

KSP Math Library

User's Guide

V200

R. D. Villwock

4-19-08

Table Of Contents

1.0 Using the KSP Math Library

1.1	Introduction.....	3
1.2	The Math Library Package	3

2.0 Equal-Power Crossfading Tutorial

2.1	Background and Theory	4
2.2	Using the change_vol Function	6
2.3	Wrapping It Up.....	7
2.4	The First Demo Instrument.....	8
2.5	Setting Up Your Speakers	8
2.6	The Panning Demo	8
2.7	The Vel Xfade Demo	9
2.8	Strolling Through the Source Code	10
2.9	The NewDemo and NewVolume Routines.....	10
2.10	About the Library Functions	12
2.11	Pseudo Calling.....	12
2.12	Callback Coding.....	14

3.0 The ATFade Function

3.1	Description	16
------------	--------------------------	-----------

4.0 Crossfading Instrument Groups

4.1	Introduction.....	17
4.2	The 2nd Demo Instrument	17
4.3	Running The Demo	17
4.4	The Source Code	17

Appendix

	The change_vol Bug	18
--	---------------------------------	-----------

1.0 Using the KSP Math Library

1.1 Introduction

The **KSP Math Library** contains a number of trigonometric and logarithmic functions which may be useful in a variety of Scripting Applications such as formant-corrected pitch bending, velocity crossfading, various forms of tempo/pitch computation, etc. In addition, the library provides a number of very useful ‘volume control’ conversion routines. For example, there are routines to convert volume ratio data to db or volume ratio data to the corresponding Engine Parameter values, etc. Therefore, you can easily perform an equal-power crossfade of two notes or even two Kontakt Groups if desired.

By using the latest features of **Nils Liberg’s KScript Editor V1.25.6**, the **KSP Math Library** can be **imported** into your host script without ‘bloating your code’. No data or code will be added to your script other than that which is needed for the functions you actually make use of. If your script doesn’t call on any of the library’s functions, **no data or code** will be added to your script. Yet, all of the library routines are always ‘on tap’ should you need them.

The source code for this library is well-commented and each routine is preceded by a header comment block which details what the routine does and how your code needs to interface with it. It is not necessary for you to understand how these routines work in order to benefit from them. It is only necessary that you know the function they perform and how to interface with them. However, to help you to visualize how these functions can be used to do something practical, this **User’s Guide** contains some hands-on tutorials. These tutorials (which begin in Section 2.0) illustrate how to use the library routines to implement equal-power crossfading, **EPXF** for several important applications. The **EPXF Tutorials** use a pair of small instruments and a few demo scripts that you can experiment with. The source code of the demo script can be used, in conjunction with the tutorial text, as guidelines for incorporating **EPXF** into your scripts.

1.2 The Math Library Package

The **KSP Math Library** is supplied in source-code form and can be viewed in any text editor but it is best viewed in the **KScript Editor**. Moreover, the source code is not intended to be run by itself; the library must be **‘included’** in a host script by means of the **import** statement. If any of the identifiers used in the library are in conflict with names that you have used in your host script, simply use the **import as** feature of the **KS Editor**. For example, you could use **import “KSPMathV2xx_KSM.txt” as BBM** and then simply reference the library’s functions and constants using a **BBM.** ‘prefix’. The full KSP Math package contains:

- | | |
|-----------------------------------|---|
| 1. KSP Math Library Source Module | KSPMathV2xx_KSM.txt |
| 2. EPXF Tutorial Demo Scripts | XFadeDemo#1_KS.txt and XFadeDemo#2_KS.txt |
| 3. EPXF Tutorial Demo Instruments | XFadeDemo#1.nki and XFadeDemo#2.nki |
| 4. User’s Guide File | KSPMathUserGuide.pdf |
| 5. Technical Guide File | KSPMathTechGuide.pdf |
| 6. K2 Panning Test Source file | K2PanTest_KS.txt |

You won’t need to read the **Technical Guide** if you just want to use some of the Math Library’s functions in your scripts. However, if you want to extend the Library or would like to modify some of its routines, or if you are just interested in learning something about how the Math Library works, the **Technical Guide** will be helpful. The **Technical Guide** discusses the algorithmic details of each library routine and provides useful information on how to change angular units, input/output scaling, overall precision, etc. Overall, the **Technical Guide** is just the sort of reading material you will want to curl up in a corner with; on some long, rainy evening. ;-)

2.0 Equal-Power Crossfading Tutorial

2.1 Background and Theory

One of the earliest applications of **EPXF** technology, was using it to solve a classic panning problem. With the introduction of consumer stereo playback equipment (in the late 50s to early 60s), mixing boards added a stereo L/R bus and with it, each input strip now included a Pan Pot (Panorama Potentiometer). Of course the idea was to take a bunch of mono sources and spread them between the consumer's left and right loudspeakers to create a stereo panorama. Combined with the rise of multitrack recording, mixdown slowly but surely became a post-production operation.

The idea behind panning is very simple. If you send your mono signal only to the left bus, it will ultimately come out of the consumer's left speaker and sound like it's coming from that direction. Similarly, if you send the signal only to the right bus, the sound will appear to be coming from the right side of the room. If you send equal amounts of the mono signal to both the left and right buses, the sound will appear to come from between the speakers (ie from the center). Early Pan Pots were simply two linear potentiometers ganged together on one concentric shaft. The pots were wired oppositely so that as one was increasing its output, the other was decreasing its output and vice-versa.

If we were to make a plot of the left bus signal and right bus signal as we rotate such a pan pot from min to max, we would get the classical linear crossfade graph as shown by the **Green Lines** in **Figure 1**. But there was a problem with this implementation. As you move a source from the extreme left to the extreme right, you would like the volume to stay the same. For example, you may have painstakingly mixed a bunch of instruments with a trumpet near the left side and now you decide you'd like it better if the trumpet was in the center. What you would like to do is simply turn your trusty Pan Pot from left to center and be done with it. However, it was soon discovered that using linear crossfading produced a noticeable drop of volume (about 3db) in the center of the field. So as you swing slowly across the panorama from full left to full right, the ears hear the volume dipping in the middle and then increasing again as you move to the right side.

Users with lower-cost mixing boards pretty much had to suffer with the problem, but this would never do for the pros buying very pricey boards. It wasn't long before high-end console makers concluded they needed to use some kind of non-linear taper for their pan pots or else use some compensating circuitry to level-out this 'hole in the middle'. It turned out that the solution to this '3db dip' panning problem, as it got to be known, was to use **EPXF** for reasons which I'll discuss shortly. I've also implemented a panning demo for you to experiment with when we get to the hands-on part of this tutorial. Out of curiosity, I checked K2's panning using the KSP's **change_pan** function. Since NI has implemented a lot of volume control and fade functions linearly, I expected to hear a 3db dip in the center of their panning function. But, strangely, NI has actually **over-compensated** with its rather peculiar panning implementation in K2 and **you can hear a definite lift of volume in the center!** You'll be able to hear this for yourself later as you exercise the various demos I've prepared.

Before returning to the classic 3db dip problem, we need to adopt some notational conventions. Whenever we are doing a crossfade, there are always two signals involved. In K2 it is most often a pair of notes or samples that are both playing and we want to 'morph' from one to the other. Instead of numbering these as 1 and 2, I'm going to refer to these as 0 and 1. This will translate better later since the KSP uses zero-based arrays and we will often use 2-element arrays to hold the various crossfade parameters for signal 0 and signal 1. Now, for the panning example we are discussing, the two signals are the left bus and right bus voltage but we'll just refer to these as V_0 and V_1 respectively.

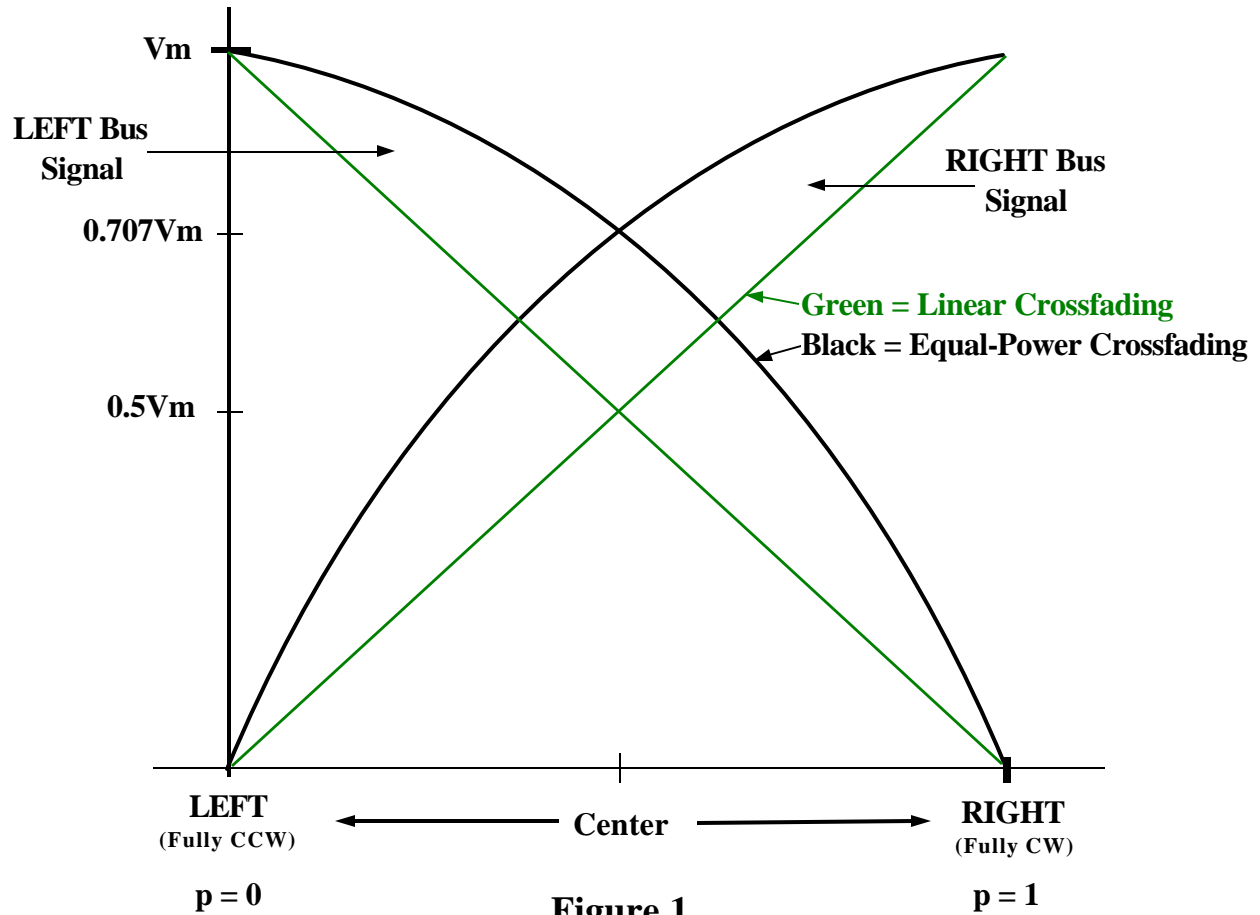


Figure 1
Linear and Equal-Power Crossfading

Now, if the maximum value of V_0 and V_1 is denoted as V_m , linear crossfading can be represented by equations (1) and (2).

$$(1) \quad V_0 = (1 - p) \cdot V_m$$

$$(2) \quad V_1 = p \cdot V_m$$

In equations (1) and (2), p is a panoramic control variable which increases from 0 to 1 as we rotate from full CCW to full CW. Thus initially, when $p = 0$, $V_0 = V_m$ and $V_1 = 0$. Then, when $p = 1$, $V_0 = 0$ and $V_1 = V_m$. However, notice that if we add equations (1) and (2) together we get equation (3) which says that the sum of V_0 and V_1 is always exactly V_m for any value of p between 0 and 1.

$$(3) \quad V_0 + V_1 = (1 - p) \cdot V_m + p \cdot V_m = V_m$$

Now, this might seem like exactly what we would want in order to keep the volume constant across the panorama so, why do we hear a dip in the middle? Well, since I promised to keep this simple, I won't get into a lengthy discussion of coherent and incoherent signals, mixing in air, and the physics and psycho-acoustics of hearing. Instead, I'll give the simple answer that when two separate signals (of the type we usually encounter in music) are added together, the apparent loudness is **not related to the sum of the amplitudes of the signals** but rather to the sum of the **power** in each signal. Since power is proportional to the square of the signal voltage, instead of us keeping $V_0 + V_1$ constant, (as we do with linear crossfading), we actually need to keep $V_0^2 + V_1^2$ constant in order to maintain equal perceived volume as we swing across the panorama.

So now the question becomes, what function of **p** should **V₀** and **V₁** be so that **V₀² + V₁²** remains fixed as **p** varies from 0 to 1? Oddly enough, the answer lies in Trigonometry. Among the zillions of trig identities, there is one that says the **sine squared plus the cosine squared (for any angle) always equals one**, see equation (4).

$$(4) \quad \text{Sin}^2(q) + \text{Cos}^2(q) = 1$$

To utilize this trig identity for our purposes we have to arrange it so the angle **q** will vary from 0 to 90° as our panning variable **p** varies from 0 to 1. The Sine varies from 0 to 1 as **q** varies from 0 to 90° and the Cosine varies from 1 to 0 as **q** varies from 0 to 90°. So, if we map **p** to **q** as shown in equation (5), we can write the needed expressions for **V₀** and **V₁** in terms of our panning variable **p** as shown in equations (6) and (7). A plot of equations (6) and (7) is shown by the two black curves in Figure 1.

$$(5) \quad q = 90 \cdot p$$

$$(6) \quad V_0 = V_m \cdot \text{Cos}(90 \cdot p)$$

$$(7) \quad V_1 = V_m \cdot \text{Sin}(90 \cdot p)$$

Note that the equal-power curves cross over each other at a height of **0.707·V_m** instead of **0.5·V_m** as for the **GREEN** linear crossfade lines. For the **EPXF** curves at any value of **p**, we can obtain the value for **V₀² + V₁²** by squaring equations (6) and (7) and adding them together as shown in equation (8).

$$(8) \quad V_0^2 + V_1^2 = V_m^2 \cdot \text{Cos}^2(90 \cdot p) + V_m^2 \cdot \text{Sin}^2(90 \cdot p) = V_m^2 [\text{Sin}^2(90 \cdot p) + \text{Cos}^2(90 \cdot p)] = V_m^2$$

When we get to the hands-on demos, you'll be able to hear for yourself how uniform the volume is across the panning field when using **EPXF** contouring.

So, what this amounts to is that we have to always arrange it so that our crossfade control variable (whatever it might be in a given application) is mapped in such a way that it will cover the range from 0 to 90° as we sweep over our desired crossfade interval. For example, if we are using the Mod Wheel to crossfade two samples, we have to map the 0 to 127 range of the MIDI controller to the range of 0 to 90° using a function such as **CV*90/127** where **CV** is the current value of the CC.

And, now is as good a time as any to mention that trig tables and algorithms can be constructed to work with any kind of angular unit you want. There are several standard systems in common use but you could also arbitrarily say that a right angle contains 128 units and name them something like **brads** (for binary radians). It really doesn't matter what the angular unit is because you can always map the crossfade control variable as needed. I'll say a little more about this later when we discuss using the **KSP Math Library** trig functions (which happen to use 1000 divisions per right angle). There is also a detailed discussion of this in the **Technical Guide**.

2.2 Using the change_vol Function

Now that I've hopefully convinced you that we're going to need some means of obtaining sines and cosines if we want to do **EPXF**, let's talk about the KSP's **change_vol** function that we will use to do all our fancy volume contouring magic with. Fortunately or unfortunately (depending on your viewpoint) the **change_vol** function requires that the control input be in **mdb** rather than just some percentage value. Actually, since the ear hears logarithmically, the db ratio system has been in broad use in the audio industry since almost day one. And, apart from the occasional bothersome chore of having to look up or calculate a logarithm or two, the system is actually quite convenient in that it allows us to cascade amplification or attenuation modules and just arithmetically add their individual gains in db to get the overall gain in db. But, whether we like the db system or not, the **change_vol** function requires us to use it.

When dealing with voltages (or the signal amplitude counterparts in K2), the formula relating our linear volume ratios to the equivalent **mdb of gain** is as given in equation (9) where V_s stands for ‘signal voltage’ and would be either V_0 or V_1 when using our 0,1 subscript notation.

$$(9) \quad G = 20,000 \cdot \log(V_s/V_m)$$

2.3 Wrapping It Up

Now let’s try to put all this together. Suppose we trigger two notes that we want to crossfade using the ModWheel. With the ModWheel at min, we want **Note[0]** to play at its full volume and **Note[1]** to play fully muted (silent). Then, as we move the ModWheel from min to max we want to fade out **Note[0]** and fade in **Note[1]** such that when the ModWheel is at max, we want **Note[1]** to be at full volume and **Note[0]** to be muted. Further, we want to do this using equal-power crossfading. Let’s arrange to put the numerical value of the ModWheel setting in a variable named **MWV** and everytime **MWV** changes, we’ll update a crossfade control variable that we’ll name **XFV**. **XFV** of course will be some function of **MWV** and that function will depend on the angular unit used by the trig functions. For the **KSP Math Library**, the angular unit used is the deci-grad (abbreviated dg). The relationship between dg and the more familiar degree system is: **1000 dg = 90°**. Therefore, when **MWV** changes, we want to recalculate **XFV = MWV*1000/127**. Doing this results in **XFV** varying from **0 to 1000** as the ModWheel moves from **min to max** (0 to 127).

Now if we compute the **Cosine** of **XFV** we will get a value between **0.0 and 1.0**. When **XFV = 0**, the **Cosine** will be **1.0** and as **XFV** increases toward **1000**, the Cosine will drop to **0.0**. Similarly, the Sine of **XFV** is **0.0** when **XFV = 0** and increases to **1.0** as **XFV** increases to **1000**. Now what does this mean insofar as what we need to tell the **change_vol()** function to do? For instance, when the ModWheel is about half way up, **MWV = 64**, **XFV** will be about 500 and both the Sine and Cosine will be **0.7071** (this is the crossover point for the functions). Now, how do we tell the **change_vol** function that we want each note to play at 70.7% of their full volume?

The answer is, we use equation (9) and plug in the value of **0.70711** for the ratio of V_s/V_m . When we do that we get **G = -3010 mdb**. And in this special case (at the half-way point) both notes should be set to this same gain value (which is actually an attenuation since **G < 0**). Thus both **Note[0]** and **Note[1]** will be reduced by about 3 db from their max volume (instead of the 6 db they would be reduced with linear crossfading).

When using the **change_vol** function, we need to mention the 3rd parameter which specifies whether the gain change is to be made relative to the initial volume of the note, **mode (0)**, or relative to the last volume setting, **mode (1)**. We would prefer to use **mode 0** since we’re computing our gain relative to 0db (without regard to history). However, if the note we’re controlling has a velocity component in its initial volume, using 0 for the 3rd parameter has a problem. **You’ll find this bug discussed in the Appendix of this User’s Guide** but, because of it, we have to use the last volume relative **mode (1)**, and, this forces us to keep track of the accumulated level relative to 0db so we will know how to get back to 0db before we change to our new desired gain. If you are controlling notes in a group that have no velocity modulation, you can use zero for the 3rd parameter and avoid this complication. However, for the demo script, I’ve assumed the more general case of having to deal with a possible velocity modulation and so I’ve illustrated how to use the **change_vol** function in its relative **mode (1)**.

Now before we wrap up this theory section and move into the laboratory for some hands-on fun, let me say this. If you found the foregoing to be a bit daunting, you’ll be happy to know that you can incorporate **EPXF** into your scripts without having to worry much about most of the foregoing. The **KSP Math Library** can do most of the hard work for you. We’re also going to walk through some solid examples of using these library functions in a real working script that you can use as a guide for incorporating the **EPXF** function into your scripts. So now that you’ve paid your dues, here comes the fun part ;-).

2.4 The 1st Demo Instrument

The **XFadeDemo#1.nki** file included with the download package is a simple monolith instrument complete with samples and two demo scripts. It only contains two very short samples that are looped to provide a sustained tone. One is that of a solo trombone playing F3 and the other is a solo clarinet (also playing F3). I made this instrument complete with the samples so that everyone that uses this tutorial will be able to hear the same sounds.

The samples are laid out with the trombone on 3 keys and the clarinet on one key. Regardless of the key assignments, the instruments only play their root pitch of F3. The key C3 is the trombone panned hard left while C4 is the trombone panned hard right. These two notes are used for the panning demo. The trombone is also assigned to E3 where it is panned to the center. The clarinet is assigned to G3 and also panned to the center. These two center-panned samples are used for the ModWheel, Velocity crossfade demo.

The main demo script is installed in slot 2 (named 'XFade Demo #1' on the tab) and there is a small bonus script (to exercise K2's panning function) installed in slot 3. This latter script should be initially bypassed. Most of the hands-on stuff will just use the main script in slot 2 so leave slot 3 bypassed unless specifically mentioned otherwise. This instrument and the demos are intended to be run with **K2 in the Standalone mode**. You should also hookup your MIDI keyboard because the demos will require you to use your Mod Wheel and Pitch Wheel.

2.5 Setting Up Your Speakers

To get the most from the panning demo, it is important that your monitors are well balanced. The best way to check and set these is as follows. The main script in slot 2 normally blocks all keyboard notes so banging on the keyboard won't do anything. However, you can temporarily Bypass the script and then hit C3 and C4 on your keyboard simultaneously and adjust the overall listening level to be moderately loud but not uncomfortable. It should never get any louder than this during the demos unless you raise K2's volume slider (which should be set at about 0 db). Now, alternately hit C3 and C4 one at a time and you will hear the left-panned and right-panned trombone tone. The idea is to set the volume of your left and right speaker until C3 and C4 sound equally loud (when played individually). The effect should be that the sound comes from the left (for C3) or the right (for C4) but at the same volume level. In other words, just the position should change, not the volume. Needless to say that you should pick some comfortable place to sit between your monitors (as if you were going to do a mixdown). Once you have completed this balancing act, disengage the Bypass button for slot 2, re-enabling the script.

2.6 The Panning Demo

If you haven't already done so, please balance your monitor speakers as described above. To run this demo, click the 'Panning Demo' button in the upper left corner of the panel. Raise or lower your ModWheel for a comfortable listening level. You should be hearing the sustained trombone sound coming from the center of the stereo field.

In this demo, the Mod Wheel acts as an overall master fader with the **Range** knob limiting the maximum attenuation. Typically, setting Range to about 75% will give you a nice smooth working range for volume control but if the lowest setting of the Mod Wheel is not sufficiently quiet, you can raise the Range knob toward 100% (or lower it if you want to bring up the bottom end). If you do this while the Mod Wheel is at min, you will hear the sound get quieter as you increase the range and vice versa. In this demo, the Mod Wheel is the primary control of the **ATFfade** routine. See **Section 3.1** for a detailed description of the **ATFfade** library function and how the Mod Wheel and Range controls interact.

You will also notice 4 display 'windows' on the panel that display pertinent crossfade data. For now, just watch the 4th window (with the caption 'Master Fader') as you move the Mod Wheel and/or Range knob. The number displayed there is the overall gain attenuation that is added to both the left and right channel notes (on top of any attenuation given to those notes by the crossfade machinery).

Now, set the ModWheel for a normal listening level and then move the Pitch Bender all the way down and you should hear the trombone move to the left of your stereo field. Next, move the pitch wheel all the way up and you should hear the trombone move way to the right of your stereo field. Now, hold the Wheel at min and slowly raise it up through the center and onward to the top. You should hear the trombone move slowly from the left side to the right side of your stereo field. If you have setup your speakers properly, only the position of the trombone should change as it moves across the stereo panorama, the volume should remain constant. Try this same slow sweep (in both directions) at various different volume settings (Mod Wheel positions). This is true equal-power crossfading in action. As you are moving the Pitch Wheel, note the first 3 display windows. The first window shows the crossfade control variable which should swing from 0 to 1000 as your Pitch Wheel moves from min to max. When this demo is started, the script triggers both the C3 and C4 notes. The crossfade machinery then sets the individual attenuation amounts for these notes. So, as you pan with the Pitch Wheel, note the 2nd and 3rd windows and watch the attenuations in **mdb**. When you are done playing with this demo, click the 'Panning Demo' button again to stop the demo.

Now before going on to the 2nd demo, you may want to listen to NI's panning function. If so, select the script in slot 3 and disengage the bypass button. Then click the button labeled 'KSP Panning Test'. You can now use the Pitch Wheel to pan from left to right just as you did with the **EPXF** panning demo (but it will always be at max volume, ie this script has no master volume control). When you slowly sweep from left to right, it should sound like it gets noticeably louder in the center. A +3db 'bump' in the center is a natural consequence of NI's implementation of the panning function. When you're done playing with this test, click the 'KSP Panning Test' button again to stop the test and then turn on the Bypass button and return to the script in slot 2.

2.7 The Vel Xfade Demo

For this demo, the Mod Wheel controls the crossfade and optionally the volume. This demo allows you to crossfade or 'morph' the clarinet timbre with the trombone timbre. To start the demo, first set the **Range knob to zero** and then set the Mod Wheel to min. Now, click on the button labeled 'Vel XFade Demo'. You should hear a sustained clarinet tone. Now, move the Mod Wheel slowly from min to max and the clarinet should become a trombone. Of course this isn't usually how you would use this function. Normally, the clarinet sound would be a lower volume sample of the same trombone. In other words this demo is to illustrate how you could use **EPXF** to do velocity crossfades with the Mod Wheel (or any other MIDI CC). I used a different timbre (the clarinet) so your ears could more easily distinguish how the two sounds are crossfading.

Now normally if you control velocity layers with the Mod Wheel, you may also want to have the volume increase as you raise the Mod Wheel. So for this demo, the Mod Wheel has a dual role. It is firstly assigned as the Crossfade Control Variable, but secondly, it is also assigned to the **ATFade** function. So, if you set the Mod Wheel to minimum and then slowly raise the **Range** knob, the clarinet will start to get quieter. You would ordinarily set this level to be that which you desire for the minimum volume of the velocity-layered instrument. Once you have set the Range knob for the min volume desired, if you then raise the Mod Wheel, the clarinet (the lower velocity layer) will crossfade into the trombone (the upper velocity layer) but at the same time, the volume will increase.

For this demo there is no actual Master Fader but you could easily add one because the **ATFade** function can be 'called' multiple times by various controllers and functions. I'll say more about this as we are 'walking through' the code of the main script. When we control the pair of notes from more than one source like this, the db system really works to our advantage. In the first demo, the Pitch Wheel controls the crossfading which in turn generates an attenuation amount for each note. In addition, the Master Fader generates an overall attenuation. With the db system, we need merely add the channel attenuation to the Master Fader attenuation before sending it to the **change_vol** function. If we were using a linear control system we would have to multiply each of these attenuation control values and, with only integer arithmetic this would likely require some messy interim re-scaling effort.

2.8 Strolling Through the Source Code

Included in the tutorial package is a text file named **XFadeDemo#1_KS.txt**. You should load this file into the **KScript Editor** so you can view it comfortably with syntax highlighting and line numbers. Please do not try to understand the source code by viewing it in the KSP editor — Big Headache! **Note:** if you want to compile the source script, you need to use **V1.25.6** or higher of the **KS Editor** and you also need to put the file named **KSPMathV2xx_KSM.txt** in the same folder since **XFadeDemo#1** imports the **KSP Math Library**.

2.9 The NewDemo and NewVolume Routines

First we'll examine the two routines that start the note pair and handle the crossfading and volume control. This is the area we need to concentrate on in order to clearly see how the **KSP Math Library** is used. Later we'll surf the rest of the code so we can get the general idea of how these routines can be invoked. Refer now to the routine named **NewDemo** (line 162 of the source code). This routine is invoked when we first click on either the **Panning Demo** or **Vel XFade Demo** buttons. If the **Panning Demo** button is pressed, **XNote[0]** and **XNote[1]** is set to MIDI note numbers named **LeftNote** and **RightNote** (which trigger the left-panned and right-panned trombone notes). On the other hand, if **NewDemo** is invoked because of pressing the **Vel XFade Demo** button, then **XNote[0]** and **XNote[1]** are set to the MIDI note numbers named **ClarNote** and **BoneNote** (for the center-panned Clarinet and Trombone respectively). Once the **XNote** parameters are set appropriately, the pair of **play_note** functions are called to trigger the specified pair of notes and the note IDs are saved in the **xid** array. As a last step, both elements of the **Odb** array are reset to zero. This array will contain the gain/attenuation value that must be added to the 'current gain' of each note to get back to 0db. And, since we have just started the notes at full volume, they are currently sitting at a level of 0db and therefore need zero change to 'get back' to 0db.

Now take a look at the next routine named **NewVolume** (line 176). This routine is invoked whenever anything happens that requires the gain of either, or both notes to be changed. This could be caused by a change in the panning control, the Master Volume control, or the **Range** knob. Or, it could be a result of having just started a demo. Notice from our discussion in the last paragraph that when the pair of notes are triggered, they are both triggered at full volume regardless of the settings of the Mod Wheel, Pitch Wheel, or **Range** knob. So obviously, **NewDemo** must be followed immediately by **NewVolume** so that the desired attenuation for each note of the pair can be established **before** the notes sound. Now let's walk through the **NewVolume** code.

The first thing we do is set the crossfade control variable named **XFV** (which we want to vary over the full, right-angle range as we make the crossfade). For the current **KSP Math Library** (as mentioned previously) the angular unit is the deci-grad, so we want to map our crossfade control so **XFV** will vary from 0 to 1000. However, since it is possible to change the angular unit for the library, the most convenient way to map a controller to **XFV** is to use the library-supplied constant named **Ang90**. For the current trig function implementation, **Ang90** is defined as 1000, meaning there are 1000 angular units in a right angle. However, if the library is altered to use a different angular unit, the value of **Ang90** will also change accordingly. So, if you use **Ang90**, instead of the literal value of 1000, your script will still work properly even if later the library is changed to use a different angular unit. Note: **Ang90** is available for your use only if your script references one or more of the library's trig functions.

Continuing now with the **NewVolume** routine, if we are running the Panning Demo, we want **XFV** to vary from **0** to **Ang90** as the Pitch Wheel varies from -8192 to +8191. And since we keep the Pitch Wheel value in the variable named **PWV**, we use the mapping function $XFV = Ang90 * (PMV + 8192) / 16383$. On the other hand, if we're running the **Vel XFade Demo**, we need to map the Mod Wheel (whose value we have in the variable named **MWV**) to **XFV**, which we do using the function $XFV = Ang90 * MWV / 127$.

Next, we want to set the relative attenuations for **xid[0]** and **xid[1]** according to the value of **XFV** and the attenuation determined by the **ATFade** function (which in turn is based on the **Range** knob setting and the Mod Wheel value **MWV**). Let's walk through this line by line starting at line #187.

First we call the library routine **ATFade**, giving it the two input parameters **MWV** and **Range**, and the routine returns the appropriate attenuation in the local variable named **MVol** (for master volume). This value as supplied by **ATFade** is already in mdb (just as we need it). Refer to **Section 3.0** for a more detailed description of this library function and how the Mod Wheel and Range controls interact. Next, we call the library routine **SinCos** with an input of **XFV** deci-grads. We take the two output values from **SinCos** and put them in the local variables named **Vol[0]** and **Vol[1]**. We put the Sine in **Vol[1]** and the Cosine in **Vol[0]** because note 0 is the one we want to fade out while note 1 is the one we want to fade in as **XFV** swings from 0 to 1000. Now the non-scaled sine and cosine have values between 0.0 and 1.0 but because we only have integers in the KSP, the routine **SinCos** generates the sine and cosine scaled by 10,000 and presents them as integers. Thus, if we wanted to know the ‘real’ sine and cosine we would need to divide the outputs of the **SinCos** routine by 10,000. Note that at this point, the values in **Vol[0]** and **Vol[1]** are $\text{Vol}[0] = 10000 \cdot V_0 / V_m$ and $\text{Vol}[1] = 10000 \cdot V_1 / V_m$. So, the next thing we have to do is convert these ratios into **mdb** for the **change_vol** function.

Now, since we are going to have to convert **two** ratios to mdb and then set the required volume change for the **two** playing notes, this might suggest some economy if we used a small loop. The double calls to **change_vol** don’t take up much space because these are real functions that NI provides. However, the **Get_db** library function has to be expanded inline and therefore it would be desirable not to ‘call’ it any more often than we need to because each linear call will expand the code inline again. This is the main reason I used a loop for the next operations. This is also one reason why it’s convenient to keep all these crossfade parameter pairs in arrays. So, let’s walk through the first pass of the loop now with **n = 0** (starting at line 190). First we call the library function named **Get_db** with an input of **Vol[0]** and place the output in the local array element **Atn[0]**. Let’s see if we can figure out what this does. If you read the header comments for the library function **Get_db**, you’ll see that it requires an input ratio scaled by 10,000 and it then generates the equivalent value (for the ratio) in **mdb**. So if we give the routine **Vol[0]** which as stated previously contains $10000 \cdot V_0 / V_m$, then **Get_db** will give us $20000 \cdot \log(V_0 / V_m)$ as an output (in mdb) which we place in **Atn[0]**. So we now have converted our ratio of V_0 / V_m into the equivalent **mdb**. Line 191 now adds **Atn[0]** to **MVol** and puts the result in **Vol[0]**. Thus, **Vol[0]** now contains both the crossfade attenuation plus the Master Volume attenuation. Remember that in the db system, you just add attenuations (or gains) to multiply their effects. Now, since we have the total amount we want to reduce the note 0 volume by (relative to 0db) we can now call **change_vol**. We do this on line 192 where we give **change_vol** the ID for note 0, **xid[0]**, and then give it the attenuation needed relative to its last known gain in **0db[0]**. The way to look at this is that we need first to raise the gain by the amount in **0db[0]** (which gets it back to 0db) and then we need to set the gain to our new desired value in **Vol[0]** (which incidentally is negative because it’s really an attenuation rather than a gain). The last thing we do in the loop body (line 193) is to update the variable **0db[0]** to the new gain value that will restore the volume to 0db. That value of course is the negative of the attenuation we just used, namely **-Vol[0]** (which makes **0db[0]** a positive value or an actual gain amount).

The loop then swings back to line 190 with **n = 1**. So the whole process now repeats, only this time the arrays will be indexed with 1 instead of 0. So, we call **Get_db** with an input of $\text{Vol}[1] = 10000 \cdot V_1 / V_m$ and get back **Atn[1]** in mdb. We then compute $\text{Vol}[1] = \text{Atn}[1] + \text{MVol}$ as the combined crossfade and master volume amounts. So now we call **change_vol** with **xid[1]** asking for a new attenuation of **Vol[1]** for this note. To do this we first raise the gain back to 0db using the gain value in **0db[1]** and then change the gain to the attenuation amount in **Vol[1]**. Finally, we update **0db[1]** to **-Vol[1]**. The only thing remaining in the **NewVolume** routine is a call to the display update routine named **ShowData**. Of course you normally wouldn’t display this data in your scripts, except maybe for test purposes. It may also be worth mentioning that when your script uses any of the logarithmic functions, including **Get_db**, the library makes available a constant named **Muted**. The value of **Muted** is currently defined as **-200,000 mdb**. You can use this constant in your script as is done in the **ShowAtn** routine.

2.10 About the Library Functions

Before we run through the rest of the code, let me point out some things about the **KSP Math Library**. First off, read the header comments. You don't have to study the routines themselves and how they are coded, but, you should become familiar with how you interface to them if you expect to get any benefit at all from them. Print out a copy of the source code file and look at it. The opening comments include an alphabetical list of the Functions the library contains, along with a brief summary of what each function does. But before you actually try to use a function, find the function itself (they're coded in alphabetical order within their classifications) and read its header comments. Preceding each function is a header with a detailed description of what the function does and what the parameters in and out are all about.

Note for example that the **SinCos** routine has the same output scaling as the input scaling required by **Get_db** so the two routines dovetail nicely and don't require any parameter rescaling in between. I also want to point out that **in V200 of the Math Library, the names of two of the Functions have changed**. The full angle versions of the trig functions have been grouped with the extended log functions and their names have been changed from **FSinCos** and **FTangent** to **XSinCos** and **XTangent** for consistency. Also, in V200, the input argument range for the fast log functions **Ln**, **Log2**, and **Log10** has been increased to a max of **65,536** (instead of **16,384** as in the prior version of the Math Library). In addition, V200 has 10 new functions, including some that will be very useful in working with Engine Parameter volume control. There are 3 extended range log functions **XLn**, **XLog2**, and **XLog10** that provide more accuracy than their DeVos'-approximation counterparts and in addition allow an input argument covering the entire positive number range from 1 to 2,147,483,647. The 3 corresponding exponential (antilog) functions are also provided. Finally, there are now a total of 5 volume control format conversion routines (including the former **Get_db**). This set of routines collectively will allow you to easily convert 'every which way' between things like Volume Ratio, Gain in mdb, and Engine Parameter control of the volume function.

2.11 Pseudo Calling

Now, before we go surfing through the rest of the Demo Script code to see how the **NewDemo** and **NewVolume** routines are invoked, let's talk a little about one of the KSP **catch-22** problems. The KSP doesn't support true user functions and on the other hand it doesn't work too well with very large scripts either! This is kind of a 'double whammy' situation. And, while the **KS Editor** allows us to write nicely modular code at the **source level**, when the code is compiled and actually loaded into the KSP, all those compact invocations of user functions have to be expanded inline. So if you have to 'call' a user function multiple times, the K2-ready, ie **KR** code can get very large (even though the **KS** source code is nice and compact). Because many of us are writing fairly ambitious scripts, we have to be very inventive and try to work around this 'catch-22' situation. So, often times you will see certain tricks being used to simulate calling a function or procedure. What we want to be able to do is to invoke a routine multiple times but only expand the code body once (just like a real user function). One such trick, which works when a routine is used multiple times in one location, is to use a loop similar to the one we examined in the **NewVolume** routine. There are several variants of this idea in use. Sometimes the loop has to be structured similar to an FSM (Finite State Machine) so that a more complex variation in the parameters can be handled but still with only one inline expansion of lengthy routines. Other times the loop will involve a **wait()** function call or two to 'handshake' with some other code elsewhere (perhaps in a different callback).

As we start to run through the remainder of the demo script, we'll see another common trick that can be used to avoid multiple inline expansion of routines that must be **called from multiple places**. This technique might be called '**pseudo-calling**' or perhaps '**remote triggering**'.

There are also many variations on this theme in use but the general idea is still the same. You basically ‘cock the gun’ one place in your code and ‘fire the gun’ some other place. The ‘gun-firing’ location (generally in the RCB) usually contains a single, inline expansion of each routine whereas the ‘gun-triggering’ machinery is scattered around in the code (but hopefully uses short sequences to do the triggering).

For an example of the **pseudo-calling** technique being used in the demo script, let’s look at the **on controller** callback beginning at line 106. The demo script is looking for activity from either the Mod Wheel or Pitch Bender. If either has changed, and thus produces a controller callback, lines 108 and 109 updates the global variables named **MWV** and **PMV**. Then, before dismissing the callback, we see the user function reference **Call(UpdateVolume)**. Obviously when either **MWV** or **PMV** changes, we will need to change our note-pair volume settings by executing **NewVolume**. This ‘call’ to **Update Volume** will accomplish just that. So, let’s take a look at how this is done.

We begin by examining two support functions named **Call** (line 140) and **XEQ_Call** (line 148). **Call** is our ‘cock-the-gun’ or trigger function so let’s walk through it. Line 141 tests a global flag named **CallActv**. The demo script uses a simple, ‘one-at-a-time’ strategy because all calls are basically for the purpose of ‘forcing’ an update of volume. In a more general situation where calls may not be so conveniently related, we would need a more complex gate-keeping function. But, for this demo script, we simply don’t do anything if **CallActv** is already set. However, assuming that no other call operation is ‘doing its thing’ or pending, **CallActv** will be off and the **if** body will execute. When it does, line 142 sets the **CallActv** flag so no other calls can be made until this one completes. Then line 143 sets the global variable named **Actions** to the bit-mapped arguments passed to the **Call** function. For the demo script, there are only two ‘callable’ commands named **StartDemo** and **UpdateVolume**.

StartDemo and **UpdateVolume** are predefined constants whose values need to be distinct integer powers of two. Specifically, **StartDemo** = 1 and **UpdateVolume** = 2. If we had additional commands they would be encoded as 4, 8, 16, etc. The power-of-two thing is used so that these ‘command’ constants can be added together to form one **Action** parameter set that can easily be decoded at the ‘gun-firing’ point as we’ll soon see.

Finally, line 144 calls on the KSP **play_note** function to generate a short MIDI note 0. In the case of the demo script, we know this note # is not being used but, it’s probably a pretty safe bet in general that this subsonic note will not be in use. The reason for this ‘fake note’ is to ‘force’ a release callback, **RCB**, to occur. The ‘gun-firing’ routine named **XEQ_Call** is placed in the **RCB** and is waiting for such triggers to occur. When we invoke the **play_note** function, we also store this note’s event id in the global variable named **Call_id**.

Now, take a look at **XEQ_Call** on line 148. On line 149 we check to see if the cause of this **RCB** is our ‘fake’ call note. If not, we bypass this routine on the assumption that this **RCB** results from some ordinary note ending. But, if the **RCB** is invoked with an event id of **Call_id**, then we execute the body of the routine. On line 150 we test **Actions** to see if the bit for **StartDemo** is on. If so, we call the **NewDemo** routine. Next, on line 153, we test **Actions** to see if the bit for **UpdateVolume** is on and if so, we call the **NewVolume** routine. Finally, we turn off the **CallActv** flag so that other calls can now be accepted.

Note that lines 151 and 154 are the only places in the script where **NewDemo** and **NewVolume** are actually inline expanded. Yet, as we’ll see, these routines are ‘called’ from several places in the script. Again I want to emphasize however, that it’s not always quite as simple as this. When we have more action routines that are only loosely related, we can’t always get by with this simple one-at-a-time idea. However, the basic scheme is still workable, it just needs to be embellished somewhat. For example, instead of just using a global **id** variable like **Call_id**, when we invoke the **play_note** function, we can store an identifying code in one of the four **event-parameter** slots (newly added in K2.1). Then when the **RCB** strikes, we test the event for this special identifying code to help sort things out.

We can also pass the **Action** information in one or more of the 3 remaining **event-parameter** slots. If we do it this way, we don't need a **gate-keeper** at the call points because we can just allow multiple 'concurrent' calls. This is possible because, regardless of the arrival order of the note 0 events at the **RCB**, each event is uniquely tagged and can thus be dealt with correctly. There are a lot of possibilities for variations here but I think this should give you the general idea. Since this **pseudo-call** scheme doesn't actually make a 'return' to the call point, the name is somewhat of a misnomer. So maybe **Task-triggering** would be a more accurate name.

My new **Interscript Operating System** for K3, uses a very advanced form of pseudo calling that actually does mimic the behavior of the traditional subroutine call/return process in that the 'called' function actually returns to the called location when it completes. The **IOS** supports both local **Subroutine** and **Task** pools and subroutine calls can be nested to any desired depth. In addition, the **IOS** supports a **Common Subroutine Library** that can reside in any one of the 5 script slots but whose routines can be 'called' from any script slot without replication of the code. Because of this powerful code-saving feature, the **KSP Math Library** has been updated for K3 and will be made a standard part of the **Common Library**. Thus, with the **IOS** installed, you will be able to invoke any of the Math Library's functions from any script slot without any further inline expansion of any of the called routines.

One other point worth mentioning here is that the **pseudo-call** idea has value beyond keeping the compiled code size smaller. There are times when you will want to invoke a function like a volume update from a **ui_control** callback. And, even if you don't care about the code size, you may not be able to code the update routine inline because certain KSP functions cannot be executed from a **ui_control** callback. For example, the **change_vol** and **change_pan** functions. So, in case you were thinking that I introduced an unnecessary complication into these demo scripts by using pseudo calling, I'll point out a few examples of where it solved just such a problem; as we now 'waltz through the coding' of the callbacks.

2.12 Callback Coding

The **on note** callback simply uses **ignore_event** so that banging on the keyboard won't do anything (since it was intended that only clicking one or the other of the Demo buttons would start the sound). Of course if you do 'bang on the keyboard', it will produce a barrage of **RCBs** but none of them will have the correct id associated with them so these events will be ignored. The **RCB**, or **on release** callback only contains the **XEQ_Call** routine as already discussed. The **on controller** callback was already looked at in the **pseudo-calling** discussion and, you may recall that it is one of the places where a **pseudo-call** to **UpdateVolume** is made. Next we come to the **PanDemo** callback starting at line 114. Line 115 turns off any sounding notes in case we are switching from the **Vel XFade Demo** or if we are simply ending the **PanDemo**. Next, if the **PanDemo** button is on (meaning we have just asked to start this demo), we turn off the **VelDemo** button (in case it was still on) and then we make a **pseudo-call**. This time we set **Actions** to both **StartDemo** and **UpdateVolume** so in the **RCB**, **XEQ_Call** will invoke both **NewDemo** and then **NewVolume**.

If the **PanDemo** button is off when this callback occurs, we set **PanDemo = 0**. Strictly speaking this isn't necessary but I did it because when a button is just clicked off, the labeling becomes fainter and harder to read. Overtly setting it to 0 with an assignment statement brings it back more legibly. I'm not sure why NI chose to have buttons with 3 visible states like this. Next we come to the **VelDemo** button callback starting on line 124. This coding is almost identical to that of the **PanDemo** and you should be able to follow it easily. It also ends up doing a **pseudo-call** with **Action** set to both **StartDemo** and **UpdateVolume**.

Then finally, we have the **Range** knob callback. This one merely makes a **pseudo-call** to **UpdateVolume**. Now notice that the **NewDemo** is executed (via **StartDemo**) from two locations and both are **ui_control** callbacks. Also notice that **NewVolume** is executed (via **UpdateVolume**) from a total of 4 places and 3 of them are **ui_control** callbacks. We need to update volume whenever we start a new demo and whenever a pertinent controller is changed or when the **Range** knob is changed. It would not be possible to inline **NewVolume** in the 3 **ui_control** callbacks because **NewVolume** uses the **change_vol** function which NI disallows in **ui_control** callbacks. So these are good examples of one of those alternate benefits of the **pseudo-call** technique that I alluded to earlier.

Finally, if I were writing this kind of script for general use, I would probably add some additional safeguards. For example, you might want to be sure when a controller changes that there are notes sounding (ie that the **xid** array has legitimate current event numbers in them) before calling **NewVolume**, etc. However, I thought that sort of thing would only clutter up the code and somewhat obscure the things we are trying to focus on. So, if you see some holes like that in the code, don't be too harsh on me ;-)

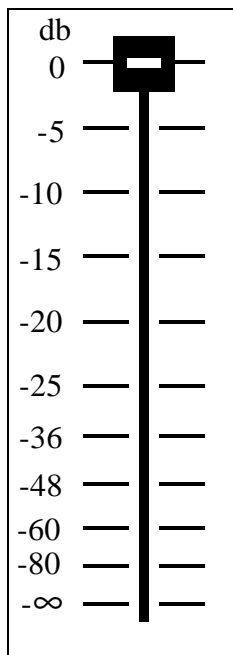
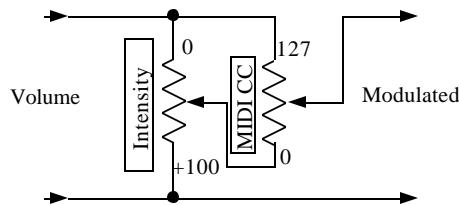
3.0 The ATFade Function

3.1 Description

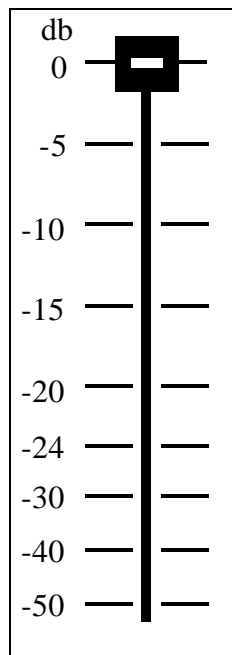
The **ATFade** function is intended to be incorporated into a host script that requires overall volume control to be handled by a MIDI CC acting as an Audio-Taper Fader. This function is called with 3 parameters as follows: **ATFade(cv,rng,atn)**. ‘**cv**’ is an input value, **0..127** from the desired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

The electronic equivalent of K2’s MIDI CC control is shown in **Figure 2**. The **ATFade** function operates in a similar way except that with **ATFade**, the contouring of the MIDI CC is more like that of a traditional ‘audio’ fader whereas in K2, the contour of the potentiometers is simply linear. So, when you assign a CC to modulate the amplifier in K2, some of the most-needed volume levels are cramped together in a small region of the CC’s travel. **ATFade** on the other hand, spreads the most-needed attenuation range of **0 to -25 db** across the top 60% of the CC’s travel (just like a real mixing-board fader). In addition, by setting a suitable value for the Range parameter (equivalent to K2’s Intensity slider), you can narrow the total range of the CC by bringing up the bottom end. This is equivalent to removing the lower end of the fader scale and then stretching the remainder back to the full size (see **Figure 3**). With such tapers, you can easily control volume expressively.

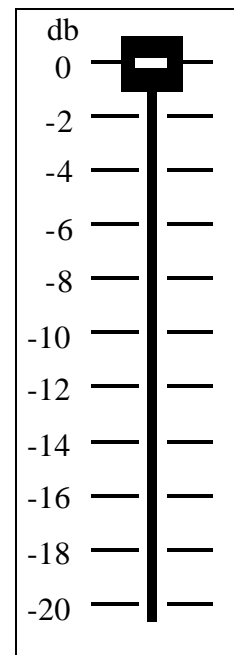
Figure 2
Electronic Equivalent For
K2’s Positive Modulation



Range = 100%



Range = 80%



Range = 50%

Figure 3
Audio Contours
at various ‘Range’ settings

4.0 Crossfading Instrument Groups

4.1 Introduction

There are often situations in which you may want to use the Engine Parameters to control instrument group volume. For example, you might want to do Mod-Wheel crossfading of velocity groups just as many libraries do by putting the samples for each velocity layer into their own individual groups. Such libraries usually assign the Mod Wheel as a volume modulator and use ReScaling curves to determine how the velocity groups are faded in and out over the Mod Wheel's 0 to 127 range. However, with script control, instead of using the ReScale graphs to contour the Group crossfades, you can easily use EPXF (true Sin/Cos shaping) by using the Math Library functions to control the Engine Parameters for Group Volume. V200 of the Math Library provides several new routines which will enable you to do this sort of thing quite easily. A second Demo Instrument and script have been added to the tutorial to illustrate how this sort of thing can be done.

4.2 The 2nd Demo Instrument

The **XFadeDemo#2.nki** file included with the download package is a simple monolith instrument complete with samples and a demo script. The instrument is constructed with 2 groups (ie Groups 0 and 1, labeled 'Lo Vel - Clarinet' and 'Hi Vel - Trombone'). The sounds are made from two very short samples that are looped to provide a sustained tone. One is a solo clarinet playing F3 which is assigned to the F3 key of Group 0 and the other is that of a solo trombone (also playing F3) and assigned to the F3 key of Group 1. Both samples in both groups are panned to the center. By crossfading Group 0 and Group 1 you can 'morph' the Clarinet into the Trombone (or vice versa). Of course normally you would be using lower and higher velocity samples of the *same* instrument but for this demo, using the two different timbres makes it easier to hear the crossfade in action.

4.3 Running the Demo

Demo Script #2 is intended to perform in a manner similar to the VelXFade mode of Demo Script #1 except that instead of crossfading a pair of notes, the Instrument Groups are crossfaded using the Engine Parameters that control Group Volume. The demo can be run by clicking the GroupXFade button and the demo can be stopped by clicking the button again.

When the demo is started, a single F3 note is generated. But, because both Group 0 and Group 1 are not 'qualified' in any way, the F3 zone of each group is activated. By setting the VolRange knob to 0%, the Master Fader control will remain at its maximum of zero db and the Mod Wheel will only control the crossfade volume of the two groups. The first box on the left displays the crossfade variable mapped to the 0 to 1000 range. The next two boxes display the Group 0 and Group 1 Volume in db. If the Master Fader VolRange knob is set to something higher than 0%, the Mod_Wheel will also control the overall volume of the samples. Except for the fact that the crossfade is done with the Engine Parameters controlling the two groups, instead of the `change_vol()` function operating on a pair of notes, the performance of this demo script is essentially identical with that of the VelXFade demo mode of Demo Script #1.

4.4 The Source Code

For this discussion, load the file named **XFadeDemo#2_KS** into the **KScript Editor** for viewing (or print it out). The primary difference between this demo script and Demo #1 is in the **NewVolume** function. Note that the master volume control **MVol** only need be applied to the single note (see line 143) and so is done outside of the 'for loop'. After computing the needed volume ratio fractions via the **SinCos** routine, **VR[0]** contains the volume ratio needed for **Group 0** while **VR[1]** contains the volume ratio needed for **Group 1** (these are of course scaled by 10,000).

At this point in the Demo #1 script we needed to convert these ratios to mdb for the `change_vol()` function to control the relative volume of each individual note of the pair. However, for this demo, we need to convert the VR ratio data into their equivalent Engine Parameter values. This is done with line 147 by using the **VR_to_ep** function. The corresponding **ep** control value (in output variable **EPV**) is then sent by line 148.

The **ShowData** function has also been changed to display the Group Volume in db by reading the display **ep**, see the routine named **ShowGroupAtn** on line 161.

Appendix

The `change_vol(x,y,0)` Bug

The `change_vol()` function using a **relative bit = 0** does not work right after a **wait()** function call. For example try the following with some **velocity-sensitive** instrument:

```
on init
  declare $New_ID
end on

on note
  ignore_event($EVENT_ID)
  $New_ID := play_note(60,30,0,-1)
  change_vol($New_ID,0,0)
  wait(2000000)
  change_vol($New_ID,0,0)
end on
```

When you hit a key, the note C3 is generated with a velocity of 30. The **change_vol** function call is supposed to change the volume by **0 db** relative to the note's initial volume level (which should be no change). The first **change_vol** function works correctly and makes no change to the initial volume. However, after the two-second delay when the 2nd **change_vol** function is called, you will hear the volume jump up. Apparently after a wait call, the note velocity's effect is forgotten because the volume level is that of a C3 with velocity = 127.

The `change_vol` function seems to work OK when using a **relative bit = 1** so if you keep track of your accumulated changes in a variable '\$A' you can work around this problem by using:

```
change_vol($New_ID,v-$A,1)
```

To simulate:

```
change_vol($New_ID,v,0)
```

Where \$A contains the accumulated volume changes since the original note and 'v' is the amount of volume change that you want relative to the original note's volume.