appropriate action when there is a problem. For example, you could use a code fragment something like the following.

```
if NKA_Status # 0 { one or more needed nkas didn’t load correctly }
        DisplayWarning
        HaltScript
end if
```

NKA_Status is actually a bit-map of the file-code numbers that didn’t load properly. If FLog (F1) didn’t load correctly bit 1 (2^1) will be set, if FExp (F2) didn’t load correctly, bit 2 (2^2) will be set, and if FSin (F3) didn’t load correctly bit 3 (2^3) will be set. So, for example, if your script uses F1+F2+F3 in the option list and none of the three .nka files are loaded, the value of NKA_Status would be 2+4+8 = 14. Therefore, if your host script uses more than one nka file, your script can even report which files are missing by decoding the **NKA_Status** bits accordingly. Note that even if your script doesn’t query the NKA_Status, the Math Library will still report a non-zero value by displaying the status line message **'WARNING: NKA File(s) Missing'**

3.0 Regenerating the .nka Files

In the **Demo-Instruments** folder, you’ll find an instrument file named **NKA_Writer.nki** that you can load into K4/5 to write any .nka file with a single mouse click as discussed in **section 3.1**. **NKA_Writer** uses a simple script (that you’ll find in the **KSE Scripts** folder) named **NKA_Utility.txt**. You can easily recreate both this script and custom variations of the **NKA_Writer** instrument, see **sections 3.3** and **3.4**.

3.1 Using the NKA_Writer

To use the **NKA_Writer Instrument** to write any of the three nka files, launch Kontakt and then load the **NKA_Writer.nki** that you’ll find in the **Demo-Instruments** folder. Use the drop-down menu to select the nka file you want to write. The file name will blink for a few seconds to let you know the file has been written. You will find the nka file in the **Data** folder alongside the instrument. You can repeat this for each nka file you wish to write. This demo instrument uses the **NKA_Utility.txt** script which you will find in the source-code folder named **KSE Scripts**.

3.2 The File Naming Problem

As written and compiled, the script in slot 1 of the **NKA_Writer** instrument will always write .nka files using their standard names **FLog.nka**, **FExp.nka**, and **FSin.nka**. However, if you import the Math Library with **import ‘as xxx’**, all library references including the nka array names, need to be prefixed with **‘xxx.’** Unfortunately, you can’t just rename the standard .nka files because Kontakt embeds the original array name in the file itself so you can’t load it into an array with a different name. So, for example, if you write the standard **FSin.nka** file and then decide you want to use the Math Library by importing it with with the directive **import “KSPMathV450.txt” as Math**, you can’t just rename the **FSin.nka** file and use it. Instead, you must rewrite the file with the required name **Math__FSin.nka**. However, it is very easy to edit the **NKA_Utility.txt** script to accomplish this as will be discussed in **section 3.3**.

3.3 The NKA_Utility Script

If you examine the **NKA_Utility** script source code, you will see that it consists of only two lines of code. The first line merely imports the Math Library and the second line invokes the library macro named **write_nka**. As supplied, the **NKA_Writer** instrument (which uses this script) will write the nka files with their standard names **FLog.nka, FExp.nka,** and **FSin.nka**. However, if your host script will import the library ‘as xxx’ the nka file names need to be named **xxx__FLog.nka**, **xxx__FExp.nka**, and **xxx__FSin.nka**. To write the nka files with the correct names for the way your host script imports the Math Library, you need to modify the NKA_Utility script and then rewrite the nka files. For example, if your host script will import the Math Library ‘as Math’, the two lines of code will need to be edited to look like the following.

```
import "KSPMathV450.txt" as Math
Math.write_nka
```

Of course when you import the library 'as Math', all library references have to be prefixed with **Math.** and therefore you must invoke **write_nka** as **Math.write_nka** as shown above.

3.4 Making the NKA_Writer Instrument

To make your own **NKA_Writer** Instrument from scratch at any time, proceed as follows. Create an empty file folder, named **NKA**, in some convenient place and then put copies of the KSE source files named **KSPMathV450.txt** and **NKA_Utility.txt** in this new folder. In Kontakt, load a new default instrument and then save it, with the name **NKA_Writer.nki**, in the NKA folder. Save this instrument as patch only (no samples). Now create a new **Resources** folder (with a **Data** folder alongside) as follows. In instrument options, the Resource Container should show as <none>. Click the **Create** button and navigate to your new NKA folder and then enter the name **NKA_Writer**, choose the .nkr file type and click **Save**. Kontakt will present a dialog asking if you want to create a resource folder. Answer Yes and then click OK. The Resource Container box should now show **NKA_Writer.nkr (read from the resources folder)**. Close the Instrument Options dialog and resave the instrument.

If you have done this correctly, your NKA folder will now contain five files (KSP_Math.txt, NKA_Utility.txt, NKA_Writer.nki, NKA_Writer.nkr, and NKA_Writer.nkc) and two folders named **Re-sources** and **Data**. You can now delete the **Resources** folder and its contents but *don't* delete the .nkr file or the **Data** folder (which is ultimately where your .nka files will be written). With the **NKA_writer** instrument still loaded in Kontakt, Launch the **KSE** and use it to open the utility script named **NKA_Utility.txt**. In the settings menu of the KSE, enable Compact output and Optimize compiled code and then hit **F5** to compile the **NKA_Utility** script and put it in the clipboard. Paste this compiled script into the first script slot of Kontakt and hit Apply. Enter an appropriate name for the script title such as 'Write Std nka files'. You can now put Kontakt into the performance view and resave the completed **NKA_Writer** Instrument.

This instrument can be loaded into Kontakt at anytime in the future that you need to write a new .nka file or files. The files you write will always be placed in the **Data** folder and you can then copy these files to the resource **Data** folder of whatever instrument you are working on at the time.

Of course this version of the **NKA_Writer** instrument only writes the nka files with their standard names (just as does the instrument provided in the Demo-Instruments folder). However, if you frequently import the Math Library as *something*, probably the best way to handle this is to compile an additional **NKA_Utility** script variation and put it in the 2nd script slot of the **NKA_Writer** instrument. You can then use this same utility instrument to make your nka files for any future project (unless you use yet a different 'import as' name of course).

For example, if you routinely import the Math Library as 'Math', you could make up an additional NKA_Utility script as follows. First edit the existing NKA_Utility script as was illustrated in section 3.3. Compile this script and paste it into the 2nd script slot of the NKA_Writer instrument. Then, enter a title for this script slot, maybe something like 'Write Math.nka files'. Then, put the instrument back in performance view and save it again. Now you can use this NKA_Writer to write either the standard nka files or those prefixed with Math depending on whether you run the script in slot 1 or slot 2 respectively.

Similarly, if you frequently import the library with one or two other prefixes, you could create several more versions of the NKA_Utility script and compile and load them into the remaining script slots of your NKA_Writer instrument.

4.0 Invoking Math Functions During Initialization

KN functions (Native Kontakt User Functions) have a few limitations, one of which is that they cannot be used in the ICB. Since most of the Math Library functions now utilize KN functions for code efficiency, this limitation means that you cannot generally invoke library functions directly from your ICB.

However, if some part of your initialization process requires that you invoke library functions (or for that matter any other callable functions), there is an easy way for you to accomplish this without directly invoking the routines from your script's ICB**.** What can be done, instead of executing the functions inside of the ICB, is to execute them as part of a post-ICB operation that takes place immediately after the ICB exits. This post-ICB operation is executed in the pgs callback (one time only) and thus this process essentially 'extends' the ICB into a final 'zone' where **it is permissible** to execute KN functions.

When you include **PI** in the option_list of **SetMathMode**, a pgs callback is triggered immediately upon exit from the ICB. In addition, an internal flag (post_icb_math) is set and this flag will be detected in the pgs callback with one of a pair of supplied macros (more on this in the next paragraph). These macros can then invoke any KS or KN function of your choosing and after it executes, the post_icb_math flag will be reset so that your initialization function will not run again (should there be any further pgs callbacks). The KS/KN function that you provide can contain any kind of KSP code, including 'calls' to KN functions and therefore you can also invoke any desired Math Library functions

Two macros are provided to handle post initialization. If your script already has a pgs callback, you can invoke the macro **do_post_init(init_rtn)** anywhere within the body of the pgs callback, preferably near the beginning. Alternatively, if your script doesn't use a pgs callback, you can instead invoke the macro **on_pgs_do_post(init_rtn)** anywhere after the ICB (but not inside another callback). This macro will essentially create the pgs callback header/trailer **and** embed the equivalent of the do_post_init function as well.

The **init_rtn** parameter of these macros should be the name of a KS function you write to handle all of the initialization that must be done outside of the ICB. The body of the KS function should contain all the things you need to do for the post-initialization process. Of course you can include invokation of any Math Library functions and/or 'calls' to various other KN functions that you may have written. Alternatively, if you want to write **init_rtn** as a KN function, simply prefix its name with the **call** keyword when you invoke do_post_init or on_pgs_do_post. For example, if your KN initialization function is named, **app_init**, you would invoke either **do_post_init(call app_init)** or **on_pgs_do_post(call app_init)**.

This post-initialization feature can be useful for things other than invoking Math Library functions. For example, it's not unusual that certain lengthy callback routines also need to be executed in the ICB as part of the initialization process. And, if you have to do this with only KS functions, a lot of redundant inline code can result. Instead, you can code these lengthy routines as KN functions and 'call' them where needed. You can side-step the non-ICB limitation of KN functions by including calls to these functions in the post-initialzation routine.

Post initialization has an additional advantage in that persistent variables are loaded by Kontakt **after** the normal ICB finishes. Many times this requires that certain initialization steps have to be preceded by read_persistent_var commands. However, if these initialization routines are placed in the post-initialization code, all persistent vars will already have been updated as a matter of course. Thus you will not need to explicitly use read_persistent_var commands to accomplish this.

5.0 The Library Functions

The source code of the KSP Math Library is divided into 4 main sections as follows:

5.1 Utility Macros
5.2 Basic Functions
5.3 Specialty Functions
5.4 Format Conversion

These four sections are followed by a number of internal support sections. However, your application will ordinarily only reference the functions in the first four sections of the source code. Normally you won't need to concern yourself with any of the support routines in the source code unless you intend to modify the library in some way. Each of the four source code sections will be discussed in the corresponding sections 5.1 through 5.4 of this User's Guide which follows next.

5.1 Utility Macros

This section currently includes **do_post_init**, **on_pgs_do_post, SetMathMode**, and **write_nka** which were discussed in sections 4.0, 2.1, and 3.3. The remaining macros will be discussed next.

5.1.1 **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 these arrays are declared they can be referenced with a standard, multi-dimensional array syntax. For example:

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

The reference indices can be either constants or variables but please note that no runtime range checking is performed so your application must insure 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 arrays must not exceed 32,768 (the current size limit for KSP arrays).

5.1.2 **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 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.

You specify the field format 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
```

5.2 Basic Functions

This section currently contains 24 generally useful math functions that you can use to support your application code. Here you will find such things as extended arithmetic and transcendental functions along with some miscellaneous functions such as conditional predicates and a resetable random number generator.

5.2.1 Extended Arithmetic

Cabs(X)

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

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 $Y = Y_1 \bullet Y_2$ and then doing something like $X \bullet Y_1 / Z \bullet Y_2$ to avoid arithmetic overflow.

RoundDiv(X,Y)

The divide operation in the KSP always results in a truncated quotient. However, it is often desireable to round the quotient to the 'nearest' integer. This is farily easy to do when the quotient is positive by simply half-adjusting. But, when signed numbers are involved, the rounding process becomes trickier. **RoundDiv(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.

Caution: The KSE does arithmetic somewhat differently than the KSP. So if you try the above illustration with literal values, you will get different results when the KSE is in optimize mode versus when it isn't. Of course you normally wouldn't use this function with literal values for X and Y.

Note also that RoundDiv is a one-line function so it can be referenced anywhere in your code.

5.2.2 Log and Exponential

Loge, Log2, Log10

These functions return $10^6 \cdot \text{Log}_e(X)$, $10^6 \cdot \text{Log}_2(X)$, and $10^6 \cdot \text{Log}_{10}(X)$ respectively.

Expe(X), Exp2(X), Exp10(X)

These anti-log functions can be used to compute e^Z, $2^{Z,}$ and 10^Z respectively. Where $Z = X \cdot 10^{-6}$

5.2.3 Powers and Roots

Power(X,n,d,dd)

This general power function allows you to compute $X^{n/d} \cdot 10^{dd}$ where X, n, and d are positive integers. The routine uses Log/Exp 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 F1+F2 in your **SetMathMode** option_list.

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. See the header comments of these routines for more detail.

5.2.4 Trig Functions

Cos(X), Sin(X), Tan(X)

These functions return the Cosine, Sine, and Tangent of X respectively. Where X is a first-quadrant angle in deci-grads (1000 dg = 90°). Note all trig functions return the actual value scaled by 10^4.

XCos(X), XSin(X), XTan(X)

These are the extended-angle versions Cos, Sin, and Tan where X can be any signed integer in deci-grads.

SinCos(X,sin,cos), XSinCos(X,sin,cos)

SinCos outputs both the Sin and Cos of a first-quadrant angle X in deci-grads. **XSinCos** is the full angle counterpart which accepts *any* integer value for X in deci-grads and outputs both the Sin and Cos of X.

NOTE: SinCos and XSinCos are **not** return-value functions. The dual outputs are simply passed to the variables you stipulate as part of the parameter list.

5.2.5 Miscellaneous

Rand(a,b)   ResetRand

Rand(a,b) returns a random value between a and b inclusive providing the same function as that of the KSP **random** function. However, **Rand** is resetable by invoking **ResetRand**. Therefore, you can use this generator whenever you need a 'repeatable' sequence of pseudo-random numbers.

Boolean(X)   Sign(X)

These are conditional predicates. **Boolean(X)** returns 0 when X = 0 and 1 otherwise. **Sign(X)** returns +1 when X ≥ 0 and -1 when X < 0. Since these are one-line functions, they can be referenced anywhere in your code.

5.3 Specialty Functions

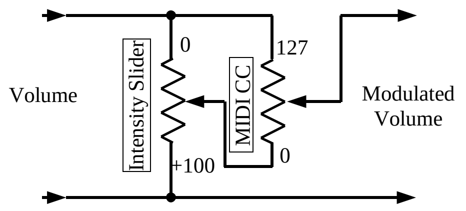
This section currently contains **ATFade** and four morphing Crossfade functions.

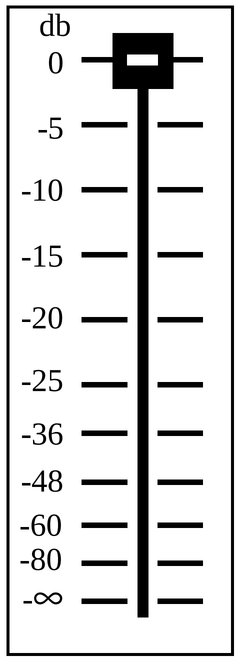
5.3.1 ATFade(cv,rng)

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 2 parameters as follows: **ATFade(cv,rng)**. **cv** is an input value from **0..127** from the desired MIDI CC. **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.

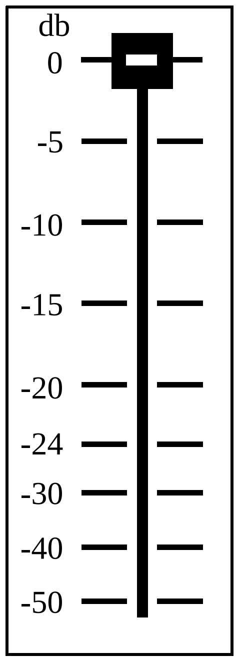
The electronic equivalent of Kontakt's MIDI CC control is shown in **Figure 5-1**. The **ATFade** function operates in a similar way except that with **ATFade**, the contouring of the MIDI CC is more like that of a traditional 'audio' fader whereas in Kontakt, the contour of the potentiometers is simply 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 to the full size (see **Figure 5-2**). With such tapers, you can easily control volume more expressively.

Figure 5-1
Electronic Equivalent For Kontakt's Positive Modulation

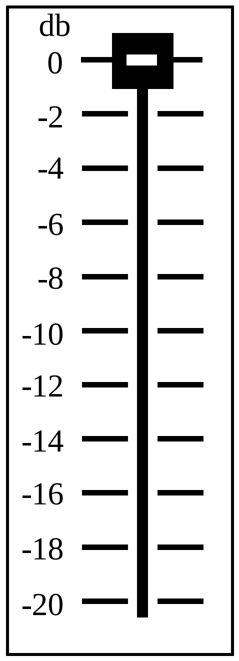




Range = 100%



Range = 80%



Range = 50%

Figure 5-2
Audio Contours
at various 'Range' settings

5.3.2 Morphing Crossfade Functions

There 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 2nd-degree exponential 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 0 to 100% to watch its effect on the shape (the Morph knob swings **mv** from 0 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% (the XFade 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 for input to the library functions **VR_to_mdb** and **epVR** so you can use the VR outputs with **VR_to_mdb** to crossfade events via **change_vol** or you can use them with **epVR** to perform group volume crossfades via engine pars.

Using A Negative Morphing Value

These routines were 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%.

If you'd like to see what the curves look like with negative **mv** values, the V451 package has a modified **XFade Demo** instrument where the **Morph** knob 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.

5.4 Format Conversion

In this section you will find all the engine parameter converters as well as miscellaneous, non-ep format converters.

5.4.1 Miscellaneous Format Converters

CVHex(val)

CVHex returns an 8-digit, hexadecimal text string representation of the input integer value. **NOTE:** Besides referencing **CVHex** in your application, you must also include **HX** in the **SetMathMode** options_list in order to compile the hex character array needed for proper operation of CVHex.

D.FmtVal(val,dd)

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

VR_to_mdb(VR)

This routine returns the **mdb** equivalent of the scaled input volume ratio, $\mathbf{VR = 10^4 \cdot V/V_0}$. 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 legacy routine named **Get_db** except that **VR_to_mdb** allows a wider **VR** range from 0.0000 to 4.0000 (vs 0.0000 to 1.0000).

5.4.2 ModInt_to_ep(type,P)

This function will convert P = % modulation intensity (scaled by 10000) to the equivalent engine parameter. Thus this function essentially linearizes control of the modulation intensity slider. This routine works with either ENGINE_PARAMETER_INTMOD_INTENSITY or with MOD_TARGET_INTENSITY by specifying a **type** parameter as one of the library constants **IMOD** or **TARG** respectively. For use with type = IMOD, the linear control parameter **P** can be in the range of $-1000000 \le \mathbf{P} \le +1000000$. When **P** is negative, the modulator’s **Invert** button will be activated. When using 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 parmeter command to activate the **Invert** button.

5.4.3 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 with the parameter list. There are two library 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. Wheras **epVolume(E2V,ep)** will return the equivalent volume in mdb for the input **ep** value. Also please note that all bi-directional converters begin with **ep**, such as **epVolume**, **epAtkTime**, etc.

Miscellaneous Bi-Directional Converters

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

epTune(mode,T) This routine converts instrument/source tuning in the range from -3600 to +3600 cents to/from the equivalent ep.

epVolume(mode,vol) This routine converts instrument/group volume in the range from $-\infty$ to +12000 mdb to/from the equivalent ep.

epVR(mode,VR) This routine converts scaled volume ratio ($10000 \cdot V/V_0$) in the range from 0.0000 to 4.0000 to/from the equivalent ep for group/instrument volume control

AHDSR Envelope Converters

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

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

EQ Parameter Converters

epEqBW(mode,bw) This routine converts the Bandwidth (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(mode,F) This routine converts Frequency (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(mode,G) This routine converts Gain (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.

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

epSGEqFreq(mode,band,F) This routine converts the LF/LMF/HMF/HF frequency (for the Solid-G EQ) to/from the equivalent ep. Conversion direction is specified with **mode = V2E** or **mode = E2V** (as with all the bi-directional converters). 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(mode,G) This routine converts Solid-G EQ Gain (for LF/LMF/HMF,HF) in the range from -20.0 to +20.0 db (scaled by 10) to/from the equivalent ep.

Filter Cutoff Frequency Converters

epARFreq(mode,F) This routine converts cufoff frequency in the range from 8.2 to 35500.0 Hz (scaled by 10) to/from the equivalent ep. This routine is useable with AR-LP2, AR-LP4, AR-LP2/4, AR-HP2, AR-HP4, AR-HP2/4, AR-B2, AR-B4, and AR-B2/4.

epHBFreq(mode,F) This routine converts cufoff frequency in the range from 36.1 to 18100.0 Hz (scaled by 10) to/from the equivalent ep. This routine is useable with the **Legacy** filters HP1, HP4, BP2, BP4, and BR4. For **HP2** use **epHP2Freq**.

epHP2Freq(mode,F) This routine converts cufoff frequency 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.

epLPFreq(mode,F) This routine converts cufoff frequency 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(mode,F) This routine converts cufoff frequency 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.

epSVFreq(mode,F) This routine converts **SV** and **Ladder** cutoff frequency 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(mode,F) This routine converts the **Multi SV** filter cufoff frequency 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.

epV3x2Freq(mode,F) This routine converts the **Multi** filter cufoff frequency in the range from 26.0 to 15800.0 Hz (scaled by 10) to/from the equivalent ep. This routine can be used with the **Versatile 3x2** filter.

6.0 Math Overflow and TCM Exceptions

Generally, when the Math Library routines incur an overflow situation, it will be indicated by returning either MaxInt or MinInt (whichever is most appropriate). For example, if you attempt to compute Log2(0), MinInt will be returned. In general such results are the closest to the real answer that can be represented in a 32-bit signed integer. However, when debugging your script, such overflow conditions may go unnoticed so it would be better if you could more easily detect such situations and correct them.

If you are using the new **Task Control Module**, (TCM) now included in the KScript Editor, (KSE) and you are using the **TCM Debug** option, the Math Library will report math overlow conditions as a TCM Exception using the code **MATH_OVERFLOW**. If you design your TCM exception handler to watch for this code, you will be able to detect any math overflow that occurs during program debugging (see section 6.4 of the TCM User's Guide).

PLEASE NOTE: In order for the Math Library to 'know' that you are running in TCM_DEBUG mode, you must include the **SET_CONDITION(TCM_DEBUG)** directive **before** importing the library (instead of in the more usual ICB location). Your preamble code should be ordered something like this.

```
SET_CONDITION(TCM_DEBUG)
import "KSPMathV450.txt"

on init
  message('')
  tcm.init(50)
  SetMathMode(0)
        { other initialization code }
end on
```