A Collection of BuzzDev Resources

BM* format. I wrote this for myself so it may be a bit hard to understand. :)
BMW is just like BMX except it doesn't include the 'WAVE' section.

Header:

Type/Size	Description
4	"Buzz"
dword	number of sections
12*31	up to 31 section dir entries

==

Section dir entry:

Type/Size	Description
4	four-char name of section
dword	offset from begin of file
dword	size in bytes

==

Section 'MACH' - machines

Type/Size	Description
word	number of machines
(first machine - always master)	
asciiz	name
byte	type (0 = master, 1 = generator, 2 = effect)
asciiz	name of DLL if type is 1 or 2
float	X coordinate in machines view [-1..1]
float	Y coordinate in machines view [-1..1]
dword	size of machine specific data
x	data
word	number of attributes
(first attribute)	
asciiz	key
dword	value
(second attribute)	
...	
x	state of global parameters
word	number of tracks
x	state of track parameters for each track
(second machine)	
...	

==

Section 'CONN' - machine connections

Type/Size	Description
word	number of connections
(first connection)	
word	index of source machine
word	index of destination machine
word	amp
word	pan
(second connection)	
...	

=== =======

Section 'PATT' - patterns for each machine

Type/Size	Description
(first machine)	
word	number of patterns
word	number of tracks
(first pattern)	
asciiz	name
word	length of pattern in number of ticks (rows)
x	pattern data
(second pattern)	

```
...

(second machine)
...

========================================================================

Section 'SEQU' - pattern sequences

Type/Size         Description
---------------------- --------------------------------------------------
dword             end of song
dword             begin of loop
dword             end of loop
word              number of sequences

(first sequence)
word              index of machine
dword             number of events
byte              bytes per event pos
byte              bytes per event (2 if there are more than 112 patterns)
x                          event list (pos, event, pos, event, pos, event...). events:
                                    00 = mute, 01 = break, 02 = thru
                                    0x10 = first pattern, 0x11 = second pattern, etc.
                                    msb=1 indicates loop

(second sequence)
...

=============== =========================================================

Section 'WAVT' - wavetable

Type/Size         Description
------------------------------------------------------------------------
word              number of waves

(first wave)
word              index
asciiz            full file name, e.g. "c:\waves\blah.wav"
asciiz            name
float             volume
byte              flags:
                                    bit 0: loop
                                    bit 1: don't save
                                    bit 2: floating point memory format
                                    bit 3: stereo (since 1.2)
                                    bit 4: bidirectional loop (since 1.2)
                                    bit 7: envelopes follow (since alph a 14)

if flag bit 7 --------->
         word              number of envelopes

         (first envelope)
         word              Attack time
         word              Decay time
         word              Sustain level
         word              Release time
         byte              ADSR Subdivide
         byte              ADSR Flags: 0-1 = attack mode, 2-3 = release mode, 4 = linear decay , 5 = sustain

         word              number of points (can be zero) (bit 15 set = envelope disabled)

         (first point)
         word              x
         word              y
         byte              flags: bit 0 = sustain

         (second point)
         ...

         (second envelope)
         ...
<----------- end of if flag bit 7

byte              number of levels

(first level)
dword             number of samples
dword             loop begin
dword             loop end
dword             samples per second
byte              root note

(second level)
...

(second wave)
...
```

```
========================================================================

Section 'WAVE' - wave data

Type/Size         Description
------------------------------------------------------------------------
word              number of waves

(first wave)
word              index of wave
byte              format:
                                    0 - raw 16bit, intel byteorder
dword             number of bytes in all levels

(data for first level)
..

(data for second level)
...

(second wave)
...

========================================================================

Section 'BLAH' - song info

Type/Size         Description
--------------------------------------------------- ----------------
dword             number of characters
x                 raw ascii text (no zero at end)

========================================================================

Section 'PARA' - parameter information for machines

 - added in v1.2. This section is not required  for loading the song if user
   has right versions of all machines installed. otherwise the information
   here can be used to convert pattern data to the new format.

Type/Size         Description
------------------------------------------------------------ -----------
dword             number of machines

(first machine)
asciiz            name
asciiz            type (for example "Jeskola Tracker")
dword             number of global parameters
dword             number of track parameters

(first parameter - all global first followed track -parameters)
 see CMachineParameter in MachineInterface.h for more information
 all fields except "Description" are saved

byte              type
asciiz            name
int               minvalue
int               maxvalue
int               novalue
int               flags
int               defvalue

(second parameter)
...

(second machine)
...


================== =======================================================

Section 'PDLG' - parameter dialog placements on screen
- added in v1.2

byte              flags:
                           bit 1: dialogs visible

list of positions followed by terminating zero byte:

asciiz            name of machine
WINDOWPLACEMENT   win32 window placement structure (see win32 documentation)


========================================================================

Section 'MIDI' - midi controller bindings
- added in v1.2
```

list of bindings followed by terminating zero byte:

asciiz	**name of machine**
byte	**parameter group**
byte	**parameter track**
byte	**parameter number**
byte	**midi channel**
byte	**midi controller number**

===

```
// allocated track:
// 1. same LastNote if note not in pattern
// 2. first free with no note in pattern
// 3. first free
// 4. track with oldest high note (C6+)
// 5. track with oldest mid note (C3+)
// 6. track with oldest note

int mi::AllocateTrack(CSequence *pseq, int note)
{

	for (int c = 0; c < numTracks; c++)
	{
		if (Tracks[c].LastNote == note)
		{
			if (pseq != NULL)
			{
				byte *pdata = (byte *)pCB->GetPlayingRow(pseq, 2, c);
				if (*pdata == NOTE_NO)
					return c;

			}

		}
	}

	int best = -1;


	for (c = 0; c < numTracks; c++)
	{
		if (Tracks[c].Voices[Tracks[c].FreeVoice^1].State == CVoice::inactive)
		{
			if (pseq != NULL)
			{
				byte *pdata = (byte *)pCB->GetPlayingRow(pseq, 2, c);
				if (*pdata == NOTE_NO)
					return c;

			}

			if (best == -1)
				best = c;
		}
	}

	if (best != -1)
		return best;

	// if we got here it means all voices are active
	int oldt = -1;

	// check high
	for (c = 0; c < numTracks; c++)
	{
		if (Tracks[c].LastNote >= ((6 << 4) + 1))		// >= C6
		{
			int at = Tracks[c].Voices[Tracks[c].FreeVoice^1].ActiveTime;
			if (at > oldt)
			{
				oldt = at;
				best = c;
			}
		}

	}

	if (best != -1)
		return best;

	// check mid
	for (c = 0; c < numTracks; c++)
	{
		if (Tracks[c].LastNote >= ((3 << 4) + 1))		// >= C3
		{
			int at = Tracks[c].Voices[Tracks[c].FreeVoice^1].ActiveTime;
			if (at > oldt)
			{
				oldt = at;
				best = c;
			}
		}

	}

	if (best != -1)
		return best;

	// check low
	for (c = 0; c < numTracks; c++)
	{
		int at = Tracks[c].Voices[Tracks[c].FreeVoice^1].ActiveTim e;
		if (at > oldt)
		{
			oldt = at;
			best = c;
		}
	}


	assert(best != -1);

	return best;

}

void miex::MidiControlChange(int const ctrl, int const channel, int const value)
{
	// TODO: check channel

	if (ctrl != 64)   // 64 = sustain pedal
		return;


	if (value < 64)
	{
		CSequence *pseq;

		int stateflags = pmi ->pCB->GetStateFlags();

		if (stateflags & SF_PLAYING && stateflags & SF_RECORDING)
			pseq = pmi ->pCB->GetPlayingSequence(pmi ->ThisMachine);
		else
			pseq = NULL;

		int notedelay = p mi->pMasterInfo ->PosInTick * 24 / pmi ->pMasterInfo ->SamplesPerTick;

		pmi->SustainPedal = false;
		for (int c = 0; c < pmi ->numTracks; c++)
		{
			if (pmi->Tracks[c].Note != NOTE_NO && pmi ->Tracks[c].Sustained)
			{
				pmi->Tracks[c].Sustained = false;
				pmi->MidiNoteOff(c, pseq, notedelay);
			}
		}
	}
	else
	{
		pmi->SustainPedal = true;
	}



}


void mi::MidiNoteOff(int c, CSequence *pseq, int notedelay)
{
	Tracks[c].NoteOff();

	if (pseq != NULL)
	{
		byte *pdata = (byte *)pCB ->GetPlayingRow(pseq , 2, c);

		if (notedelay > 0)
		{
			if (pdata[0] != NOTE_NO)
			{
				if (pdata[2] == 0x0d)
				{
					int ondelay = pdata[3] >> 1;
					pdata[2] = 0x0b;
					pdata[3] = (ondelay << 4) | (notedelay >> 1);
				}
				else
				{
					// ondelay = 0
					pdata[2] = 0x0b;
					pdata[3] = (notedelay >> 1);
				}
			}
			else
			{
				pdata[0] = NOTE_OFF;
				pdata[2] = 0x0d;
				pdata[3] = (byte)notedelay;
			}
```

```
                }
                else
                {
                        pdata[0] = NOTE_OFF;
                }


        }

}


void mi::MidiNote(int const channel, int c onst value, int const velocity)
{
        // TODO: check channel

        if (value / 12 > 9)
        return;

        byte n = (((value / 12) -1) << 4) | ((value % 12) + 1);

        CSequence *pseq;

        int stateflags = pCB ->GetStateFlags();

        if (stateflags & SF_PLAYING && statefl ags & SF_RECORDING)
                pseq = pCB ->GetPlayingSequence(ThisMachine);
        else
                pseq = NULL;


        int notedelay = pMasterInfo ->PosInTick * 24 / pMasterInfo ->SamplesPerTick;

        if (velocity > 0)
        {
                int c = AllocateTrack(pseq, n);

                Tracks[c].Note = n;
                Tracks[c].Velocity = velocity;
                Tracks[c].NoteOn();

                if (pseq != NULL)
                {
                        byte *pdata = (byte *)pCB ->GetPlayingRow(pseq, 2, c);
                        pdata[0] = n;
                        pdata[1] = velocity;

                        if (notedelay > 0)
                        {
                                pdata[2] = 0x0d;
                                pdata[3] = (byte)notedelay;
                        }
                }


        }
        else
        {
                for (int c = 0; c < numTracks; c++)
                {
                        if (Tracks[c].Note == n)
                        {
                                if (SustainPedal)
                                        Tracks[c].Sustained = true;
                                else
                                        MidiNoteOff(c, pseq, notedelay);

                                break;
                        }


                }
        }


}
```

Date: Fri, 12 Jun 1998 20:12:27 +0300
From: Oskari Tammelin <ot@iki.fi>
Subject: Re: wavetable

At 11:31 AM 6/11/98 -0500, you wrote:
>is it possible for a plug -in to access the samples in the wavetable?

Yep, Jeskola Tracker wouldn't work otherwise (it's just like any other machine). You can also write to the wavetable if you want to write a nonrealtime samplegenerator or something like that.

```
virtual CWaveInfo const *GetWave(int const i);
virtual CWaveLevel const *GetWaveLevel(int const i, int const level);
virtual int GetFreeWave();
virtual bool AllocateWave(int const i, int const size, char const *name);
virtual CWaveLevel const *GetNearestWaveLevel(int const i, int const note);
```

In tracker you just need to do this:

```
CWaveInfo const *pWave = pCB ->GetWave(pstate->iWave);
CWaveLevel const *pLevel = pCB ->GetNearestWaveLevel(pstate ->iWave,
pstate->Note);
```

See the attached file for example of writing to wavetable. It's the MOD importer code of the Tracker.

--
Oskari Tammelin .oOo. oskari@twilight3d.com .oOo. http://www.twil ight3d.com

----

Date: Tue, 16 Jun 1998 03:18:37 +0300
From: Oskari Tammelin <ot@iki.fi>
Subject: Re: importing .IT file samples

At 04:32 PM 6/14/98 -0500, you wrote:
>Currently it is messy - you add the machine, and the mi::Init() method pops
>open a file dialog to select the .IT file to load samples from. In the
>meantime
>a mutex lock error message box pops up.

You shouldn't show any dialogs in Init().

>What would be the correct way to implement something like this which isn't
>really
>a generator or effect?
>I was thinking the context menu of the machine should have an "Import..."
>command, just like the Tracker machine does.   How do I do that, if possible?

Yep, just implement the Command() method:

```
void mi::Command(int const i)
{
        switch(i)
        {
        case 0: ImportModule(); break;
        case 1: AnotherCommand(); break;
        }
}
```

And change last line of MacInfo to "Import Module... \nAnother command"

Note that you must lock the machines if you are changing something that the player thread is using at the same time. The easiest and safest way to do is to use the MACHINE_LOCK macro like this:

```
{
    MACHINE_LOCK   // this constructs a C++ objec t


    // do something..


} // <- unlock happens here
```

--
Oskari Tammelin .oOo. oskari@twilight3d.com .oOo. http://www.twilight3d.com

----

Date: Wed, 17 Jun 1998 07:03: 05 +0100
From: Oskari Tammelin <ot@iki.fi>
Subject: Re: simple buzz tracker question

At 04:55 PM 6/16/98 +0200, you wrote:
>im trying to code a simple buzz tracker. for now it plays samples nicely
>at their default rate. How do i calculate the playbac k rate according to
>the entered notes and resample the samples to this rate in the generator
>code ?? Are there some build -in functions to do this ?

Rate = 2.0 ^ (N / 12.0)

Where N is the note relative to the root note. For example if root note is
C-4 then N=1 would be C#4 and N= -12 would be C -3.

dsplib provides a resampler function, so you should use it (see resample.h
for help). The resampler has separate inner loops for all special cases so
it's very efficient.

--
Oskari Tammelin .oOo. oskari@ twilight3d.com .oOo. http://www.twilight3d.com

----

Date: Fri, 26 Mar 1999 03:36:33 +0200
From: Oskari Tammelin <ot@iki.fi>
Subject: Re: New machine released: JoyPlug 1

At 06:35 PM 3/25/99 -0600, you wrote:
>JoyPlug 1 has been released!
>JoyPlug 1 is a joystick -controlled lowpass resonant filter with
>distortion. Instead of preprogramming in your filter data, just
>control it with the joystick, live. The source code is inclu ded as a
>slightly more complicated example of how to write something in Buzz.
>The plugin works with Buzz 1.x and a Win95 compatable joystick (3
>axis joystick preffered)

Sounds like a cool machine. I wish I had Win95 and a joystick. :)
The documentat ion was very nice too but there were some wrong information.
Here are some corrections so that people don't get confused.

If there are functions that you don't need (like SetnumTracks and Tick in
your machine) you don't need to implement them either.

The range of the signal is -32768..32767 inclusive if you want full dynamic
range without lowering the master volume. It's not limited to that however.
Machines can use much larger values if they need to but the user must
manually lower the amps to avoid c lipping.

Those were both just minor things but this one is quite important: The
'mode' parameter in Work() tells you what you can do to the samples for
effect machines, generators can ignore it.

WM_READWRITE is the normal mode.
WM_READ means you shoul d only read but not write to the buffer. This
happens when machine is in <thru> mode (set in seq editor).
WM_WRITE is the opposite. it happens when machine has no input (in other
words when all input samples are 0). Machines that generate output even
if there's no input should support this mode. That includes reverbs, delays
and even filters.
WM_NOIO means there's no input and you shouldn't write to the buffer.

In all cases, you should run your machine normally, but not read or write
to the buffer depend ing on the mode. If your machine's output doesn't
depend on past inputs you should "turn it off" to save cpu time when mode
is WM_READ or WM_NOIO. Reverbs and delays should turn themselves off after
they haven't had any input for a while (reverb/delay time).

Otherwise the code was fine. Nice work. :)

--
Oskari Tammelin .oOo. oskari@twilight3d.com .oOo. http://www.twilight3d.com

----

Date: Fri, 26 Mar 1999 16:55:38 +0200
From: Oskari Tammelin <ot@iki.fi>
Subject: Re: New machine released: JoyPlug 1

At 09:31 AM 3/26/99 +0100, you wrote:
>Why?
>This is something I've never quite understood.
>Isn't it better to have the same load on the computer
>all the time, instead of load chan ging?
>I guess not =)

Why the hell would it better? Low CPU usage is always better. We are not talking about a game like quake where constant FPS rate is usually more important. For example if part A of your song uses a machine connected to reverb1 and part B uses machine connected to reverb2 you think it would be better to run both reverbs all the time? :)

There's another reason too. When a feedback delay line runs for too long without input it starts to generate floating point exceptions (too small numbers). This makes it run around five times slower.

[Yes, the interrupt is masked but it's still very slow. anyone know the exact number of cycles wasted here?]

The machines are turned off when their output is so low that it can't be heard so no-one can notice when it's turned off.

--
Oskari Tammelin .oOo. oskari@twilight3d.com .oOo. http://www.twilight3d.com

----

Date: Fri, 09 Apr 1999 07:52:21 +0300
From: Oskari Tammelin <ot@i ki.fi>
Subject: Re: resampling with dsplib

>At 03:37 AM 6/19/98 -0500, you wrote:
>I'm having some trouble figuring out how to use the resampler in dsplib.
>
>If you want to resample some data to a given Rate, what do you set StepInt
>and StepFrac to?
>(or what do you pass to SetStep()?)
>
>Also, what is the difference between the two step modes?

I was going thru old buzz-dev mails and found this.. Maybe I should have replied it earlier. :)
Anyway:

Step is the value to add to the sample read pointer for each sample. So Step=2 would play the sample at double speed and Step=2^(1/12) would play C#3 if the sample is sampled at C3.

RSS_ONE means the resampler assumes Step=1 (happens often with percussive samples) and doesn't do any resampling which is o f course much faster.

RSS_CONSTANT means Step can be anything but stays constant. The resampler doesn't support pitch sliding so there's no RSS_INTERPOLATE mode.
SetStep is just a helper function so you don't have to set StepInt and StepFrac manually. Yo u can use it or not.

--
Oskari Tammelin .oOo. oskari@twilight3d.com .oOo. http://www.twilight3d.com

Adding skins to your machines is quite easy. Just add these resources to your machine dll:

Resource type	name	description
Bitmap	"skin"	bitmap for the machine, size must be 100x50
Bitmap	"skinled"	"led on" patch bitmap (if not given, default led is used)
"position"	"skinledpos"	x and y coordinates inside skin bitmap for the led on bitmap (two bytes)
"color"	"textcolor"	machine name text color (3 bytes, RGB) (optional, defaults to black)

notes:

-quotes ("") are important in the above names!
-The panning bar goes from 2,38 to 97,47 inclusive. Make sure your skin design accounts for this
-font is between y=20 and y=30

Some advices:

- use 256 color bitmaps to save memory
- led can be redrawn quite often so keep it small
- if you want all your machines to use same bitmap, do not include same bitmap in each dll.
  instead create another dll and add fo llowing resource to your machine dlls:

Resource type	name	description
"asciiz"	"skinref"	name of skin dll (for example "Geonik's Skin.dll")

this dll should be installed to Buzz \Gear\Skins dir.

Some Hints for BUZZ machine writing (version 0.3)
by vII

Contents

Introduction/Purpose

Generators:
- Generator note pitches
- Alternate tunings
- MIDI playability

Effects:
- Several separate inputs

General:
- Own Dialogs
- DirectX usage
- Maintaining version compatibility
- the Save/Init functions
- Positions
- Stop->Start?
- Event mechanism
- Accessing wave data on startup

Credits
Contacting the author

Introduction/Purpose
I've just started writing some machines for BUZZ, the first and only published being vMidiOut and vGraphity. In the BUZZ mailing list there are often questions regarding the machine programming and maybe the desire to have some kind of tutor. I'm not that expert (yet?) and don't want to or can cover every detail of programming the machines, especially I'm not experienced with DSP stuff and haven't written an effect machine yet nor used the dsplib of BUZZ. But I've found some things that aren't in the supplied sources or don't come into sight easily, which I think are worth to be summed up here (some of them are pure assumptions, marked with an "?"):

Starting the thing seems quite easy, just go as Jeskola explains and use the supplied sources as a start. Visual C++ 6.0 seems to work o.k., but the resulting machine DLL's have double the size than compiled with VC 5, so I'm mostly staying with the older version for the releases (any settings to cure this?).

Generators:

Generator note pitches: You need to know the note format of BUZZ. They are stored as bytes with the higher 4 bits as octave and the lower 4 bit the note value. The notes are from 1 for C to 0x0c (12) for B (or H in german), the octaves range from 0 to 9. So the least possible note is 1 for C-0, than it goes 2 for C#0,... 9 for G#0,0x0a for A-0, 0x0b for A#0 (german B), 0x0c for B-0, jumps to 0x11 for C-1 and so on. The formula for a continuous note range from 0 to 96 is

```
        Note = (tv.note>>4)*12+(tv.note&0x0f)-1;
```

if tv.note is the name for your BUZZ track note. Special values are 0 for no note and 0xff for a note off command.
The resulting note may be shifted by octaves (multiples of 12) to get the correct MIDI note, the corresponding frequencies would be

```
        freq =  base_freq* pow(2.0, Note/12.0);
```

with a base_freq = 16.3516 Hz for C-0 (-> A-4 = 440 Hz).

Alternate tunings: A bit more complicated are alternate tunings. For vMidiOut I implemented them via pitchwheel commands (which may sound very bad and only allows monophone MIDI channels), but I've also found an archive with over 1000 scales, which could be used as standard format for any machines experimenting with this topic. The basic idea is to fill a float array of all 127 possible MIDI notes (or for BUZZ some less) with the according frequencies. The scale file contains comment lines (starting with an '!'), the scale name, the number of steps *s* for a whole scale (for the standard chromatic scale or many others this is 12) and following the intervall between the single steps and the base note of the scale either in cents (100 cents is 1 half tone step) or as ratio (an octave is 2/1). To fill the frequency array you need additionally a root frequency *f0* to start the scale with and associate a MIDI note *m0* with this frequency. Now just set all frequencies with an index less or equal to *m0* to *f0*, and calculate the next *s* frequencies from the file. Repeat the last step until you reach the end of your array. Easy, isn't it?

MIDI playability: While I found no way to add a MIDI record option to a generator, the (at least monophonic) playing of a generator via MIDI should be very easy to implement. You have just to add the mi::MidiMote(int const channel, int const value, int const velocity) to your code, where value is the midinote this time (no translation necessary) and velocity the "volume" in the range from 0 (for note off) to 0x7f (127), and do in the function just the same stuff you would do in the mi::Tick() option, when a track note arrives. The main problem is to switch this feature on and off and assign a track to listen to. For a new machine you could easily add an attribute, but for existing ones with compatibility in mind, you should add an extra dialog for this.

Effects:

Several separate inputs for an effect machine: BUZZ 2 shall bring the possibility of using several inputs for one effect without simply mixing them together first. This is needed for a real vocoder (not using the wavetable samples) or similar machines. But we don't have to wait for BUZZ 2 to achieve this and we also don't have to restrict to a limited bandwidth as in Ynzn's Multiplexer/Demultiplexer (which nonetheless is a nice idea to solve this basic task completely according to the BUZZ 1 machine interface; and it gave me the idea to avoid the interface for this one):
All machines of the same kind can share global data of the machine dll. So it would be easy to have several ringbuffers (several -> to allow more than one pair of inputs) of let's say 1k samples. Every machine now has a parameter, telling which buffer to use, and another, if it shall write to the buffer. A second machine can now use it's own input and the input from any of the ringbuffers (even from more than one) to do what it wants...

For something like a vocoder this would look like:

```
        (output is set to zero,but also written into ringbuffer N)
input 1 -------> vocoder 1  -------------|
                                         |
input 2 ------------------------------vocoder 2---------> output
        (has access to input 2 and ringbuffer N)
```

General:

Own dialogs: For communication with the machine (input) you have the attributes, the global and track parameters, the Init/Save functions and the machine menu (with the command function). As output the supplied MessageBox() just isn't enough, as aren't the menus in case you want to switch some more parameters. So it's very soon you want to add a DialogBox, but to access resources in the DLL you have to add something like

```
HINSTANCE dllInstance;
mi *p_mi;

BOOL WINAPI DllMain( HANDLE hModule,
                        DWORD fdwreason,  LPVOID lpReserved )
{
    switch(fdwreason) {
    case DLL_PROCESS_ATTACH:
    // The DLL is being mapped into process's address space
    //  Do any required initialization on a per application basis, return FALSE if failed
    dllInstance=(HINSTANCE) hModule;
    break;
    case DLL_THREAD_ATTACH:
    // A thread is created. Do any required initialization on a per thread basis
    break;
    case DLL_THREAD_DETACH:
    // Thread exits with  cleanup
    break;
    case DLL_PROCESS_DETACH:
    // The DLL unmapped from process's address space. Do necessary cleanup
    break;
    }
    return TRUE;
}
```

so you can create your dialog with

```
p_mi=this;
DialogBox(dllInstance, MAKEINTRESOURCE(IDD_DIALOG1), NULL, ConfigDialog);
```

in the Command() function and access your data in your ConfigDialog() message handler with the p_mi function.To find the BUZZ main window I used the EnumWindows() function, looking for a "- Buzz - " in the title.

Using DirectX: To use any components of DirectX (as vGraphity for instance uses DDraw) you usually need to give a window handle for initializing. If BUZZ itself already uses DirectX because of the DirectSound driver, you are forced to use the handle of the main BUZZ window (see last point), or the initialization will fail. With vGraphity for instance this causes some inconsistencies with the associated window being BUZZ but the really targeted window being that of vGraphity...

Compatible machine versions: To allow future versions of your machines to be compatible with older, there are some things to concern just from the beginning. Of course the global and track data size must not change anymore, also the default values if possible (but changing them doesn't cause BUZZ crashes). It is possible to add attributes later to a machine without, but once the machine has some of them, they can't be changed anymore (really?), so best you generally add some more for future use.
Another thing is to implement things and data sizes a bit generous, even if this increases the pattern and song size. A good idea may be a "method" attribute (as in vMidiOut) or a similar byte/integer in the patterns (as in Geonik's Omega). This allows you to redesign your machine a lot, without losing compatibility. But always keep in mind, that you also just can give your machine a new name.

Save/Init: If you begin to use the Save and Init functions, you should establish your own mechanism to deal with all data sizes (of current, former and maybe even future versions), especially check for an input pointer of zero, to allow loading of older songs without data at all. I always read and write the data size first, followed by some kind of version indication for the machine a song was written with. This way it would be even possible to say: "Hey, you should use at least version xxx of this machine, to play this song correctly."

Positions: Unfortunately the machine interface doesn't give you any access to song positions (yet?). The GetPlayPosition() and GetWritePosition() commands just start if you press the "Play" button the first time after loading a song and won't never stop again (with the pause button). Both count samples and wrap around at 0x800000. The write position always is a bit ahead, so I guess it's something like the actual position of data written to the wave driver, while the play position tells you, what you're listening to. For the DirectSound and the silent driver they seem to run smoother (or even continuous?) with a constant difference (depending on driver settings, especially latency), while with the standard wave driver they increase in larger steps and also the difference varies (why this?).

Stop->Start? If there's a Stop() function, shouldn't there be a similar start function, too? I guess yes, but there's really none implemented yet and also nothing to replace this missing one: the above position functions run through and so do the Tick() and Work() functions. But I've heard about the possibility of hooking the BUZZ play button ;)

Event mechanism: BUZZ has a built-in event mechanism (based on a 5ms timer?), which should avoid an own timer for any machine, but sometimes it seems to be left behind in favour of the wave output (especially on NT?). The standard usage would be to call ScheduleEvent(GetWritePos(),data) in the Tick() function and implement the Event(data) method. This would give you an event triggered when the play position reaches the data processed in the above Tick()-call. There seems to be a limit on the number of events waiting (at least I had some crashes with many of them), so I reduced them to one per Tick() in vMidiOut. You also must not call ScheduleEvent() in the very first Tick() (where the last active machine parameters, which always get saved with the song, are written back to the machine). I guess there's also a problem with waiting events when closing the machine (calling the destructor), or at least with accessing the machine data when processing them.

Accessing wave data on song startup: My vGraphity concept of storing machine data in the wavetable section has one disadvantage: on startup (for instance for autoshowing an image) the machines are loaded and initialized *before* the waves are loaded. The easy solution: I've used a 2sec timer to delay loading anything from the waves. This may cause difficulties with machines in the song which need a longer initialization time for themselves or if the listener is very quick with pressing the play button...

Credits:
Found out some things myself, but others not:
Thanks to Jeskola for the help, especially concerning the event mechanism (and of course for the source of all this pleasure -and sometimes trouble- BUZZ).
And to Hagen: hooking the play button is just a too fine idea to let it be unmentioned... I'm still thinking about OCR for the song position ;)

contact me or mail contributions to: (vII) or look at my Homepage

Buzz Machines Programming Tutorial

Contents

- Introduction
- Tutorial
  - Setting up the project
  - Defining your machine
    - Setting up a basic machine info structure
    - Setting up parameters
    - Setting up attributes
    - Setting up context menu options
  - Implementing your machine
    - Inheriting from CMachineInterface
    - The Init method
    - The SetNumTracks method
    - The Tick method
    - The Work method
    - The Stop method
    - The Command method
    - Using a separate CTrack class
  - General advice for working with digital signals
  - Using the dsplib library
- Reference
  - The CMasterInfo structure
  - The CMICallbacks services
    - General
      - GetWave
      - GetWaveLevel
      - MessageBox
      - Lock
      - Unlock
      - GetWritePos
      - GetPlayPos
      - GetAuxBuffer
      - ClearAuxBuffer
      - GetFreeWave
      - AllocateWave
      - ScheduleEvent
      - MidiOut
      - GetOscillatorTable
    - Envelopes
      - GetEnvSize
      - GetEnvPoint
      - GetNearestWaveLevel
    - Pattern Editing
      - SetNumberOfTracks
      - CreatePattern
      - GetPattern
      - GetPatternName
      - RenamePattern
      - DeletePattern
      - GetPatternData
      - SetPatternData
    - Sequence Editing
      - CreateSequence
      - DeleteSequence
      - GetSequenceData
      - SetSequenceData

Introduction

One of the strengths of Buzz is that it is possible for any programmer (assuming access to Visual C++) to extend it by adding new effects and generators. Unfortunately, the documentation is currently lacking in some areas. Rather than complaining, it seemed sensible to make some documentation up myself - after all, I'd rather have more updates to Buzz than have the developers waste their time on perfect documentation.

This tutorial is not complete. If you know how something is done that is not currently in this tutorial (or if you think something is wrong, or badly explained), please email me at steve@lurking.demon.co.uk.

This tutorial assumes a working knowledge of C++, including classes and virtual functions.

Tutorial

Programming a Buzz machine is not a particularly complex programming task. My largest Buzz machine so far has 718 text lines, and around 250 lines of that is straightforward but repetitive code to generate waveforms efficiently.

Most of the code is very similar for any Buzz machine, and so the first step in a project is to copy the source to one of the example Buzz machines. Using this as a template will save a lot of typing.

The basic structure of a Buzz machine does take some getting used to, but a little work should pay off.

Setting up the project

Buzz machines can only be developed (as far as I know) using Microsoft Visual C++ v5.0 onwards. I have tried using Borland C++ v5.02, but could not find options that worked.

This isn't necessarily all that bad. Although I generally prefer Borland C++, Visual C++ does produce cleaner and smaller DLLs. As a buzz machine is a DLL, this is a significant consideration.

To set up a suitable project in Visual C++ 5.0...

1. Create a new 'Win32 Dynamic-Link Library' project.
2. Select Project/Settings, and enter the following options...
   - C/C++ Tab, Category='Code Generation'
     - Processor: Pentium
     - Use run-time library: Multithreaded DLL
     - Calling convention: __fastcall
     - Struct member allignment: 4 bytes
   - Link Tab, Category='Input'
     - Object/library modules: Add 'dsplib.lib'
     - Additional library path: '..\dsplib'
3. Select Build/Set Active Configuration, and choose the release configuration
4. Select Project/Settings, and enter the following options...
   - C/C++ Tab, Category='Code Generation'
     - Processor: Pentium
     - Use run-time library: Multithreaded DLL
     - Calling convention: __fastcall
     - Struct member allignment: 4 bytes
   - C/C++ Tab, Category='Optimizations'
     - Optimizations: Maximize speed
     - Inline function expansion: Any suitable
   - Link Tab, Category='Input'
     - Object/library modules: Add 'dsplib.lib'
     - Additional library path: '..\dsplib'

Optimisations should be set up for speed rather than size, because Buzz is a real time system which makes considerable demands on many systems (My Pentium 166mmx cannot cope with some buzz tunes, so I suppose its time to upgrade again). Even those with high end Pentium II systems can probably achieve more if the code is well optimised. Besides which, Buzz machines tend to be so small that optimising for size is pointless.

Setting up a basic machine info structure

The main definition of a Buzz machine should be similar to the following example...

```
CMachineInfo const MacInfo =
{
  MT_GENERATOR,                 //  Type
  MI_VERSION,                   //  Version
  0,                            //  Flags
  1,                            //  minTracks
  MAX_TRACKS,                   //  maxTracks
  6,                            //  numGlobalParameters
  2,                            //  numTrackParameters
  pParameters,                  //  Parameters
  0,                            //  numAttributes
  NULL,                         //  Attributes
#ifdef _DEBUG
  "Soft Tone 2 (Debug build)",  //  Name
#else
  "Soft Tone 2",
#endif
  "Softy2_",                    //  ShortName
  "Steve Horne",                //  Author
  "Option1\nOption2"            //  Commands
};
```

The fields in this structure are set up as follows...

Type
This field defines whether the machine is an effect or a generator. Use one of the following symbols...

- MT_GENERATOR
- MT_EFFECT

The MT_MASTER symbol is also available, but it is not clear how or why it should be used.
Version
This should always be set using MI_VERSION. It allows future Buzz versions to identify and work with machines that were developed for older versions. In Buzz v1.04, MI_VERSION is set to 12, so it looks like quite a few variants already exist.
Flags
This field is normally set to zero, but can use any or all of the following flags (combine flags using the bitwise OR operator)...
MIF_MONO_TO_STEREO
Indicates that the machine converts a mono input to a stereo output. Probably only used for effects.
MIF_PLAYS_WAVES
Indicates that the machine plays waves from the wave table. Probably only used for generators.
minTracks
The minimum number of tracks to allow in the pattern editing screen. Seems to be always zero for effects, and always one for generators.
maxTracks
The maximum number of tracks to allow in the pattern editing screen. Seems to be always zero for effects.

For generators, the maximum setting should be based on the speed of your code - the faster it is, the more tracks it can cope with in real time. It should certainly allow at least four or five tracks, so that decent chords can be played.

numGlobalParameters
The number of global parameters which can be secified for the machine in the pattern editing screen. Refer to the section on Setting up parameters for more details.
numTrackParameters
The number of track parameters which can be secified for the machine in the pattern editing screen. Refer to the section on Setting up parameters for more details.
Parameters
Points to an array of parameter definitions. Refer to the section on Setting up parameters for more details.
numAttributes
The number of attributes which can be secified for the machine by right-clicking and selecting 'Attributes' in the context menu. Refer to the section on Setting up attributes for more details.
Attributes
Points to an array of attribute definitions. Refer to the section on Setting up attributes for more details.
Name
The long name for the machine (displayed in the tree view when adding a new machine to a Buzz module). It is normal to use the C preprocessor to choose a different name for the debug version.
ShortName
The short name for the machine, which is the default name given to an instance of the machine when it is added to a Buzz module.
Author
Your route to fame and fortune.
Commands
This field specifies a number of options to add to the right-click context menu for the machine. The menu options are stored in a single string, with '\n' separating the available options.

When one of these options is selected, the Command method of CMachineInterface is called.

The most common use for this is to allow customised editing of the machines settings. It is therefore a prefered alternative to using attributes. Other uses include About boxes and links to authors web sites - though these are probably better handled using the HTML help file.

Setting up parameters

Parameters are the settings which are defined using the pattern editor in Buzz - the ones that can vary while a Buzz song is playing.

Global parameters are those on the left of the pattern screen, which only occur once. Track parameters are on the right of the pattern screen, repeated for each track in the pattern.

Both types of parameter are defined using a structure similar to the following examples...

```
CMachineParameter const paraWForm =
{
  pt_byte,                 //  Type
  "Waveform",              //  Name
  "Shape of waveform",     //  Description
  1,                       //  MinValue
  7,                       //  MaxValue
  0,                       //  NoValue
  MPF_STATE,               //  Flags
  1                        //  DefValue
};

CMachineParameter const paraNote =
{
  pt_note,                 //  Type
  "Note",                  //  Name
  "Note",                  //  Description
  NOTE_MIN,                //  MinValue
  NOTE_OFF,                //  MaxValue
  NOTE_NO,                 //  NoValue
  0,                       //  Flags
  0                        //  DefValue
};
```

The fields are used as follows...

Type
This defines the range of values for the parameter. The possibilities are...
pt_switch
A single hex digit in the pattern editor, which can be set to '0' or '1'.
pt_byte
Two hex digits in the pattern editor, with the maximum range '00' to 'FF'.
pt_word
Four hex digits in the pattern editor, with the maximum range '0000' to 'FFFF'.
pt_note
A note in the pattern editor, e.g. 'C#3'.
With note parameters, fields other than the name and description are never changed from the example above.
Name
The name of the parameter.
Description
The description that is displayed in the status bar when the parameter is edited.
MinValue
The minimum value for the parameter.
MaxValue
The maximum value for the parameter.
NoValue
A value that indicates an unchanged state. This must be outside of the range MinValue to MaxValue. It only applies if the MPF_STATE flag is used.

A special value must be designed as a no change value because the tick method recieves a complete set of parameters on each call - not just the ones that have changed.

Flags
The following flags can be ORed together using the bitwise OR operator...
MPF_WAVE
???
MPF_STATE Indicates that the parameter is persistent - the value is remembered until it is explicitly changed. If this flag is used, the NoValue field must specify a value which indicates no change.

I presume that parameters which don't use MPF_STATE must either act in a one-shot way, or revert immediately to their default value if not set. The usual case, however, is to set the MPF_STATE flag.

MPF_TICK_ON_EDIT
???
DefValue
This specifies the default value of the parameter, which normally only applies when the machine is first added to a Buzz module. It is important to ensure that this is consistent with your machine, as Buzz will not necessarily send this value back in a Tick call.

An array of pointers to the parameter structures is also required, similar to the following...

```
CMachineParameter const *pParameters[] =
{
  // global parameters

  &paraWForm,
  &paraAttack,
  &paraDecay,
  &paraEffectLow,
  &paraEffectHigh,
  &paraDepth,

  // track parameters

  &paraNote,
  &paraVolume
};
```
The machine info structure points to this array, which in turn references all parameters. The parameters are in the same order that they will appear in the pattern editor screen.

Finally, classes similar to the following should be set up...

```
#pragma pack(1)

class gvals  //  Global parameter values
{
  public:
    byte wform;
    word attack;
    word decay;
    word effect_low;
    word effect_high;
    byte depth;
};

class tvals  //  Track parameter values
{
  public:
    byte note;
    byte volume;
};

#pragma pack()
```
These classes are not required, but make the decoding of the parameters much easier.

The Command method

This method is called when one of the context menu options for your machine is selected. A parameter value of zero indicates the first menu option.

The context menu options are specified in the machine info structure.

General advice for working with digital signals

Calculating Note Frequencies

The following calculations are useful for converting note numbers to frequencies...

Convert to Hertz
```
int l_Note = ((tv.note >> 4) * 12) + (tv.note & 0x0f) - 70;

Freq = (440.0 * pow (2.0, ((float) l_Note) / 12.0)) / pmi->pMasterInfo->SamplesPerSec;
```
Convert to Cycles per Sample
```
int l_Note = ((tv.note >> 4) * 12) + (tv.note & 0x0f) - 70;

Freq = (440.0 * pow (2.0, ((float) l_Note) / 12.0)) / pmi->pMasterInfo->SamplesPerSec;
```
They are based on the common definition of middle A as 440 Hz.

Speed Optimizations

- Rather the perform the complete calculation for a particular sample before moving on to the next, it is often better to perform a simple step in the calculation for all samples, buffering the results at each step.

This takes best advantage of the processors instruction cache, and also creates an opportunity to use DSP-libraries which take advantage of MMX and Pentium-III extended DSP-like instruction sets.

- Look for opportunities to pre-calculate values, and therefore move processing from the frequently called Generate method to the less frequently called Tick, Command and Init methods.
- If a particular calculation changes very slowly, don't calculate for every sample. Calculate every other sample (or however few you can get away with) to reduce the processor load.

Alternatively, use a simple interpolation to avoid calculating a complex function on every step.

Be careful with calculations that can produce rapid variations, however.

- When values become very small, disable the processing - calculations on very small values cause exceptions that drastically reduce performance.
- Avoid calling the math.h functions in inner loops.

The CMICallbacks services

MessageBox

Displays a message box, similar to the windows standard version. My guess is that using the standard windows message box at the wrong time could freeze buzz until it was closed, and as message boxes are so useful for debugging a safe version was considered useful. But thats just a guess.

GetAuxBuffer

This function returns the address of an auxiliary buffer, which can be used for intermediate processing during the CMachineInterface::Work function. The address returned is always the same, and is shared by all machines, so the contents must be assumed to be garbage at the start of the Work function.

A lot of machines will not need an auxiliary buffer, and e ven some quite complex machines will only need the one buffer supplied above. However, if more buffers are needed, they should be big enough to contain MAX_BUFFER_LENGTH samples.

ClearAuxBuffer

This function clears the buffer referenced by GetAuxBuffer to all zeros.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <assert.h>
#include <math.h>
#include "../../MachineInterface.h"

#define MAX_TAPS              1
#define   FILTER_SECTIONS     2                              /* 2 filter sections for 24 db/oct filter */

typedef struct
{
    unsigned int length;                                     /* size of filter */
    float *history;                                /* pointer to history in filter */
    double *coef;                                            /* pointer to coefficients of f ilter */
} FILTER;

typedef struct
{
          double a0, a1, a2;                       /* numerator coefficients */
          double b0, b1, b2;                       /* denominator coefficients */
} BIQUAD;

BIQUAD ProtoCoef[FILTER_SECTIONS];                           /* Filter prototype coefficients,
                                                             1 for each filter section  */

void prewarp(
    double *a0, double *a1, double *a2,
    double fc, double fs)
{
    double wp, pi;

    pi = (4.0 * atan(1.0));
    wp = (2.0 * fs * tan(pi * fc / fs));

    *a2 = (*a2) / (wp * wp);
    *a1 = (*a1) / wp;
}

void bilinear(
    double a0, double a1, double a2,     /* numerator coefficients */
    double b0, double b1, double b2,     /* denominator coefficients */
    double *k,                                     /* overall gain factor */
    double fs,                                     /* sampling rate */
    double *coef                                   /* pointer to 4 iir coefficients */
)
{
    double ad, bd;
    ad = (4. * a2 * fs * fs + 2. * a1 * fs + a0);
    bd = (4. * b2 * fs * fs + 2. * b1* fs + b0);
    *k *= ad/bd;
    *coef++ = ((2. * b0  - 8. * b2 * fs * fs) / bd);                              /* beta1 */
    *coef++ = ((4.  * b2 * fs * fs  - 2. * b1 * fs + b0) / bd);          /* beta2 */
    *coef++ = ((2. * a0  - 8. * a2 * fs * fs) / ad);                              /* alpha1 */
    *coef = ((4. * a2 * fs * fs  - 2. * a1 * fs + a0) / ad);           /* alpha2 */
}

void szxform(
    double *a0, double *a1, double *a2,     /* numerator coefficients */
    double *b0, double *b1, double *b2,  /* denominator coefficients */
    double fc,                                     /* Filter cutoff frequency */
    double fs,                                     /* sampling rate */
    double *k,                                     /* overall gain factor */
    double *coef)                                  /* pointer to 4 iir coefficients */
{
          prewarp(a0, a1, a2, fc, fs);
          prewarp(b0, b1, b2, fc, fs);
          bilinear(*a0, *a1, *a2, *b0, *b1, *b2, k, fs, coef);
}

CMachineParameter const paraCutoff =
{
          pt_word,                                           // type
          "Cutoff",
          "Cutoff in Hz",                                    // description
          1,                                                 // MinValue
          22050,                                             // MaxValue
          65535,                                             // NoValue
          MPF_STATE,                                         // Flags
          5000
};

CMachineParameter const paraResonance =
{
          pt_word,                                           // type
          "Resonance",
          "Resonance in units",                              // description
          10,                                                // MinValue
          10000,                                             // MaxValue
          65535,                                             // NoValue
          MPF_STATE,                                         // Flags
          10
};

CMachineParameter const *pParameters[] =
{
          &paraCutoff,
          &paraResonance,
};

#pragma pack(1)
```

```
class gvals
{
public:
          word cutoff;
          word resonance;
};

#pragma pack()

CMachineInfo const MacInfo =
{
          MT_EFFECT,                                        // type
          MI_VERSION,
          0,                                                // flags
          0,                                                // min tracks
          0,                                                // max tracks
          2,                                                // numGlobalParameters
          0,                                                // numTrackParameters
          pParameters,
          0,
          NULL,
#ifdef _DEBUG
          "asedev a4pFilter01 (Debug build)",     // name
#else
          "asedev a4pFilter01",
#endif
          "a4pFilter",                                      // short name
          "ase development",                      // author
          NULL
};

class mi : public CMachineInterface
{
public:
          mi();
          virtual ~mi();

          virtual void Init(CMachineDataInput * const pi);
          virtual void Tick();
          virtual bool Work(float *psamples, int numsamples, int const mode);

          virtual char const *DescribeValue(int const param, int const value);


private:

          float iir_filter(float input);

private:
          FILTER    iir;
          float     Cutoff;
          float     Resonance;

private:

          gvals gval;

};

DLL_EXPORTS

mi::mi()
{
          AttrVals = NULL;
          GlobalVals = &gval;
          TrackVals = NULL;
}

mi::~mi()
{
          delete[] iir.coef;
          delete[] iir.history;
}

char const *mi::DescribeValue(int const param, int const value)
{
          static char txt[16];

          switch(param)
          {
          case 0:
                    sprintf(txt, "%.0fHz", (float)(value));
                    break;
          case 1:
                    sprintf(txt, "%.1f", (float)(value / 10.0));
                    break;
          default:
                    return NULL;
          }

          return txt;
}

void mi::Init(CMachineDataInput * const pi)
{
          /* Section 1 */
          ProtoCoef[0].a0 = 1.0;
          ProtoCoef[0].a1 = 0;
          ProtoCoef[0].a2 = 0;
          ProtoCoef[0].b0 = 1.0;
          ProtoCoef[0].b1 = 0.765367;
          ProtoCoef[0].b2 = 1.0;
          /* Section 2 */
          ProtoCoef[1].a0 = 1.0;
          ProtoCoef[1].a1 = 0;
          ProtoCoef[1].a2 = 0;
          ProtoCoef[1].b0 = 1.0;
          ProtoCoef[1].b1 = 1.847759;
          ProtoCoef[1].b2 = 1.0;

          iir.length = FILTER_SECTIONS;            /* Number of filter sections */

          //allocate memory
          iir.coef = (double *) calloc(4 * iir.length + 1, sizeof(double));
          iir.history = (float *) calloc(2*iir.length,sizeof(float));

          Resonance = 1.0f;                                  /* preset Resonance */
          Cutoff = 5000;                                     /* preset Filter cutoff (Hz) */

}

void mi::Tick()
{
          double              *coef;
          unsigned  nInd;
          double              a0, a1, a2, b0, b1, b2, k;
          bool                doit;

          doit = false;

          if (gval.cutoff != paraCutoff.NoValue)
          {
                    Cutoff = (float)(gval.cutoff);
                    doit = true;
          }
          if (gval.resonance != paraResonance.NoValue)
          {
                    Resonance = (float)((gval.resonance) / 10.0);
                    doit = true;
          }

          if (doit=true)
          {
                    k = 1.0;
                    coef = iir.coef + 1;
                    for (nInd = 0; nInd < iir.len gth; nInd++)
                    {
                              a0 = ProtoCoef[nInd].a0;
                              a1 = ProtoCoef[nInd].a1;
                              a2 = ProtoCoef[nInd].a2;
                              b0 = ProtoCoef[nInd].b0;
                              b1 = ProtoCoef[nInd].b1 / Resonance;
                              b2 = ProtoCoef[nInd].b2;
                              szxform(&a0, &a1, &a2, &b0, &b1, &b2, Cutoff, pMasterIn fo->SamplesPerSec, &k, coef);
                              coef += 4;
                    }
                    iir.coef[0] = k;
          }
}

/*
 * --------------------------------------------------------------------
 *
 * iir_filter - Perform IIR filtering sample by sample on floats
 *
 * Implements cascaded direc t form II second order sections.
 * Requires FILTER structure for history and coefficients.
 * The length in the filter structure specifies the number of sections.
 * The size of the history array is 2*iir ->length.
 * The size of the coefficient array is 4 *iir->length + 1 because
 * the first coefficient is the overall scale factor for the filter.
 * Returns one output sample for each input sample.  Allocates history
 * array if not previously allocated.
 *
 * float iir_filter(float input,FILTER *iir)
 *
 *     float input        new float input sample
 *     FILTER *iir        pointer to FILTER structure
 *
 * Returns float value giving the current output.
 *
 * Allocation errors cause an error message and a call to exit.
 * ---------------------------- ----------------------------------------
 */
//float iir_filter(float input,FILTER *iir)
float mi::iir_filter(float input)
{
    unsigned int i;
    float *hist1_ptr,*hist2_ptr;
          double *coef_ptr;
    float output,new_hist,history1,history2;

    coef_ptr = iir.coef;                /* coefficient pointer */

    hist1_ptr = iir.history;            /* first history */
    hist2_ptr = hist1_ptr + 1;           /* next history */

    output =(float) (input * (*coef_ptr++));

    for (i = 0 ; i < iir.length; i+ +)
          {
        history1 = *hist1_ptr;                                                                        /* history values */
        history2 = *hist2_ptr;

        output = (float) (output  - history1 * (*coef_ptr++));
        new_hist = (float) (output  - history2 * (*coef_ptr++));    /* poles */

        output = ( float) (new_hist + history1 * (*coef_ptr++));
        output = (float) (output + history2 * (*coef_ptr++));      /* zeros */

        *hist2_ptr++ = *hist1_ptr;
        *hist1_ptr++ = new_hist;
        hist1_ptr++;
        hist2_ptr++;
    }

    return(output);
}

bool mi::Work(float *psamples, int numsamples, int const mode)
{
          if (mode == WM_WRITE || mode == WM_NOIO)
                    return false;

          if (mode == WM_READ)
                    return true;

          do
          {
                    *psamples = iir_filter(*psamples);
                    psamples++;
          } while(--numsamples);

          return true;
}
```

```
//  Amplitude Modulator SourceCode.
//  It shows how to demultiplex two multiplexed signals
//  and then perform an Amplitude Modulation.
//  Any part of this code can be used freely.
//  If any good programmer finds a ny big mistake in
//  this code please tell it to me.  I'd like
//  to fix it.

//  The interesting part is only the mi::work() function.

//   Ynzn
#include "malloc.h"
#include "MachineInterface.h"
#include "..\dsplib\dsplib.h"

CMachineParameter const pa raModCutOff =
{
          pt_word,
          "ModCutOff",
          "Modulator Filter CutOff",
          100,
          11025,
          0,
          MPF_STATE,
          11025
};

CMachineParameter const paraResonance =
{
          pt_byte,
          "Resonance",
          "Filter Resonance",
          0,
          128,
          255,
          MPF_STATE,
          64
};

CMachineParameter const  *pParameters[] =
{
          &paraModCutOff,
          &paraResonance,
};

#pragma pack(1)

class gvals
{
public:
          word modcutoff;
          byte resonance;
};

#pragma pack()

CMachineInfo const MacInfo =
{
          MT_EFFECT,  // type
          MI_VERSION, // version
          0, // flags
          0, // mintracks
          0, // máxtracks
          2, // globalparams
          0, // trackparams
          pParameters,  // paramlist
          0,  // attributes
          NULL, // attribute list
          "DeMux + Amplitude Modulation",  // name of the machine
          "DeMux+AM",  // short name
          "Ynzn", // I am the creator
          NULL  // cmdlist
};

//  I always get lost with CMachineInfo members.  I have to
//  put those comments.  Sorry.

class mi:public CMachineInterface
{
public:
          mi();
          virtual ~mi();

          virtual void Init(CMachineDataInput *const pi);
          virtual void Tick();
          virtual boo l Work(float *psamples, int numsamples, int const mode);

private:
          gvals gval;
          CBWState ModFilt;
          CBWState ModRes;
          float Modcutoff;
          float Resonance;
};

DLL_EXPORTS

mi::mi()
{
          GlobalVals = &gval;
          AttrVals = NULL;
          TrackVals = NULL;
}

mi::~mi()
{
}

void mi::Init(CMachineDataInput * const pi)
{
          DSP_BW_Reset(ModFilt);
          Modcutoff=11025;
          Resonance=64;
}

void mi::Tick()
{
          if (gval.modcutoff != paraModCutOff.NoValue)
          {
                    Modcutoff=gval.modcutoff;
                    DSP_BW_InitLowpass(ModFilt, Modcutoff);
          }

          if (gval.resonance != paraResonance.NoValue)
          {
                    Resonance=gval.resonance;
                    DSP_BW_InitBandpass(ModRes, Modcutoff,4);
          }
}

bool mi::Work(float *psamples, int numsamples, int const mode)
{
          if (mode==WM_READWRITE)
          {
          float *aux=(float *)malloc(numsamp les);

          DSP_Copy(psamples,aux,numsamples);
          DSP_BW_Work(ModFilt, psamples, numsamples, mode);
          DSP_BW_Work(ModRes, aux, numsamples, mode);

                    do{
                    *psamples+=*aux;
                    aux++;
                    psamples++;
                    }while(--numsamples);

          return true;
          }
          else
          return false;
}

// That's all folks!
// Ynzn
// cjan5813@alu-etsetb.upc.es
```

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <assert.h>
#include <math.h>
#include "../MachineInterface.h"

double const SilentEnough = 1.0 / log(2);

#define MAX_TAPS              8

CMachineParameter const paraLength =
{
          pt_word,                                        // type
          "Length",
          "Length in length units",                                                       // description
          1,                                                                    // MinValue
          32768,                                                      // MaxValue
          65535,                                                      // NoValue
          MPF_STATE,                                                  // Flags
          3
};

#define UNIT_TICK             0
#define UNIT_MS                         1
#define UNIT_SAMPLE           2
#define UNIT_256              3

CMachineParameter const paraDryThru =
{
          pt_switch,                                                  // type
          "Dry thru",
          "Dry thru (1 = yes, 0 = no)",                   // description
          -1,                                                                   // MinValue
          -1,                                                                   // MaxValue
          SWITCH_NO,                                                  // NoValue
          MPF_STATE,                                                  // Flags
          SWITCH_ON
};

CMachineParameter const paraUnit =
{
          pt_byte,                                        // type
          "Length unit",
          "Length unit (0 = tick (default), 1 = ms, 2 = sample, 3 = 256th of tick)",
          0,                                                                    // MinValue
          3,                                                                    // MaxValue
          0xff,                                                       // NoValue
          MPF_STATE,                                                  // Flags
          0
};


CMachineParameter const paraFeedback =
{
          pt_byte,                                        // type
          "Feedback",
          "Feedback (00 = -100%, 40=0%, 80 = 100%)",      // description
          0,                                              // MinValue
          128,                                            // MaxValue
          255,                                            // NoValue
          MPF_STATE,                                      // Flags
          0x60
};

CMachineParameter const paraWetOut =
{
          pt_byte,                                                    // type
          "Wet out",
          "Wet out (00 = 0%, 80 = 100%)",                             // description
          0,                                                          // MinValue
          128,                                                        // MaxValue
          255,                                                        // NoValue
          MPF_STATE,                                                  // Flags
          0x30
};


CMachineParameter const *pParameters[] =
{
          &paraDryThru,
          &paraLength,
          &paraUnit,
          &paraFeedback,
          &paraWetOut,
};

CMachineAttribute const attrMaxDelay =
{
          "Max Delay (ms)",
          1,
          1000000,
          10000
};

CMachineAttribute const *pAttributes[] =
{
          &attrMaxDelay
};

#pragma pack(1)
```

```
class gvals
{
public:
        byte drythru;
};

class tvals
{
public:
        word length;
        byte unit;
        byte feedback;
        byte wetout;
};

class avals
{
public:
        int maxdelay;
};

#pragma pack()

CMachineInfo const MacInfo =
{
        MT_EFFECT,                                              // type
        MI_VERSION,
        0,                                                      // flags
        1,                                                      // min tracks
        MAX_TAPS,                                       // max tracks
        1,                                                      // numGlobalParameters
        4,                                                      // numTrackParameters
        pParameters,
        1,
        pAttributes,
#ifdef _DEBUG
        "Jeskola Delay (Debug build)",                          // name
#else
        "Jeskola Delay",
#endif
        "Delay",                                                // short name
        "Oskari Tammelin",                              // author
        NULL
};

class CTrack
{
public:
        float *Buffer;
        int Length;
        int Pos;
        float Feedback;
        float WetOut;
        int Unit;
};


class mi : public CMachineInterface
{
public:
        mi();
        virtual ~mi();

        virtual void Init(CMachineDataInput * const pi);
        virtual void Tick();
        virtual bool Work(float *psamples, int numsamples, int const mode);

        virtual void SetNumTracks(int const n);

        virtual void AttributesChanged();

        virtual char const *DescribeValue(int const param, int const value);

private:
        void InitTrack(int const i);
        void ResetTrack(int const i);

        void TickTrack(CTrack *pt, tvals *ptval);
        void WorkTrack(CTrack *pt, float  *pin, float *pout, int numsamples, int const mode);

private:
        int MaxDelay;       // in samples
        int IdleCount;      // in samples
        int DelayTime;
        bool IdleMode;
        bool DryThru;

private:
        int numTracks;
        CTrack Tracks[MAX_TAPS];

        avals aval;
        gvals gval;
        tvals tval[MAX_TAPS];


};

DLL_EXPORTS

mi::mi()
{
        GlobalVals = &gval;
```

```
          TrackVals = tval;
          AttrVals = (int *)&aval;
}

mi::~mi()
{
          for (int c = 0; c < MAX_TAPS; c++)
          {
                    delete[] Tracks[c].Buffer;
          }
}

char const *mi::DescribeValue(int const param, int const  value)
{
          static char txt[16];

          switch(param)
          {
          case 1:             // length
                    return NULL;
                    break;
          case 2:             // unit
                    switch(value)
                    {
                    case 0: return "tick";
                    case 1: return "ms";
                    case 2: return "sample";
                    case 3: return "tick/256";
                    }
                    break;
          case 3:             // feedback
                    sprintf(txt, "%.1f%%", (double)(value -64) * (100.0 / 64.0));
                    break;
          case 4:             // wetout
                    sprintf(txt, "%.1f%%", (double)value * (100.0 / 128.0));
                    break;
          default:
                    return NULL;
          }

          return txt;
}



void mi::Init(CMachineDataInput *  const pi)
{
          numTracks = 1;
          DryThru = true;

          for (int c = 0; c < MAX_TAPS; c++)
          {
                    Tracks[c].Buffer = NULL;
                    Tracks[c].Unit = UNIT_TICK;
                    Tracks[c].Length = pMasterInfo ->SamplesPerTick * 3;
                    Tracks[c].Pos = 0;
                    Tracks[c].Feedback = 0.3f;
                    Tracks[c].WetOut = 0;
          }

          Tracks[0].WetOut = 0.3f;       // enable first track

          IdleMode = true;
          IdleCount = 0;

}

void mi::AttributesChanged()
{
          MaxDelay = (int)(pMasterInfo ->SamplesPerSec * (aval.maxdelay / 1000.0));
          for (int c = 0; c < numTracks; c++)
                    InitTrack(c);
}


void mi::SetNumTracks(int const n)
{
          if (numTracks < n)
          {
                    for (int c = numTracks; c < n; c++)
                              InitTrack(c);
          }
          else if (n < numTracks)
          {
                    for (int c = n; c < numTracks; c++)
                              ResetTrack(c);

          }
          numTracks = n;

}


void mi::InitTrac k(int const i)
{
          delete[] Tracks[i].Buffer;
          Tracks[i].Buffer = new float [MaxDelay];
          memset(Tracks[i].Buffer, 0, MaxDelay*4);
          Tracks[i].Pos = 0;
          if (Tracks[i].Length > MaxDelay)
                    Tracks[i].Length = MaxDelay;
}

void mi::ResetTrack(int const i)
```

```
{
          delete[] Tracks[i].Buffer;
          Tracks[i].Buffer = NULL;
}


void mi::TickTrack(CTrack *pt, tvals *ptval)
{
          if (ptval->unit != paraUnit.NoValue)
                    pt->Unit = ptval->unit;

          if (ptval->length != paraLength.NoValue)
          {
                    switch(pt->Unit)
                    {
                    case UNIT_MS:
                              pt->Length = (int)(pMasterInfo->SamplesPerSec * (ptval->length / 1000.0));
                              if (pt->Length < 1)
                                        pt->Length = 1;
                              break;
                    case UNIT_SAMPLE:
                              pt->Length = ptval->length;
                              break;
                    case UNIT_TICK:
                              pt->Length = pMasterInfo->SamplesPerTick * ptval->length;
                              break;
                    case UNIT_256:
                              pt->Length = (pMasterInfo->SamplesPerTick * ptval->length) >> 8;
                              if (pt->Length < 1)
                                        pt->Length = 1;
                              break;
                    }

          }

          if (pt->Length > MaxDelay)
          {
                    pt->Length = MaxDelay;
          }


          if (pt->Pos >= pt->Length)
                    pt->Pos = 0;

          if (ptval->feedback != paraFeedback.NoValue)
          {
                    pt->Feedback = (float)((ptval->feedback - 64) * (1.0 / 64.0));
          }

          if (ptval->wetout != paraWetOut.NoValue)
                    pt->WetOut = (float)(ptval->wetout * (1.0 / 128.0));

}

void mi::Tick()
{
          for (int c = 0; c < numTracks; c++)
                    TickTrack(Tracks + c, tval+c);


          // find max delay time slow we know when to stop wasting CPU time
          int maxdt = 0;
          for (c = 0; c < numTracks; c++)
          {
                    int dt = Tracks[c].Length + (int)(SilentEnough / log(fabs(Tracks[c].Feedback)) * Tracks[c].Length);
                    if (dt > maxdt)
                              maxdt = dt;
          }

          DelayTime = maxdt;



          if (gval.drythru != SWITCH_NO)
                    DryThru = gval.drythru != 0;
}

#pragma optimize ("a", on)

static void DoWork(float *pin, float *pout, float *pbuf, int c, double const wetout, double const feedback)
{
          do
          {

                    double delay = *pbuf;
                    *pbuf++ = (float)(feedback * delay + *pin++);
                    *pout++ += (float)(delay * wetout);

          } while(--c);
}

static void DoWorkNoFB(float *pin, float *pout, float *pbuf, int c, double const wetout)
{
          do
          {

                    double delay = *pbuf;
                    *pbuf++ = (float)(*pin++);
                    *pout++ += (float)(delay * wetout);

          } while(--c);
}

static void DoWorkNoInput(float *pout, float *pbuf, int c, double const wetout, double const feedback)
```

```
{
	do
	{
		double delay = *pbuf;
		*pbuf++ = (float)(feedback * delay);
		*pout++ += (float)(delay * wetout);

	} while(--c);
}



static void DoWorkNoInputNoFB(float *pout, float *pbuf, int c, double const wetout)
{

	do
	{
		double delay = *pbuf;
		*pbuf++ = 0;
		*pout++ += (float)(delay * wetout);
	} while(--c);
}

static void DoWorkNoInputNoOutput(float *pbuf, int c, double const feedback)
{
	do
	{
		*pbuf = (float)(*pbuf * feedback);
		pbuf++;
	} while(--c);
}

static void DoWorkNoOutput(float *pin, float *pbuf, int  c, double const feedback)
{
	do
	{

		double delay = *pbuf;
		double dry = *pin++;
		*pbuf++ = (float)(feedback * delay + dry);

	} while(--c);
}

static void DoWorkNoOutputNoFB(float *psamples, float *pbuf, int c)
{
	memcpy(pbuf, psamples, c*4);
}


#pragma optimize ("", on)


void mi::WorkTrack(CTrack *pt, float *pin, float *pout, int numsamples, int const mode)
{
	do
	{
		int count = __min(numsamples, pt ->Length - pt->Pos);

		if (count > 0)
		{
			if (mode == WM_NOIO)
			{
				if (pt->Feedback != 0)
					DoWorkNoInputNoOutput(pt ->Buffer + pt ->Pos, count, pt ->Feedback);
			}
			else if (mode == WM_WRITE)
			{
				if (pt->Feedback != 0)
					DoWorkNoInput(pout, pt ->Buffer + pt ->Pos, count, pt ->WetOut, pt ->Feedback);
				else
					DoWorkNoInputNoFB(pout, pt ->Buffer + pt ->Pos, count, pt ->WetOut);
			}
			else if (mode == WM_READ)
			{
				if (pt->Feedback != 0)
					DoWorkNoOutput(pin, pt ->Buffer + pt ->Pos, count, pt ->Feedback);
				else
					DoWorkNoOutputNoFB(pin, pt ->Buffer + pt ->Pos, count);

			}
			else
			{
				if (pt->Feedback != 0)
					DoWork(pin, pout, pt ->Buffer + pt ->Pos, count, pt ->WetOut, pt ->Feedback);
				else
					DoWorkNoFB(pin, pout, pt ->Buffer + pt ->Pos, count, pt ->WetOut);
			}

			pin += count;
			pout += count;
			numsamples -= count;
			pt->Pos += count;
		}

		if (pt ->Pos == pt ->Length)
			pt->Pos = 0;

	} while(numsamples > 0);

}

bool mi::Work(float *psamples, int numsamples, int const mode)
{
        if (mode & WM_READ)
        {
                IdleMode = false;
                IdleCount = 0;
        }
        else
        {
                if (IdleMode)
                {
                        return false;
                }
                else
                {
                        IdleCount += numsamples;
                        if (IdleCount >= DelayTime + MAX_BUFFER_LENGTH)
                        {
                                // clear all buffers
                                for (int c = 0; c < numTracks; c++)
                                        memset(Tracks[c].Buffer, 0, Tracks[c].Length*4);
                                IdleMode = true;
                        }
                }
        }

        float *paux = pCB->GetAuxBuffer();

        if (mode & WM_READ)
                memcpy(paux, psamples, numsamples*4);

        if (!DryThru || !(mode & WM_READ))
                memset(psamples, 0, numsamples*4);

        for (int c = 0; c < numTracks; c++)
                WorkTrack(Tracks + c, paux, psamples, numsamples, mode);

        return true;
}
```

```
#include <string.h>
#include <stdlib.h>
#include <assert.h>
#include <math.h>
#include "../MachineInterface.h"

CMachineParameter const paraThreshold =
{
          pt_word,                                    // type
          "+threshold",
          "Positive threshold level",                                          // description
          0,                                                         // MinValue
          0xfffe,                                          // MaxValue
          0xffff,                                          // NoValue
          MPF_STATE,                                       // Flags
          0x200,
};

CMachineParameter const paraClamp =
{
          pt_word,                                    // type
          "+clamp",
          "Positive clamp  level",          // description
          0,                                                         // MinValue
          0xfffe,                                          // MaxValue
          0xffff,                                          // NoValue
          MPF_STATE,                                       // Flags
          0x1000
};

CMachineParameter const paraNegThreshold =
{
          pt_word,                                    // type
          "-threshold",
          "Negative threshold level",                                          // description
          0,                                                         // MinValue
          0xfffe,                                          // MaxValue
          0xffff,                                          // NoValue
          MPF_STATE,                                       // Flags
          0x200
};

CMachineParameter const paraNegClamp =
{
          pt_word,                                    // type
          "-clamp",
          "Negative clamp level",           // description
          0,                                                         // MinValue
          0xfffe,                                          // MaxValue
          0xffff,                                          // NoValue
          MPF_STATE,                                       // Flags
          0x1000
};

CMachineParameter const paraAmount =
{
          pt_byte,                                    // type
          "Amount",
          "Amount",                                   // description
          0,                                                         // MinValue
          0x7f,                                            // MaxValue
          0xff,                                       // NoValue
          MPF_STATE,                                       // Flags
          0x7f
};

CMachineParameter const *pParameters[] =
{
          // global
          &paraThreshold,
          &paraClamp,
          &paraNegThreshold,
          &paraNegClamp,
          &paraAmount

};

CMachineAttribute const attrSymmetric =
{
          "Symmetric",
          0,
          1,
          0
};

CMachineAttribute const *pAttributes[] =
{
          &attrSymmetric
};


#pragma pack(1)

class gvals
{
public:
          word threshold;
          word clamp;
          word negthreshold;
          word negclamp;
          byte amount;
```

```
};

class avals
{
public:
          int symmetric;
};

#pragma pack()

CMachineInfo const MacInfo =
{
          MT_EFFECT,                              // type
          MI_VERSION,
          0,                                                // flags
          0,                                                // min tracks
          0,                                                // max tracks
          5,                                                // numGlobalParameters
          0,                                                // numTrackParameters
          pParameters,
          1,
          pAttributes,
#ifdef _DEBUG
          "Jeskola Distortion  (Debug build)",              // name
#else
          "Jeskola Distortion",                                                 // name
#endif
          "Dist",                                 // short name
          "Oskari Tammelin",                                                    // author
          NULL
};


class mi : public CMachineInterface
{
public:
          mi();
          virtual ~mi();

          virtual void Init(CMachineDataInput * c onst pi);
          virtual void Tick();
          virtual bool Work(float *psamples, int numsamples, int const mode);

private:



private:
          float Threshold;
          float Clamp;
          float NegThreshold;
          float NegClamp;
          float Amount;

          gvals gval;
          avals aval;

};

DLL_EXPORTS

mi::mi()
{
          GlobalVals = &gval;
          AttrVals = (int *)&aval;
}

mi::~mi()
{
}

void mi::Init(CMachineDataInput * const pi)
{
          Threshold = 65536 * 16.0;
          Clamp = 65536 * 16.0;
          NegThreshold = -65536 * 16.0;
          NegClamp = -65536 * 16.0;
          Amount = 1.0;
}

void mi::Tick()
{
          if (gval.threshold != paraThreshold.NoValue)
                    Threshold = (float)(gval.threshold * 16.0);

          if (gval.clamp != paraClamp.NoValue)
                    Clamp = (float)(gval.clamp * 16.0);

          if (gval.negthreshold != paraNegThreshold.NoValue)
                    NegThreshold = (floa t)(gval.negthreshold *  -16.0);

          if (gval.negclamp != paraNegClamp.NoValue)
                    NegClamp = (float)(gval.negclamp *  -16.0);

          if (gval.amount != paraAmount.NoValue)
                    Amount = (float)(gval.amount * (1.0 / 0x7f));
}


bool mi::Work(float *psamples, int numsamp les, int const mode)
{
          if (mode == WM_WRITE || mode == WM_NOIO)
                    return false;
```

```
        if (mode == WM_READ || Amount == 0)
                return true;

        double const drymix = 1.0  - Amount;
        float const clamp = (float)(Amount * Clamp);
        double const threshold = Threshold;
        float negclamp;
        double negthreshold;

        if (aval.symmetric)
        {
                negthreshold = -threshold;
                negclamp = -clamp;
        }
        else
        {
                negthreshold = NegThreshold;
                negclamp = (float)(Amount * NegClamp);
        }


        if (drymix < 0.001)
        {
                do
                {
                        double const s = *psamples;

                        if (s >= threshold)
                                *psamples = clamp;
                        else if (s <= negthreshold)
                                *psamples = negclamp;

                        psamples++;

                } while(--numsamples);
        }
        else
        {
                do
                {
                        double const s = *psamples;

                        if (s >= threshold)
                                *psamples = (float)(s * drymix + clamp);
                        else if (s <= negthreshold)
                                *psamples = (float)(s * drymix + negclamp);

                        psamples++;

                } while(--numsamples);
        }

        return true;
}
```

```
/*
 just got an idea how to make a fast bassdrum algorithm without use   of
trigonometrical functions or lookup tables
 in the inner loop. this one uses only 8 fmuls, 4 fadds...

 the algo is easy:
   we have two complex numbers "dr" and "ddr".
   "dr" specifies an angular rotation of "da" radians anticlockwise, while
"ddr" specifies a much smaller angular
   rotation clockwise. now, in the main loop we just take some third complex
number "r" and continuously multiply
   it by "dr" -- this way we perform a continuous rotation. this would
produce a steady sinusoidal waveform (i. e.
   constant frequency). now, if we also start multiplying "dr" by "ddr"
which goes the opposite direction, "dr"
   will steadily become smaller (i.e. the rotation angles will get smaller
and thus the frequency smaller too).
   what we've got here is sim ple frequency modulation with the frequency
being high at the start and continuously
   getting smaller, i.e. the tune starting at a high pitch and ending at a
low one, and that's exactly how bassdrum
   sounds are generated.
   there are a lot of other wa ys do make bassdrum sounds though. some do it
by filtering white noise starting at a
   high cutoff frequency and ending at a low one. to give this sound a more
"natural" touch you may also add some
   noise, perform some clipping etc.
   feel free to expe riment! hearing the same odd 909's all the time is
getting boring anyway.. we should always be
   in search of new exciting sounds ;)

 btw: there are faster algorithms available to calculate steady sinus
waveforms than the complex exponential one
 i use here...

 mail me if you want the CSoundIO source too, which is not by me, but an
adaptation i made from the directsound
 playback code of stk98 by perry r cook.

 (c) 1998 jan marguc aka fontex^k
*/

#include <conio.h>
#include <math.h>
#include "soundio.h"

const blocksize = 256;

class cplx
{
public:
 double real;
 double imag;

 cplx ()
 {
  real = 0.0;
  imag = 0.0;
 };

 cplx (double real, double imag)
 {
  this->real = real;
  this->imag = imag;
 };

 void operator *= (const cplx &c)
 {
  double tmp = real * c.real - imag * c.imag;
  imag = real * c.imag + imag * c.real;
  real = tmp;
 };

 void set (double real, double imag)
 {
  this->real = real;
  this->imag = imag;
 };
};

main ()
{
 short *buffer = new short [blocksize];

 CSoundIO sound (11025, 1, blocksize);

 const double da = 0.2;
 const double dda = -0.00025;

 cplx r;
 cplx dr;
 cplx ddr;

 while (!kbhit ())
 {
  // this is the only place that we use trigonometric functions
  r.set (1.0, 0.0);
  dr.set (cos (da), sin (da));
  ddr.set (cos (dda), sin (dda));

  for (int n = 0; n < 20; n++)
  {
   for (int t = 0; t < blocksize; t++)
   {
    double tmp = 0.0;

    if (dr.imag > 0.0)
    {
     // distortion -- simple clipping
     tmp = r.real * 170000.0;
     if (tmp < -32767.0) tmp = -32767.0;
     if (tmp >  32767.0) tmp =  32767.0;

     r *= dr;
     dr *= ddr;
    }

    buffer [t] = tmp;
   }

   sound.playBuffer (buffer);
  }
 }
}
```

```
// M3 Buzz plugin by MAKK makk@gmx.de
// released on 04-21-99
// Thanks must go to Robert Bristow-Johnson pbjrbj@viconet.com
// a.k.a. robert@audioheads.com for his excellent
// Cookbook formulas for the filters.
// The code is not really speed optimized
// and compiles with many warnings  - i'm to lazy to correct
// them all (mostly typecasts).
// Use the source for your own plugins if you want, but don't
// rip the hole machine please.
// Thanks in advance. MAKK

#include <string.h>
#include <stdlib.h>
#include <assert.h>
#include <math.h>
#include <float.h>
#include "../MachineInterface.h"
#include "../dsplib/dsplib.h"


#pragma optimize ("a", on)

#define MAX_TRACKS                                                              4

#define EGS_NONE                                0
#define EGS_ATTACK                              1
#define EGS_SUSTAIN                              2
#define EGS_RELEASE                             3


float *freqTab = new float [NOTE_MAX+1];
float *coefsTab = new float [4*128*128*8];
float *LFOOscTab = new float [0x10000];

CMachineParameter const paraNote =
{
        pt_note,                                        //  type
        "Note",
        "Note",                               //  description
        NOTE_MIN,                                       //  Min
        NOTE_MAX,                                       //  Max
        0xff,                                           //  NoValue
        0,                                                      //  Flags
        0                                                       //  default
};

CMachineParameter const paraWave1 =
{
        pt_byte,                                        //  type
        "Osc1Wav",
        "Oscillator 1 Waveform",              //  description
        0,                                                      //  Min
        5,                                              //  Max
        0xff,                                           //  NoValue
        MPF_STATE,                                      //  Flags
        0                                                       //  default
};

CMachineParameter const paraPulseWidth1 =
{
        pt_byte,                                        //  type
        "PulseWidth1",
        "Oscillator 1 Pulse Width",                                         // description
        0,                                                      //  Min
        127,                                            //  Max
        0xff,                                           //  NoValue
        MPF_STATE,                                      //  Flags
        0x40                                                    //  default
};

CMachineParameter const paraWave2 =
{
        pt_byte,                                        //  type
        "Osc2Wav",
        "Oscillator 2 Waveform",              //  description
        0,                                                      //  Min
        5,                                              //  Max
        0xff,                                           //  NoValue
        MPF_STATE,                                      //  Flags
        0                                                       //  default
};

CMachineParameter const paraPulseWidth2 =
{
        pt_byte,                                        //  type
        "PulseWidth2",
        "Oscillator 2 Pulse Width",                                         // description
        0,                                                      //  Min
        127,                                            //  Max
        0xff,                                           //  NoValue
        MPF_STATE,                                      //  Flags
        0x40                                            //  default
};


CMachineParameter const paraMix =
{
        pt_byte,                                        //  type
        "Mix",
        "Mix Osc1 <-> Osc2",
        0,                                                      //  Min
        127,                                                    //  Max
        0xff,                                                   //  NoValue
        MPF_STATE,                                              //  Flags
        0x40                                                    //  default
};

CMachineParameter const paraMixType =
{
        pt_byte,                                                //  type
        "MixType",
        "MixType",                                              //  description
        0,                                                                //  Min
        8,                                                                //  Max
        0xff,                                                   //  NoValue
        MPF_STATE,                                              //  Flags
        0                                                                 //  default
};



CMachineParameter const paraSync =
{
        pt_switch,                                              //  type
        "Sync",
        "Sync: Osc2 synced b y Osc1",                                      // description
        SWITCH_OFF,                                             //  Min
        SWITCH_ON,                                              //  Max
        SWITCH_NO,                                              //  NoValue
        MPF_STATE,                                              //  Flags
        SWITCH_OFF                                              //  default
};



CMachineParameter const paraDetuneSemi=
{
        pt_byte,                                                //  type
        "Semi Detune",
        "Semi Detune in  Halfnotes",                                               // description
        0,                                                                //  Min
        127,                                                    //  Max
        0xff,                                                   //  NoValu e
        MPF_STATE,                                              //  Flags
        0x40                                                    //  default
};

CMachineParameter const paraDetuneFine=
{
        pt_byte,                                                //  type
        "Fine Detune",
        "Fine Detune",                                          //  description
        0,                                                                //  Min
        127,                                                    //  Max
        0xff,                                                   //  NoValue
        MPF_STAT E,                                             //  Flags
        0x40                                                              //  default
};

CMachineParameter const paraGlide =
{
        pt_byte,                                                //  type
        "Glide",
        "Glide",                                                //  description
        0,                                                                //  Min
        127,                                                    //  Max
        0xff,                                                   //  NoValue
        MPF_STATE,                                              //  Flags
        0                                                                 //  default
};

CMachineParameter const paraSubOscWave =
{
        pt_byte,                                                //  type
        "SubOscWav",
        "Sub Oscillator Wav eform",                                                // description
        0,                                                                //  Min
        4,                                                      //  Max
        0xff,                                                   //  NoValue
        MPF_STATE,                                              //  Flags
        0                                                                 //  default
};

CMachineParameter const paraSubOscVol =
{
        pt_byte,                                                //  type
        "SubOscVol",
        "Sub Oscillator Volume",                     //  description
        0,                                                                //  Min
        127,                                                    //  Max
        0xff,                                                   //  NoValue
        MPF_STATE,                                              //  Flags
        0x40                                                    //  default
};

CMachineParameter const paraVolume =
{
        pt_byte,                                                //  type
        "Volume",
        "Volume (Sustain -Level)",                                                 // description
        0,                                                                //  Min
        127,                                                    //  Max
        0xff,                                                   //  NoValue
        MPF_STATE,                                              //  Flags
        0x40                                                    //  default
```

```
};

CMachineParameter const paraAEGAttackTime =
{
        pt_byte,                                        //  type
        "AEGAttack",
        "AEG At tack Time in ms",                  //  description
        0,                                                        //  Min
        127,                                            //  Max
        0xff,                                           //  NoValue
        MPF_STATE,                                      //  Flags
        10                                                        //  default
};

CMachineParameter const paraAEGSustainTime =
{
        pt_byte,                                        //  type
        "AEGSustain",
        "AEG Sustain Time in  ms",                                          // description
        0,                                                        //  Min
        127,                                            //  Max
        0xff,                                           //  NoValue
        MPF_STATE,                                      //  Flags
        50                                                        //  default
};

CMachineParameter const paraAEGReleaseTime =
{
        pt_byte,                                        //  type
        "AEGRelease",
        "AEG Release Time in ms",                                          // description
        0,                                                        //  Min
        127,                                            //  Max
        0xff,                                           //  NoValue
        MPF_STATE,                                      //  Flags
        30                                                        //  default
};

CMachineParameter const paraFilterType =
{
        pt_byte,                                        //  type
        "FilterType",
        "Filter Type ... 0=LP 1=HP 2=BP 3=BR",             // description
        0,                                                        //  Min
        3,                                              //  Max
        0xff,                                           //  NoValue
        MPF_STATE,                                      //  Flags
        0                                                         //  default
};

CMachineParameter const paraCutoff =
{
        pt_byte,                                        //  type
        "Cutoff",
        "Filter Cutoff Frequency",                                         // description
        0,                                                        //  Min
        127,                                            //  Max
        0xff,                                           //  NoValue
        MPF_STATE,                                      //  Flags
        127                                                       //  default
};

CMachineParameter const paraResonance =
{
        pt_byte,                                        //  type
        "Res./Bandw.",
        "Filter Resonance/Bandwidth",                             // description
        0,                                                        //  Min
        127,                                            //  Max
        0xff,                                           //  NoValue
        MPF_STATE,                                      //  Flags
        32                                                        //  default
};

CMachineParameter const paraPEGAttackTime =
{
        pt_byte,                                        //  type
        "PEG Attack",
        "PEG Attack Time",                              //  description
        0,                                                        //  Min
        127,                                            //  Max
        0xff,                                           //  NoValue
        MPF_STATE,                                      //  Flags
        0                                                         //  default
};


CMachineParameter const paraPEGDecayTime =
{
        pt_byte,                                        //  type
        "PEG Release",
        "PEG Release Time",                             //  description
        0,                                                        //  Min
        127,                                            //  Max
        0xff,                                           //  NoValue
        MPF_STATE,                                      //  Flags
        0                                                         //  default
};


CMachineParameter const paraPEnvMod =
{
        pt_byte,                                        //  type
        "PEnvMod",
        "Pitch Envelope Modulation",                                    // description
        0,                                                             //  Min
        127,                                                  //  Max
        0xff,                                                 //  NoValue
        MPF_STATE,                                            //  Flags
        0x40                                                           //  default
};


CMachineParameter const paraFEGAttackTime =
{
        pt_byte,                                              //  type
        "FEG Attack",
        "FEG Attack Time",                                    //  description
        0,                                                             //  Min
        127,                                                  //  Max
        0xff,                                                 //  NoValue
        MPF_STATE,                                            //  Flags
        0                                                              //  default
};

CMachineParameter const paraFEGSustainTime =
{
        pt_byte,                                              //  type
        "FEG Sustain",
        "FEG Sustain Time",                                   //  description
        0,                                                             //  Min
        127,                                                  //  Max
        0xff,                                                 //  NoValue
        MPF_STATE,                                            //  Flags
        0                                                              //  default
};


CMachineParameter const paraFEGReleaseTime =
{
        pt_byte,                                              //  type
        "FEG Release",
        "FEG Release Time",                                   //  description
        0,                                                             //  Min
        127,                                                  //  Max
        0xff,                                                 //  NoValue
        MPF_STATE,                                            //  Flags
        0                                                              //  default
};


CMachineParameter const paraFEnvMod =
{
        pt_byte,                                              //  type
        "FEnvMod",
        "Filter Envelope Modulation",                                   // description
        0,                                                             //  Min
        127,                                                  //  Max
        0xff,                                                 //  NoValue
        MPF_STATE,                                            //  Flags
        0x40                                                           //  default
};

// LFOs
CMachineParameter const paraLFO1Dest =
{
        pt_byte,                                              //  type
        "LFO1Dest",
        "LFO1 Destination",                                   //  description
        0,                                                             //  Min
        15,                                                   //  Max
        0xff,                                                 //  NoValue
        MPF_STATE,                                            //  Flags
        0                                                              //  default
};

CMachineParameter const paraLFO1Wave =
{
        pt_byte,                                              //  type
        "LFO1Wav",
        "LFO1 Waveform",                                      //  description
        0,                                                             //  Min
        4,                                                    //  Max
        0xff,                                                 //  NoValue
        MPF_STATE,                                            //  Flags
        0                                                              //  default
};

CMachineParameter const paraLFO1Freq =
{
        pt_byte,                                              //  type
        "LFO1Freq",
        "LFO1 Frequency",                                     //  description
        0,                                                             //  Min
        127,                                                  //  Max
        0xff,                                                 //  NoValue
        MPF_STATE,                                            //  Flags
        0                                                              //  default
};

CMachineParameter const paraLFO1Amount =
{
        pt_byte,                                              //  type
        "LFO1Amount",
        "LFO1 Amount",                                        //  description
        0,                                                             //  Min
        127,                                                  //  Max
        0xff,                                                 //  NoValue
        MPF_STATE,                                          //  Flags
        0                                                             //  default
};

// lfo2
CMachineParameter const paraLFO2Dest =
{
        pt_byte,                                            //  type
        "LFO2Dest",
        "LFO2 Destination",                                 //  description
        0,                                                            //  Min
        15,                                                 //  Max
        0xff,                                               //  NoValue
        MPF_STATE,                                          //  Flags
        0                                                             //  default
};

CMachineParameter const paraLFO2Wave =
{
        pt_byte,                                            //  type
        "LFO2Wav",
        "LFO2 Waveform",                                    //  description
        0,                                                            //  Min
        4,                                                  //  Max
        0xff,                                               //  NoValue
        MPF_STATE,                                          //  Flags
        0                                                             //  default
};

CMachineParameter const paraLFO2Freq =
{
        pt_byte,                                            //  type
        "LFO2Freq",
        "LFO2 Frequency",                                   //  description
        0,                                                            //  Min
        127,                                                //  Max
        0xff,                                               //  NoValue
        MPF_STATE,                                          //  Flags
        0                                                             //  default
};

CMachineParameter const paraLFO2Amount =
{
        pt_byte,                                            //  type
        "LFO2Amount",
        "LFO2 Amount",                                      //  description
        0,                                                            //  Min
        127,                                                //  Max
        0xff,                                               //  NoValue
        MPF_STATE,                                          //  Flags
        0                                                             //  default
};



CMachineParameter const *pParameters[] = {
        &paraNote,
        &paraWave1,
        &paraPulseWidth1,
        &paraWave2,
        &paraPulseWidth2,
        &paraDetuneSemi,
        &paraDetuneFine,
        &paraSync,
        &paraMixType,
        &paraMix,
        &paraSubOscWave,
        &paraSubOscVol,
        &paraPEGAttackTime,
        &paraPEGDecayTime,
        &paraPEnvMod,
        &paraGlide,

        &paraVolume,
        &paraAEGAttackTime,
        &paraAEGSustainTime,
        &paraAEGReleaseTime,

        &paraFilterType,
        &paraCutoff,
        &paraResonance,
        &paraFEGAttackTime,
        &paraFEGSustainTime,
        &paraFEGReleaseTime,
        &paraFEnvMod,

        // LFO 1
        &paraLFO1Dest,
        &paraLFO1Wave,
        &paraLFO1Freq,
        &paraLFO1Amount,
        // LFO 2
        &paraLFO2Dest,
        &paraLFO2Wave,
        &paraLFO2Freq,
        &paraLFO2Amount,
};

#pragma pack(1)


class tvals
{
```

```
public:
        byte Note;
        byte Wave1;
        byte PulseWidth1;
        byte Wave2;
        byte PulseWidth2;
        byte DetuneSemi;
        byte DetuneFine;
        byte Sync;
        byte MixType;
        byte Mix ;
        byte SubOscWave;
        byte SubOscVol;
        byte PEGAttackTime;
        byte PEGDecayTime;
        byte PEnvMod;
        byte Glide;

        byte Volume;
        byte AEGAttackTime;
        byte AEGSustainTime;
        byte AEGReleaseTime;

        byte FilterType;
        byte Cutoff;
        byte Resonance;
        byte FEGAttackTime;
        byte FEGSustainTime;
        byte FEGReleaseTime;
        byte FEnvMod;

        byte LFO1Dest;
        byte LFO1Wave;
        byte LFO1Freq;
        byte LFO1Amount;
        byte LFO2Dest;
        byte LFO2Wave;
        byte LFO2Freq;
        byte LFO2Amount;
};


#pragma pack()

CMachineInfo const MacInfo =
{
        MT_GENERATOR,                   //  type
        MI_VERSION,
        0,                                                  //  flags
        1,                                                  //  min tracks
        MAX_TRACKS,                               //  max tracks
        0,                                                  //  numGlobalParameters
        35,                                                 //  numTrackParameters
        pParameters,
        0,
        NULL,
#ifdef _DEBUG
        "M3 by Makk (Debug build)",                    // name
#else
        "M3 by Makk",
#endif
        "M3",                                     //  short name
        "Makk",                                   //  author
        NULL
};


class mi;

class CTrack
{
public:
        void Tick(tvals const &tv);
        void Stop();
        void Init();
        void Work(float *psamples, int numsamples);
        inline float Osc();
        inline float CTrack::Filter( float x);
        inline float VCA();
        void NewPhases();
        int MSToSamples(double const ms);

public:

        // ......Osc......
        byte Note;
        const short *pwavetab1, *pwavetab2, *pwavetabsub;
        int SubOscVol;
        bool noise1, noise2;
        int Bal1, Bal2;
        int MixType;
        int Phase1, Phase2, PhaseSub;
        int Ph1, Ph2;
        float center1, center2;
        float Center1, Center2;
        float PhScale1A, PhScale1B;
        float PhScale2A, PhScale2B;
        int PhaseAdd1, PhaseAdd2;
        float Frequency, FrequencyFrom;
        // Detune
        float DetuneSemi, DetuneFine;
        bool Sync;
        // Glide
        bool Glide, GlideActive;
        float GlideMul, GlideFactor;
        int GlideTime, GlideCount;
        // PitchEnvMod
        bool PitchMod, PitchModActive;
        // PEG ... AD -Hüllkurve
        int PEGState;
        int PEGAttackTime;
        int PEGDecayTime;
        int PEGCount;
        float PitchMul, PitchFactor;
        int PEnvMod;
        // random generator... rauschen
        short r1, r2, r3, r4;

        float OldOut; // gegen extreme Knackser/Wertesprünge

        // .........AEG........ ASR -Hüllkurve
        float Volume;
        int AEGState;
        int AEGAttackTime;
        int AEGSustainTime;
        int AEGReleaseTime;
        int AEGCount;
        float Amp;
        float AmpAdd;

        // ........Filter..........
        float *coefsTabOffs; // abhängig vom FilterTyp
        int Cutoff, Resonance;
        float x1, x2, y1, y 2;
        // FEG ... ASR -Hüllkurve
        int FEnvMod;
        int FEGState;
        int FEGAttackTime;
        int FEGSustainTime;
        int FEGReleaseTime;
        int FEGCount;
        float Cut;
        float CutAdd;

        // .........LFOs...... .....
        // 1
        bool LFO_Osc1;
        bool LFO_PW1;
        bool LFO_Amp;
        bool LFO_Cut;
        // 2
        bool LFO_Osc2;
        bool LFO_PW2;
        bool LFO_Mix;
        bool LFO_Reso;

        bool LFO1Noise, LFO2Noise; // andere  Frequenz
        bool LFO1Synced,LFO2Synced; // zum Songtempo
        const short *pwavetabLFO1, *pwavetabLFO2;
        int LFO1Amount, LFO2Amount;
        int PhaseLFO1, PhaseLFO2, PhaseAddLFO1, PhaseAddLFO2;
        int LFO1Freq, LFO2Freq;



        // RANDOMS
        const short *pnoise;
        int noisePhase;
        bool RandomMixType;
        bool RandomWave1;
        bool RandomWave2;
        bool RandomWaveSub;

        mi *pmi; // ptr to MachineInterface

};


class mi : public CMachineInterface
{
public:
        mi();
        virtual ~mi();

        virtual void Init(CMachineDataInput * const pi);
        virtual void Tick();
        virtual bool Work(float *psamples, int numsamples, int const mode);
        virtual void SetNumTracks(int const n ) { numTracks = n; }
        virtual void Stop();
        virtual char const *mi::DescribeValue(int const param, int const value);
        void ComputeCoefs( float *coefs, int f, int r, int t);
        // skalefuncs
        inline float LFOFreq( int v);
        inline float EnvTime( int v);
        inline float Cutoff( int v);
        inline float Resonance( float v);
        inline float Bandwidth( int v);

public:
        float TabSizeDivSampleFreq;
        int numTracks;
        CTrack Tracks[MAX_TRACKS] ;
        tvals tval[MAX_TRACKS];
};


DLL_EXPORTS
```

```
// Skalierungsmethoden
inline float mi::Cutoff( int v)
{
        return pow( (v+5)/(127.0+5), 1.7)*13000+30;
}
inline float mi::Resonance( float v)
{
        return pow( v/127.0, 4)*150+0.1;
}
inline float mi::Bandwidth( int v)
{
        return pow( v/127.0, 4)*4+0.1;
}

inline float mi::LFOFreq( int v)
{
        return (pow( (v+8)/(116.0+8), 4) -0.000017324998565270)*40.00072;
}

inline float mi::EnvTime( int v)
{
        return pow( (v+2)/(127.0+2), 3)*10 000;
}

//////////////////////////////////////////////////////
// CTRACK METHODEN
//////////////////////////////////////////////////////

inline int CTrack::MSToSamples(double const ms)
{
        return (int)(pmi ->pMasterInfo ->SamplesPerSec * ms * (1.0 / 1 000.0)) + 1; // +1 wg. div durch 0
}


void CTrack::Stop()
{
        AEGState = EGS_NONE;
}

void CTrack::Init()
{
        noise1 = noise2 = Sync = false;
        RandomMixType = false;
        RandomWave1 = false;
        RandomWave2 = false;
        RandomWaveSub = false;
        LFO1Noise = LFO2Noise = false;
        LFO1Synced = LFO2Synced = false;
        AEGState = EGS_NONE;
                    FEGState = EGS_NONE;
                    PEGState = EGS_NONE;
        r1=26474; r2=13075; r3=18376; r4=31291; // randomGenerator
        noisePhase = Phase1 = Phase2 = PhaseSub = PhaseLFO1 = PhaseLFO2 = 0; // Osc starten neu
        x1 = x2 = y1 = x2 = 0;
        pnoise = pmi ->pCB->GetOscillatorTable( OWF_NOISE);
        OldOut = 0;
        pwavetab1=pwavetab2=pwavetabsub=pwavetabLFO1=pwavetabLFO2  =
            pmi ->pCB->GetOscillatorTable( OWF_SINE);

                    coefsTabOffs = coefsTab; // lp
        Cutoff = paraCutoff.DefValue;
        Resonance = paraResonance.DefValue;
        FEGAttackTime = MSToSamples( pmi ->EnvTime( paraFEGAttackTime.DefValue));
        FEGSustainTime = MSToSamples( pmi ->EnvTime( paraFEGSustainTime.DefValue));
        FEGReleaseTime = MSToSamples( pmi ->EnvTime( paraFEGReleaseTime.DefValue));
        AEGAttackTime = MSToSamples( pmi ->EnvTime( paraAEGAttackTime.DefValue));
        AEGSustainTime = MSToSamples( pmi ->EnvTime( paraAEGSustainTime.DefValue));
        AEGReleaseTime = MSToSamples( pmi ->EnvTime( paraAEGReleaseTime.DefValue));
                    FEnvMod = 0;
        PEGAttackTime = MSToSamples( pmi ->EnvTime( paraPEGAttackTime.DefValue));
        PEGDecayTime = MSToSamples( pmi ->EnvTime( paraPEGDecayTime.DefValue));
                    PEnvMod = 0;
        Bal1 = 127 -paraMix.DefValue;
        Bal2 = paraMix.DefValue;
                    Glide =  GlideActive = false;
                    RandomWave1 = RandomWave2 = RandomWaveSub = false;
                    SubOscVol = paraSubOscVol.DefValue;
        Center1 = paraPulseWidth1.DefValue/127.0;
        Center2 = paraPulseWidth2.DefValue/127.0;
                    DetuneSemi = DetuneFine = 1;
                    Volume = (float)(paraVolume.DefValue/245.0);
                    LFO_Osc1 = LFO_PW1 = LFO_Amp = LFO_Cut = false;
                    LFO_Osc2 = LFO_PW2 = LFO_Mix = LFO_Reso = false;
                    PhaseAddLFO1 = PhaseAddLFO2 = 0;
                    MixType = 0;
}

void CTrack::Tick( tvals const &tv)
{

        // Filter
        if( tv.FilterType != paraFilterType.NoValue)
                coefsTabOffs = coefsTab + ( int)tv.FilterType*128*128*8;
        if( tv.Cutoff != paraCutoff.NoValue)
                Cutoff = tv.Cutoff;
        if( tv.Resonance != paraResonance.NoValue)
                Resonance = tv.Resonance;
        // FEG
        if( tv.FEGAttackTime != paraFE GAttackTime.NoValue)
                FEGAttackTime = MSToSamples( pmi ->EnvTime( tv.FEGAttackTime));
        if( tv.FEGSustainTime != paraFEGSustainTime.NoValue)
                FEGSustainTime = MSToSamples( pmi ->EnvTime( tv.FEGSustainTime));
        if( tv .FEGReleaseTime != paraFEGReleaseTime.NoValue)
                FEGReleaseTime = MSToSamples( pmi ->EnvTime( tv.FEGReleaseTime));
```

```
if( tv.FEnvMod != paraFEnvMod.NoValue)
        FEnvMod = (tv.FEnvMod  - 0x40)<<1;




// PEG
if( tv.PEGAttackTime != paraPEGAttackTime.NoValue)
        PEGAttackTime = MSToSamples( pmi ->EnvTime( tv.PEGAttackTime));
if( tv.PEGDecayTime != paraPEGDecayTime.NoValue)
        PEGDecayTime = MSToSamples( pmi ->EnvTime( tv.PEGDecayTim e));
if( tv.PEnvMod != paraPEnvMod.NoValue) {
        if( tv.PEnvMod  - 0x40 != 0)
                PitchMod = true;
        else {
                PitchMod = false;
                PitchModActive = false;
        }
        PEnvMod = tv.PEnvMod  - 0x40;
}

if( tv.Mix != paraMix.NoValue) {
        Bal1 = 127 -tv.Mix;
        Bal2 = tv.Mix;
}

if( tv.Glide != paraGlide.NoValue)
        if( tv.Gl ide == 0) {
                Glide = false;
                GlideActive = false;
        }
        else {
                Glide = true;
                GlideTime = tv.Glide*10000000/pmi ->pMasterInfo->SamplesPe rSec;
        }


// SubOsc
if( tv.SubOscWave != paraSubOscWave.NoValue)
        if( tv.SubOscWave == 4) // random
                RandomWaveSub = true;
        else {
                pwavetabsub = pm i->pCB->GetOscillatorTable( tv.SubOscWave);
                RandomWaveSub = false;
        }

if( tv.SubOscVol != paraSubOscVol.NoValue)
        SubOscVol = tv.SubOscVol;

// PW
if( tv.PulseWidth1 != paraPuls eWidth1.NoValue) {
        Center1 = tv.PulseWidth1/127.0;
}
if( tv.PulseWidth2 != paraPulseWidth2.NoValue) {
        Center2 = tv.PulseWidth2/127.0;
}

// Detune
if( tv.DetuneSemi != paraDetuneSemi.N oValue)
        DetuneSemi = (float)pow( 1.05946309435929526, tv.DetuneSemi  -0x40);
if( tv.DetuneFine != paraDetuneFine.NoValue)
        DetuneFine = (float)pow( 1.00091728179580156, tv.DetuneFine  -0x40);
if( tv.Sync != SWITCH _NO)
        if( tv.Sync == SWITCH_ON)
                Sync = true;
        else
                Sync = false;


if( tv.MixType != paraMixType.NoValue)
        if( tv.MixType == 8) // random
                 RandomMixType = true;
        else {
                MixType = tv.MixType;
                RandomMixType = false;
        }

if( tv.Wave1 != paraWave1.NoValue) { // neuer wert
        if( tv.Wave1 == O WF_NOISE)
                noise1 = true;
        else
                noise1 = false;
        if( tv.Wave1 == 5) // random
                RandomWave1 = true;
        else {
                RandomWave 1 = false;
                pwavetab1 = pmi ->pCB->GetOscillatorTable( tv.Wave1);
        }
}

if( tv.Wave2 != paraWave2.NoValue) { // neuer wert
        if( tv.Wave2 == OWF_NOISE)
                noise2 = true ;
        else
                noise2 = false;
        if( tv.Wave2 == 5) // random
                RandomWave2 = true;
        else {
                RandomWave2 = false;
                pwavetab2 =  pmi->pCB->GetOscillatorTable( tv.Wave2);
        }
}
```

```
        if( tv.AEGAttackTime != paraAEGAttackTime.NoValue)
                AEGAttackTime = MSToSamples( pmi ->EnvTime( tv.AEGAttackTime));
        if( tv.AEGSustainTime != paraAEGSustainT ime.NoValue)
                AEGSustainTime = MSToSamples( pmi ->EnvTime( tv.AEGSustainTime));
        if( tv.AEGReleaseTime != paraAEGReleaseTime.NoValue)
                AEGReleaseTime = MSToSamples( pmi ->EnvTime( tv.AEGReleaseTime));
        if( tv.Volum e != paraVolume.NoValue)
                Volume = (float)(tv.Volume/245.0);

        if( tv.Note != paraNote.NoValue) { // neuer wert
                Note = tv.Note;
                if( (Note >= NOTE_MIN) && (Note <= NOTE_MAX)) { // neue note gesetzt
                        FrequencyFrom = Frequency;
                        Frequency = freqTab[Note];


                        // RANDOMS
                        if( RandomMixType) {
                                MixType = (unsigned)pnoise[noisePhase++] % 8;
                                noisePhase &= 0x7ff;
                        }
                        if( RandomWaveSub) {
                                pwavetabsub = pmi ->pCB->GetOscillatorTable( (unsigned)pnoise[noisePhase++] % 4);
                                noisePhase &= 0x7ff;
                        }
                        if( RandomWave1) {
                                pwavetab1 = pmi ->pCB->GetOscillatorTable( (unsigned)pnoise[noisePhase++] % 4);
                                noisePhas e &= 0x7ff;
                        }
                        if( RandomWave2) {
                                pwavetab2 = pmi ->pCB->GetOscillatorTable( (unsigned)pnoise[noisePhase++] % 4);
                                noisePhase &= 0x7ff;
                        }

                        if( Glide) {
                                GlideActive = true;
                                if( Frequency > FrequencyFrom)
                                        GlideMul = pow( 2, 1.0/GlideTime);
                                else
                                        GlideMul = pow( 0.5, 1.0/GlideTime);
                                GlideFactor = 1;
                                GlideCount = (int)(log( Frequency/FrequencyFrom)/log(GlideMul));
                        }
                        else
                                GlideActive = false;

                        // trigger envelopes neu an...
                        // Amp
                        AEGState = EGS_ATTACK;
                        AEGCount = AEGAttackTime;
                        AmpAdd = Volume/AEGAttackTime;
                        Amp = 0; //AmpAdd; // fange bei 0 an
                        // Pitch
                        if( PitchMod) {
                                PitchMod Active = true;
                                PEGState = EGS_ATTACK;
                                PEGCount = PEGAttackTime;
                                PitchMul = pow( pow( 1.01, PEnvMod), 1.0/PEGAttackTime);
                                PitchFa ctor = 1.0;
                        }
                        else
                                PitchModActive = false;

                        // Filter
                        FEGState = EGS_ATTACK;
                        FEGCount = FEGAttackTime;
                        CutAdd = ((float)FEnvMod)/FEGAttackTime;
                        Cut = 0.0; // fange bei 0 an


                } else
                        if( tv.Note == NOTE_OFF)
                                AEGState = EGS_NONE; // note aus

        }

        // ..........LFO...........

        // LFO1
        if( tv.LFO1Dest != paraLFO1Dest.NoValue)
        {
                LFO_Osc1 = LFO_PW1 = LFO_Amp = LFO_Cut = false;
                switch( tv.LFO1Dest) {
//              case 0: ...n one
                case 1:
                        LFO_Osc1 = true;
                        break;
                case 2:
                        LFO_PW1 = true;
                        break;
                case 3:
                        LFO_Amp = tru e;
                        break;
                case 4:
                        LFO_Cut = true;
                        break;

                case 5: // 12
                        LFO_Osc1 = true;
                        LFO_PW1 = true;
                        break;
                case 6: // 13
                        LFO_Osc1 = true;
                        LFO_Amp = true;
                        break;
                case 7: // 14
                        LFO_Osc1 = true;
                        LFO _Cut = true;
                        break;
                case 8: // 23
                        LFO_PW1 = true;
                        LFO_Amp = true;
                        break;
                case 9: // 24
                        LFO_PW1 = true;
                        LFO_Cut = true;
                        break;
                case 10: // 34
                        LFO_Amp = true;
                        LFO_Cut = true;
                        break;

                case 11: // 123
                         LFO_Osc1 = true;
                        LFO_PW1 = true;
                        LFO_Amp = true;
                        break;
                case 12: // 124
                        LFO_Osc1 = true;
                        LFO_PW1 = true;
                        LFO_Cut = true;
                        break;
                case 13: // 134
                        LFO_Osc1 = true;
                        LFO_Amp = true;
                        LFO_Cut = true;
                        break;
                case 14: // 234
                        LFO_PW1 = true;
                        LFO_Amp = true;
                        LFO_Cut = true;
                        break;
                case 15: // 1234
                        LFO_Osc1 = true;
                        LFO_PW1 = true;
                        LFO_Amp = true;
                        LFO_Cut = true;
                        break;
                }
        }
        if( tv.LFO1Wave != paraLFO1Wave.NoValue) {
                pwavetabLFO1  = pmi->pCB->GetOscillatorTable( tv.LFO1Wave);
                if( tv.LFO1Wave == OWF_NOISE)
                        LFO1Noise = true;
                else
                        LFO1Noise = false;
        }

        if( tv.LFO1Freq != paraLFO1Freq.NoValu e)
                if( tv.LFO1Freq>116) {
                        LFO1Synced = true;
                        LFO1Freq = tv.LFO1Freq  - 117;
                }
                else {
                        LFO1Synced = false;
                        LFO1Freq  = tv.LFO1Freq;
                }
                if( tv.LFO1Amount != paraLFO1Amount.NoValue)
                LFO1Amount = tv.LFO1Amount;

        if( LFO1Synced)
                if( LFO1Noise) // sample & hold
                        PhaseAddLFO1 = (int) (0x200000/(pmi ->pMasterInfo ->SamplesPerTick<<LFO1Freq));
                else
                        PhaseAddLFO1 = (int)((double)0x200000*2048/(pmi  ->pMasterInfo ->SamplesPerTick<<LFO1Freq));
        else
                if( LFO1Noise) // sample & hold
                        PhaseAddLFO1 = (int)(pmi ->LFOFreq( LFO1Freq)/pmi ->pMasterInfo ->SamplesPerSec*0x200000);
                else
                        PhaseAddLFO1 = (int)(pmi ->LFOFreq( LFO1Freq)*pmi ->TabSizeDivSampleFreq*0x200000);

        // LFO2
        if( tv.LFO2Dest != paraLFO2Dest.NoValue)
        {
                LFO_Osc2 = LFO_PW2 = LFO_Mix = LFO_Reso = false;
                switch( tv.LFO2Dest) {
//              case 0: ...none
                case 1:
                        LFO_Osc2 = true;
                        break;
                case 2:
                        LFO_PW2 = true;
                        break;
                case 3:
                        LFO_Mix = true;
                        break;
                case 4:
                         LFO_Reso = true;
                        break;
                case 5: // 12
                        LFO_Osc2 = true;
                        LFO_PW2 = true;
                        break;
                case 6: // 13
                         LFO_Osc2 = true;
                        LFO_Mix = true;
                        break;
                case 7: // 14
```

```
                        LFO_Osc2 = true;
                        LFO_Reso = true;
                        break;
                case 8:  // 23
                        LFO_PW2 = true;
                        LFO_Mix = true;
                        break;
                case 9: // 24
                        LFO_PW2 = true;
                        LFO_Reso = true;
                        bre ak;
                case 10: // 34
                        LFO_Mix = true;
                        LFO_Reso = true;
                        break;
                case 11: // 123
                        LFO_Osc2 = true;
                        LFO_PW2 = tr ue;
                        LFO_Mix = true;
                        break;
                case 12: // 124
                        LFO_Osc2 = true;
                        LFO_PW2 = true;
                        LFO_Reso = true;
                        bre ak;
                case 13: // 134
                        LFO_Osc2 = true;
                        LFO_Mix = true;
                        LFO_Reso = true;
                        break;
                case 14: // 234
                        LFO_PW2 = t rue;
                        LFO_Mix = true;
                        LFO_Reso = true;
                        break;
                case 15: // 1234
                        LFO_Osc2 = true;
                        LFO_PW2 = true;
                        L FO_Mix = true;
                        LFO_Reso = true;
                        break;
                }
        }
        if( tv.LFO2Wave != paraLFO2Wave.NoValue) {
                pwavetabLFO2 = pmi ->pCB->GetOscillatorTable( tv.LFO2Wave);
                 if( tv.LFO2Wave == OWF_NOISE)
                        LFO2Noise = true;
                else
                        LFO2Noise = false;
        }

        if( tv.LFO2Freq != paraLFO2Freq.NoValue)
                if( tv.LFO2Freq>116) {
                         LFO2Synced = true;
                        LFO2Freq = tv.LFO2Freq  - 117;
                }
                else {
                        LFO2Synced = false;
                        LFO2Freq = tv.LFO2Freq;
                }

        if( tv.LFO2Amount ! = paraLFO2Amount.NoValue)
                LFO2Amount = tv.LFO2Amount;

        if( LFO2Synced)
                if( LFO2Noise) // sample & hold
                        PhaseAddLFO2 = (int)(0x200000/(pmi  ->pMasterInfo->SamplesPerTick<<LFO2Freq));
                 else
                        PhaseAddLFO2 = (int)((double)0x200000*2048/(pmi  ->pMasterInfo->SamplesPerTick<<LFO2Freq));
        else
                if( LFO2Noise) // sample & hold
                        PhaseAddLFO2 = (int)(pmi ->LFOFreq( LFO2Freq)/pmi ->pMasterInfo->SamplesPerSec*0x200000);
                else
                        PhaseAddLFO2 = (int)(pmi ->LFOFreq( LFO2Freq)*pmi ->TabSizeDivSampleFreq*0x200000);


        if( GlideActive) {
                PhaseAdd1 = (int)(FrequencyFrom*pmi  ->TabSizeDivSampleFreq*0x10000);
                PhaseAdd2 = (int)(FrequencyFrom*DetuneSemi*DetuneFine*pmi  ->TabSizeDivSampleFreq*0x10000);
        }
        else {
                PhaseAdd1 = (int)(Frequency*pmi ->TabSizeDivSampleFreq*0x10000);
                Phase Add2 = (int)(Frequency*DetuneSemi*DetuneFine*pmi ->TabSizeDivSampleFreq*0x10000);
        }

}


inline float CTrack::Osc()
{
        float o, o2;
        int B1, B2;
        if( LFO_Mix) {
                B2 = Bal2 + ((pwavetabLFO2[((unsigned)PhaseLFO2)>>2  1]*LFO2Amount)>>15);
                if( B2<0)
                        B2 = 0;
                else
                        if( B2>127)
                                B2 = 127;
                B1 = 127 -B2;
        }
        else {
                B1 = Bal 1;
                B2 = Bal2;
        }

        // osc1
        if( noise1) {
                short t = r1+r2+r3+r4;
                r1=r2; r2=r3; r3=r4; r4=t;
                o = (float)((t*B1)>>7);
        }
        else
                o = (float)((pwa vetab1[Ph1>>16]*B1)>>7);

        // osc2
        if( noise2) {
                short u = r1+r2+r3+r4;
                r1=r2; r2=r3; r3=r4; r4=u;
                o2 = (u*B2)>>7;
        }
        else
                o2 = (pwavetab2[Ph2>>16]*B2)>>7;


        switch( MixType)
        {
        case 0: //ADD
                o += o2;
                break;
        case 1: // ABS
                o = fabs(o -o2)*2-0x8000;
                break;
        case 2: // MUL
                o *= o2*(1.0/0x4000);
                break;
        case 3: // highest amp
                if( fabs(o) < fabs(o2))
                        o = o2;
                break;
        case 4: // lowest amp
                if( fabs(o) > fabs(o2))
                        o = o2;
                break;
        case 5: // AND
                o = (int)o & (int)o2;
                break;
        case 6: // OR
                o = (int)o | (int)o2;
                break;
        case 7: // XOR
                o = (int)o ^ (int)o2;
                b reak;
        }
        return o + ((pwavetabsub[PhaseSub>>16]*SubOscVol)>>7);
}

inline float CTrack::VCA()
{
        // EG...
        if( !AEGCount --)
                switch( ++AEGState)
                {
                case EGS_SUSTAIN:
                        AEGCount = AEGSustainTime;
                        Amp = Volume;
                        AmpAdd = 0.0;
                        break;
                case EGS_RELEASE:
                        AEGCount = AEGReleaseTime;
                        AmpA dd = -Volume/AEGReleaseTime;
                        break;
                case EGS_RELEASE + 1:
                        AEGState = EGS_NONE;
                        AEGCount =  -1;
                        Amp = 0.0;
                        break;
                }

        Amp +=AmpAdd;
        if( LFO_Amp) {
                float a =
                  Amp + (pwavetabLFO1[((unsigned)PhaseLFO1)>>21]*LFO1Amount)/(127.0*0x8000);
                if( a<0)
                        a = 0;
                return( a );
        }
        else
                return Amp;
}


inline float CTrack::Filter( float x)
{
        float y;

        // Envelope
        if( FEGState) {
                if( !FEGCount --)
                        switch( ++FEGState)
                        {
                        case EGS_SUSTAIN:
                                FEGCount = FEGSustainTime;
                                Cut = (float)FEnvMod;
                                CutAdd = 0.0;
                                break;
                        case EGS_RELEASE:
                                FEGCount = FEGReleaseTime;
                                CutAdd = ((float) -FEnvMod)/FEGReleaseTime;
                                break;
                        case EGS_RELEASE + 1:
                                FEGState = EGS_NONE; // false
                                FEGCount =  -1;
                                Cut = 0.0;
                                CutAdd = 0.0;
                                break;
                         }
                Cut += CutAdd;
        }

        // LFO
        // Cut
        int c, r;
        if( LFO_Cut)
                c = Cutoff + Cut + // Cut = EnvMod
                ((pwavetabLFO1[((unsigned)PhaseLFO1)>>21]*LFO1Amount)>>(7+8));
        else
                c = Cutoff + Cut; // Cut = EnvMod
        if( c < 0)
                c = 0;
        else
                if( c > 127)
                        c = 127;
        // Reso
        if( LFO_Reso) {
                r = Resonance +
                 ((pwavetabLFO2[((unsigned)PhaseLFO2)>>21]*LFO2Amount)>>(7+8));
        if( r < 0)
                r = 0;
        else
                if( r > 127)
                        r = 127;
        }
        else
                r = Resonance;


        int ofs = (( c<<7)+r)<<3;
        y = coefsTabOffs[ofs]*x +
                coefsTabOffs[ofs+1]*x1 +
                coefsTabOffs[ofs+2]*x2 +
                coefsTabOffs[ofs+3]*y1 +
                coefsTabOffs[ofs+4]*y2;

        y2=y1;
        y1=y;
        x2=x1;
        x1=x;
        return y;
}

inline void CTrack::NewPhases()
{
        if( PitchModActive) {
                if( GlideActive) {
                        if( LFO_Osc1) {
                                float pf = LFOOscTab[(pwavetabLFO1[((unsigned)Phase  LFO1)>>21]*LFO1Amount>>7) + 0x8000];
                                Phase1 += PhaseAdd1*GlideFactor*PitchFactor*pf;
                                PhaseSub += PhaseAdd1*GlideFactor*PitchFactor*pf/2;
                        }
                        else  {
                                Phase1 += PhaseAdd1*GlideFactor*PitchFactor;
                                PhaseSub += PhaseAdd1*GlideFactor*PitchFactor/2;
                        }
                        if( LFO_Osc2)
                                 Phase2 += PhaseAdd2*GlideFactor*PitchFactor
                                   *LFOOscTab[(pwavetabLFO2[((unsigned)PhaseLFO2)>>21]*LFO2Amount>>7) + 0x8000];
                        else
                                Phase2 += PhaseAdd2*GlideFactor*PitchFa  ctor;
                        GlideFactor *= GlideMul;
                        if( !GlideCount --) {
                                GlideActive = false;
                                PhaseAdd1 = (int)(Frequency*pmi  ->TabSizeDivSampleFreq*0x10000);
                                PhaseAdd2 = (int)(Frequency*DetuneSemi*DetuneFine*pmi  ->TabSizeDivSampleFreq*0x10000);
                        }
                }
                else { // kein Glide
                        if( LFO_Osc1) {
                                 float pf = LFOOscTab[(pwavetabLFO1[((unsigned)PhaseLFO1)>>21]*LFO1Amount>>7) + 0x8000];
                                Phase1 += PhaseAdd1*PitchFactor*pf;
                                PhaseSub += PhaseAdd1*PitchFactor*pf/2;
                         }
                        else {
                                Phase1 += PhaseAdd1*PitchFactor;
                                PhaseSub += PhaseAdd1*PitchFactor/2;
                        }
                        if( LFO_Osc2)
                                 Phase2 += PhaseAdd2*PitchFactor
                                   *LFOOscTab[(pwavetabLFO2[((unsigned)PhaseLFO2)>>21]*LFO2Amount>>7) + 0x8000];
                        else
                                Phase2 += PhaseAdd2*PitchFactor;
                }

                PitchFactor *= PitchMul;

                if( !PEGCount --)
                        if( ++PEGState == 2) {// DECAY -PHASE beginnt
                                PEGCount = PEGDecayTime;
                                PitchMul =  pow( pow( 1/1.01, PEnvMod), 1.0/PEGDecayTime);
                        }
                        else  // AD -Kurve ist zu Ende
                                PitchModActive = false;
        }

        else { // kein PitchMod
                if( GlideActiv e) {
                        if( LFO_Osc1) {
                                float pf = LFOOscTab[(pwavetabLFO1[((unsigned)PhaseLFO1)>>21]*LFO1Amount>>7) + 0x8000];
                                Phase1 += PhaseAdd1*GlideFactor*pf;
                                 PhaseSub += PhaseAdd1*GlideFactor*pf/2;
                        }
                        else {
                                Phase1 += PhaseAdd1*GlideFactor;
                                PhaseSub += PhaseAdd1*GlideFactor/2;
                         }
                        if( LFO_Osc2)
                                Phase2 += PhaseAdd2*GlideFactor
                                  *LFOOscTab[(pwavetabLFO2[((unsigned)PhaseLFO2)>>21]*LFO2Amount>>7) + 0x8000];
                        else
                                Phase2 += PhaseAdd2*GlideFactor;
                        GlideFactor *= GlideMul;
                        if( !GlideCount --) {
                                GlideActive = false;
                                PhaseAdd1 = (in t)(Frequency*pmi ->TabSizeDivSampleFreq*0x10000);
                                PhaseAdd2 = (int)(Frequency*DetuneSemi*DetuneFine*pmi  ->TabSizeDivSampleFreq*0x10000);
                        }
                }
                else {
                         if( LFO_Osc1) {
                                float pf = LFOOscTab[(pwavetabLFO1[((unsigned)PhaseLFO1)>>21]*LFO1Amount>>7) + 0x8000];
                                Phase1 += PhaseAdd1*pf;
                                PhaseSub += PhaseAdd1*pf/2;
                        }
                        else {
                                Phase1 += PhaseAdd1;
                                PhaseSub += PhaseAdd1/2;
                        }
                        if( LFO_Osc2)
                                 Phase2 += PhaseAdd2
                                  *LFOOscTab[(pwavetabLFO2[((unsigned)PhaseLFO2)>>21]*LFO2Amount>>7) + 0x8000];
                        else
                                Phase2 += PhaseAdd2;
                }
        }


        if( Phase1 & 0xf8000000) { // neuer durchlauf ??
                // PW1
                if( LFO_PW1) { //LFO_PW_Mod
                        center1 = Center1 + (float)pwavetabLFO1[((unsigned)PhaseLFO1)>>21]*
                                                 LFO1Amount/(127.0*0x8000);
                        if( center1 < 0)
                                center1 = 0;
                        else
                                if( center1 > 1)
                                        center1 = 1;
                }
                else  // No LFO
                        center1 = Center1;
                PhScale1A = 0.5/center1;
                PhScale1B = 0.5/(1 -center1);
                center1 *= 0x8000000;
                // PW2
                if( LFO_PW2 ) { //LFO_PW_Mod
                        center2 = Center2 + (float)pwavetabLFO2[((unsigned)PhaseLFO2)>>21]*
                                                LFO2Amount/(127.0*0x8000);
                        if( center2 < 0)
                                 center2 = 0;
                        else
                                if( center2 > 1)
                                        center2 = 1;
                }
                else  // No LFO
                        center2 = Center2;
                Ph Scale2A = 0.5/center2;
                PhScale2B = 0.5/(1 -center2);
                center2 *= 0x8000000;

                // SYNC
                if( Sync)
                        Phase2 = Phase1; // !!!!!
        }

        Phase1 &= 0x7ffffff;
        P hase2 &= 0x7ffffff;
        PhaseSub &= 0x7ffffff;

        if( Phase1 < center1)
                Ph1 = Phase1*PhScale1A;
        else
                Ph1 = (Phase1  - center1)*PhScale1B + 0x4000000;

        if( Phase2 < center2)
                Ph2 = Phas e2*PhScale2A;
        else
                Ph2 = (Phase2  - center2)*PhScale2B + 0x4000000;

                    // LFOs
        PhaseLFO1 += PhaseAddLFO1;
        PhaseLFO2 += PhaseAddLFO2;
}

void CTrack::Work( float *psamples, int numsamples)
{
        for( int i=0; i<num samples; i++) {
                if( AEGState) {
                        float o = Osc()*VCA();
                        *psamples++ = Filter( OldOut + o); // anti knack
                        OldOut = o;
                }
                else
                        *psamples++ = 0;
                NewPhases();
        }
}

//......................................................
//.................... DESCRIPTION .....................
//......................................................

char const *mi::DescribeValue(int const param, int const value)
{
        static char txt[16];

        switch(param){
        case 2: // PW1
        case 4: // PW2
                sprintf(txt, "%u : %u", (int)(value*100.0/127),
                                                        100 -(int)(value*100.0/127));
                break;
        case 5: // semi detune
                if( value == 0x40)
                        return "±0 halfnotes";
                else
                        if( value > 0x40)
                                sprintf( txt, "+%i halfnotes", value  -0x40);
                        else
                                sprintf( txt, "%i halfnotes", value  -0x40);
                break;
        case 6: // fine detune
                if( valu e == 0x40)
                        return "±0 cents";
                else
                        if( value > 0x40)
                                sprintf( txt, "+%i cents", (int)((value  -0x40)*100.0/63));
                        else
                                sprintf( txt, "%i cents", (int)((value -0x40)*100.0/63));
                break;

        case 7: // Sync
                if( value == SWITCH_ON)
                        return( "on");
                else
                        return( "off");
                break;

        case 8: // MixType
                switch( value) {
                case 0: return( "add");
                case 1: return( "difference");
                case 2: return( "mul");
                case 3: return( "highest amp ");
                case 4: return( "lowest amp");
                case 5: return( "and");
                case 6: return( "or");
                case 7: return( "xor");
                case 8: return( "random");
                break;
                }
        case 9: // Mix
                switch( value) {
                case 0:return "Osc1";
                case 127:return "Osc2";
                default: sprintf(txt, "%u%% : %u%%", 100 -(int)(value*100.0/127),
                                                        (int)(value*100.0/127));
                }
                break;

        case 12: // Pitch Env
        case 13: // Pitch Env
        case 17: // Amp Env
        case 18: // Amp Env
        case 19: // Amp Env
        case 23: // Filt er Env
        case 24: // Filter Env
        case 25: // Filter Env
                sprintf( txt, "%.4f sec", EnvTime( value)/1000);
                break;

        case 14: // PitchEnvMod
        case 26: // Filt ENvMod
                sprintf( txt, "%i" , value-0x40);
                break;
        case 20:
                switch( value) {
                case 0: return( "lowpass");
                case 1: return( "highpass");
                case 2: return( "bandpass");
                case 3: return( "b andreject");
                }
                break;
        case 27: // LFO1Dest
                switch( value) {
                case 0: return "none";
                case 1: return "osc1";
                case 2: return "p.width1";
                cas e 3: return "volume";
                case 4: return "cutoff";
                case 5: return "osc1+pw1"; // 12
                case 6: return "osc1+volume"; // 13
                case 7: return "osc1+cutoff"; // 14
                case 8: return "pw1+volu me"; // 23
                case 9: return "pw1+cutoff"; // 24
                case 10: return "vol+cutoff"; // 34
                case 11: return "o1+pw1+vol";// 123
                case 12: return "o1+pw1+cut";// 124
                case 13: return "o1+vo l+cut";// 134
                case 14: return "pw1+vol+cut";// 234
                case 15: return "all";// 1234
                }
                break;
        case 31: // LFO2Dest
                switch( value) {
                case 0: return "none";
                case 1: return "osc2";
                case 2: return "p.width2";
                case 3: return "mix";
                case 4: return "resonance";

                case 5: return "osc2+pw2"; // 12
                case 6: return "osc2+mix";  // 13
                case 7: return "osc2+res"; // 14
                case 8: return "pw2+mix"; // 23
                case 9: return "pw2+res"; // 24
                case 10: return "mix+res"; // 34

                case 11: return "o2+pw2+mix"; // 123
                case 12: return "o2+pw2+res"; // 124
                case 13: return "o2+mix+res"; // 134
                case 14: return "pw2+mix+res"; // 234
                case 15: return "all"; // 1234
                }
                break;
        case 10: // SubOscWave
                if( value == 4)
                        return( "random");
        case 1: // OSC1Wave
        case 3: // OSC2Wave
        case 28: // LFO1Wave
        case 32: // LFO2Wave
                switch( value) {
                case 0: return "sine";
                case 1: return "saw";
                case 2: return "square";
                case 3: return "triangle";
                case 4: return "noise";
                case 5: return "random";
                }
                break;
        case 29: // LFO1Freq
        case 33: // LFO2Freq
                if( value <= 116)
                        sprintf( txt, "%.4f HZ", LFOFreq( value));
                else
                        sprintf( txt, "%u ticks", 1<<(value -117));
                break;
        default: return NULL;
        }
        return txt;
}



////////////////////////////////////////////////////////
// MACHINE INTERFACE METHODEN
////////////////////////////////////////////////////////

mi::mi()
{
        TrackVals = tval;
        GlobalVals = NULL;
        AttrVals = NULL;   // attributes
}

mi::~mi()
{
}


void mi::Init(CMachineDataInput * const pi)
{
        TabSizeDivSampleFreq = (float)(2048.0/pMasterInfo ->SamplesPerSec);

        for( int i=0; i<MAX_TRAC KS; i++)
        {
                Tracks[i].pmi = this;
                Tracks[i].Init();
        }

        // generate frequencyTab
        double freq = 16.35; //c0 bis b9
        for( int j=0; j<9; j++)
        {
                for( int i=1; i<=12; i ++)
                {
                        freqTab[j*16+i] = (float)freq;
                        freq *= 1.05946309435929526; // *2^(1/12)
                }
        }
        // generate coefsTab
        for( int t=0; t<4; t++)
                for( int  f=0; f<128; f++)
                        for( int r=0; r<128; r++)
                                ComputeCoefs( coefsTab+(t*128*128+f*128+r)*8, f, r, t);
        // generate LFOOscTab
        for( int p=0; p<0x10000; p++)
                LFOOscTab[p] = p ow( 1.00004230724139582, p -0x8000);
}


void mi::Tick()
{
        for (int i=0; i<numTracks; i++)
                Tracks[i].Tick( tval[i]);
}


bool mi::Work(float *psamples, int numsamples, int const)
{
        bool gotsomething = false;

        for ( in t i=0; i<numTracks; i++) {
                if ( Tracks[i].AEGState) {
                        if ( !gotsomething) {
                                Tracks[i].Work( psamples, numsamples);
                                gotsomething = true;
                        }
            else {
                                float *paux = pCB ->GetAuxBuffer();
                                Tracks[i].Work( paux, numsamples);
                                DSP_Add(psamples, paux, numsamples);
                        }
                }
        }
        return gotsomething;
}


void mi::Stop()
{
        for( int i=0; i<numTracks; i++)
                Tracks[i].Stop();
}

void mi::ComputeCoefs( float *coefs, int freq, int r, int t)
{

        float omega = 2*PI*Cutoff( freq)/pMasterInfo ->SamplesPerSec;
    float sn = sin( omega);
    float cs = cos( omega);
    float alpha;
        if( t<2)
                alpha = sn / Resonance( r *(freq+70)/(127.0+70));
        else
                alpha = sn * sinh( Bandwidth( r) * om ega/sn);

        float a0, a1, a2, b0, b1, b2;

        switch( t) {
        case 0: // LP
                b0 =  (1  - cs)/2;
                b1 =   1  - cs;
                b2 =  (1  - cs)/2;
                a0 =   1 + alpha;
                a1 =  -2*cs;
                a2 =   1  - alpha;
                break;
        case 1: // HP
                b0 =  (1 + cs)/2;
                b1 = -(1 + cs);
                b2 =  (1 + cs)/2;
                a0 =   1 + alpha;
                a1 =  -2*cs;
                a2 =   1 - alpha;
                break;
        case 2: // BP
                b0 =   alpha;
                b1 =   0;
                b2 =  -alpha;
                a0 =   1 + alpha;
                a1 =  -2*cs;
                a2 =   1  - alpha;
                break;
        case 3: // BR
                b0 =   1;
                b1 =  -2*cs;
                b2 =   1;
                a0 =   1 + alpha;
                a1 =  -2*cs;
                a2 =   1  - alpha;
                break;
        }

        coefs[0] = b0/a0;
        coefs[1] = b1/a0;
        coefs[2] = b2/a0;
        coefs[3] = -a1/a0;
        coefs[4] = -a2/a0;
}
```

```
// M4 Buzz plugin by MAKK makk@gmx.de
// released in July 1999
// formulas for the filters by Robert Bristow -Johnson pbjrbj@viconet.com
// a.k.a. robert@audioheads.com

#define NUMWAVES 126

#include <string.h>
#include <stdlib.h>
#include <assert.h>
#include <math.h>
#include <float.h>
#include "../../MachineInterface.h"
#include "../../dsplib/dsplib.h"


#pragma optimize ("a", on)

#define MAX_TRACKS                              8

#define EGS_NONE                                0
#define EGS_ATTACK                              1
#define EGS_SUSTAIN                             2
#define EGS_RELEASE                              3

float *coefsTab = new float [4*128*128*8];
float *LFOOscTab = new float [0x10000];


extern short waves[];

CMachineParameter const paraNote =
{
        pt_note,                                        //  type
        "Note",
        "Note",                               //  description
        NOTE_MIN,                                       //  Min
        NOTE_MAX,                                       //  Max
        0,                                              //  NoValue
        0,                                                        //  Flags
        0                                                         //  default
};

CMachineParameter const paraWave1 =
{
        pt_byte,                                        //  type
        "Osc1 Wave",
        "Oscillator 1 Waveform",              //  description
        0,                                                        //  Min
        NUMWAVES,                             //  Max
        0xff,                                           //  NoValue
        MPF_STATE,                                      //  Flags
        0                                                         //  default
};

CMachineParameter const paraPulseWidth1 =
{
        pt_byte,                                        //  type
        "Osc1 PW",
        "Oscillator 1 Pulse Width",                                       //   description
        0,                                                        //  Min
        127,                                            //  Max
        0xff,                                           //  NoValue
        MPF_STATE,                                      //  Flags
        0x40                                                      //  default
};

CMachineParameter const paraWave2 =
{
        pt_byte,                                        //  type
        "Osc2 Wave",
        "Oscillator 2 Waveform",              //  description
        0,                                                        //  Min
        NUMWAVES,                                                 //  Max
        0xff,                                           //  NoValue
        MPF_STATE,                                      //  Flags
        0                                                         //  default
};

CMachineParameter const paraPulseWidth2 =
{
        pt_byte,                                        //  type
        "Osc2 PW",
        "Oscillator 2 Pulse Width",                                         // description
        0,                                                        //  Min
        127,                                            //  Max
        0xff,                                           //  NoValue
        MPF_STATE,                                      //  Flags
        0x40                                            //  default
};


CMachineParameter const paraMix =
{
        pt_byte,                                        //  type
        "Osc Mix",
        "Oscillator Mix (Osc1 < -> Osc2) ... 00h=Osc1  7Fh=Osc2",
        0,                                                        //  Min
        127,                                            //  Max
        0xff,                                           //  NoValue
        MPF_STATE,                                      //  Flags
        0x40                                            //  default
```

```
};

CMachineParameter const paraMixType =
{
        pt_byte,                                        //  type
        "Osc MixType",
        "Oscillator Mix Type",                                                  //  description
        0,                                                      //  Min
        7,                                                      //  Max
        0xff,                                           //  NoValue
        MPF_STATE,                                      //  Flags
        0                                                       //  default
};



CMachineParameter const paraSync =
{
        pt_switch,                                      //  type
        "Osc2 Sync",
        "Oscillator 2 Sync: Oscillator 2 synced by Oscillator 1   ... 0=off  1=on",                            // description
        SWITCH_OFF,                                     //  Min
        SWITCH_ON,                                      //  Max
        SWITCH_NO,                                      //  NoValue
        MPF_STATE,                                      //  Flags
        SWITCH_OFF                                      //  default
};



CMachineParameter const paraDetuneSemi=
{
        pt_byte,                                        //  type
        "Osc2 SemiDet",
        "Oscillator 2 Semi Detune in Halfnotes ... 40h=no detune",                                  // description
        0,                                                      //  Min
        127,                                            //  Max
        0xff,                                           //  NoValue
        MPF_STATE,                                      //  Flags
        0x40                                            //  default
};

CMachineParameter const paraDetuneFine=
{
        pt_byte,                                        //  type
        "Osc2 FineDet",
        "Oscillator 2 Fine Detune ... 40h=no detune",
        0,                                                      //  Min
        127,                                            //  Max
        0xff,                                           //  NoValue
        MPF_STATE,                                      //  Flags
        0x50                                                    //  default
};

CMachineParameter const paraGlide =
{
        pt_byte,                                        //  type
        "Pitch Glide",
        "Pitch Glide ... 00h=no Glide  7Fh=maximum Glide",                                          //  description
        0,                                                      //  Min
        127,                                            //  Max
        0xff,                                           //  NoValue
        MPF_STATE,                                      //  Flags
        0                                                       //  default
};

CMachineParameter const paraSubOscWave =
{
        pt_byte,                                        //  type
        "SubOsc Wave",
        "Sub Oscillator Waveform",                                              // description
        0,                                                      //  Min
        NUMWAVES-1,                                             //  Max
        0xff,                                           //  NoValue
        MPF_STATE,                                      //  Flags
        0                                                       //  default
};

CMachineParameter const par aSubOscVol =
{
        pt_byte,                                        //  type
        "SubOsc Vol",
        "Sub Oscillator Volume",                  //  description
        0,                                                      //  Min
        127,                                            //  Max
        0xff,                                           //  NoValue
        MPF_STATE,                                      //  Flags
        0x40                                            //  default
};

CMachineParameter const paraVolume =
{
        pt_byte,                                        //  type
        "Volume",
        "Volume (Sustain-Level)",                                               // description
        0,                                                      //  Min
        127,                                            //  Max
        0xff,                                           //  NoValue
        MPF_STATE,                                      //  Flags
        0x40                                            //  default
};

CMachineParameter const paraAEGAttackTime =
{
        pt_byte,                                             //  type
        "Amp Attack",
        "Amplitude Envelope Attack Time",                    //  description
        0,                                                              //  Min
        127,                                                 //  Max
        0xff,                                                //  NoValue
        MPF_STATE,                                           //  Flags
        5                                                               //  default
};

CMachineParameter const paraAEGSustainTime =
{
        pt_byte,                                             //  type
        "Amp Sustain",
        "Amplitude Envelope Sustain Time",                                                // description
        0,                                                              //  Min
        127,                                                 //  Max
        0xff,                                                //  NoValue
        MPF_STATE,                                           //  Flags
        0x10                                                            //  default
};

CMachineParamet er const paraAEGReleaseTime =
{
        pt_byte,                                             //  type
        "Amp Release",
        "Amplitude Envelope Release Time",                                                // description
        0,                                                              //  Min
        127,                                                 //  Max
        0xff,                                                //  NoValue
        MPF_STATE,                                           //  Flags
        0x20                                                            //  default
};

CMachineParameter const paraFilterType =
{
        pt_byte,                                             //  type
        "Filter Type",
        "Filter Type ... 0=LowPass24  1=LowPass18  2=LowPass12  3=HighPass  4=BandPass  5=BandReject",                     // description
        0,                                                              //  Min
        5,                                                   //  Max
        0xff,                                                //  NoValue
        MPF_STATE,                                           //  Flags
        2 // LP12                                                                //  default
};

CMachineParameter const paraCutoff =
{
        pt_byte,                                             //  type
        "Filter Cutoff",
        "Filter Cutoff Frequency",                                              // description
        0,                                                              //  Min
        127,                                                 //  Max
        0xff,                                                //  NoValue
        MPF_STATE,                                           //  Flags
        32                                                              //  default
};

CMachineParameter const paraResonance =
{
        pt_byte,                                             //  type
        "Filter Q/BW",
        "Filter Res onance/Bandwidth",                                  // description
        0,                                                              //  Min
        127,                                                 //  Max
        0xff,                                                //  NoValue
        MPF_STATE,                                           //  Flags
        32                                                              //  default
};

CMachineParameter const paraPEGAttackTime =
{
        pt_byte,                                             //  type
        "Pitch Atta ck",
        "Pitch Envelope Attack Time",                                   //  description
        0,                                                              //  Min
        127,                                                 //  Max
        0xff,                                                //  NoVa lue
        MPF_STATE,                                           //  Flags
        7                                                               //  default
};


CMachineParameter const paraPEGDecayTime =
{
        pt_byte,                                             //  type
        "Pitc h Decay",
        "Pitch Envelope Decay Time",                         //  description
        0,                                                              //  Min
        127,                                                 //  Max
        0xff,                                                //  NoValue
        MPF_STATE,                                           //  Flags
        0x0b                                                            //  default
};


CMachineParameter const paraPEnvMod =
{
        pt_byte,                                             //  type
        "Pit ch EnvMod",
        "Pitch Envelope Modulation",                                    // description
        0,                                                              //  Min
        127,                                                 //  Max
        0xff,                                                //  NoValue
        MPF_STATE,                                      //  Flags
        0x40+32                                                   //  default
};


CMachineParameter const paraFEGAttackTime =
{
        pt_byte,                                        //  type
        "Filter Attack",
        "Filter Envelope Attack Time",                            //  description
        0,                                                        //  Min
        127,                                            //  Max
        0xff,                                           //  NoValue
        MPF_STATE,                                      //  Flags
        7                                                         //  default
};

CMachineParameter const paraFEGSustainTime =
{
        pt_byte,                                        //  type
        "Filter Sustain",
        "Filter Envelope Sustain Time",                           //  description
        0,                                                        //  Min
        127,                                            //  Max
        0xff,                                           //  NoValue
        MPF_STATE,                                      //  Flags
        0x0e                                                      //  default
};


CMachineParameter const paraFEGReleaseTime =
{
        pt_byte,                                        //  type
        "Filter Release",
        "Filter Envelope Release Time",                           //  description
        0,                                                        //  Min
        127,                                            //  Max
        0xff,                                           //  NoValue
        MPF_STATE,                                      //  Flags
        0x0f                                                      //  default
};


CMachineParameter const paraFEnvMod =
{
        pt_byte,                                        //  type
        "Filter EnvMod",
        "Filter Envelope Modulation ... <40h neg. EnvMod  40h=no EnvMod  >40h pos. EnvMod",
        0,                                                        //  Min
        127,                                            //  Max
        0xff,                                           //  NoValue
        MPF_STATE,                                      //  Flags
        0x40+32                                                   //  default
};

// LFOs
CMachineParameter const paraLFO1Dest =
{
        pt_byte,                                        //  type
        "LFO1 Dest",
        "Low Frequency Oscillator 1 Destination",                           //  description
        0,                                                        //  Min
        15,                                             //  Max
        0xff,                                           //  NoValue
        MPF_STATE,                                      //  Flags
        0                                                         //  default
};

CMachineParameter const paraLFO1Wave =
{
        pt_byte,                                        //  type
        "LFO1 Wave",
        "Low Frequency Oscillator 1 Waveform",                              //  description
        0,                                                        //  Min
        4,                                              //  Max
        0xff,                                           //  NoValue
        MPF_STATE,                                      //  Flags
        0                                                         //  default
};

CMachineParameter const paraLFO1Freq =
{
        pt_byte,                                        //  type
        "LFO1 Freq",
        "Low Frequency Oscillator 1 Frequency",                             //  description
        0,                                                        //  Min
        127,                                            //  Max
        0xff,                                           //  NoValue
        MPF_STATE,                                      //  Flags
        0                                                         //  default
};

CMachineParameter const paraLFO1Amount =
{
        pt_byte,                                        //  type
        "LFO1 Amount",
        "Low Frequency Oscillator 1 Amount",                                //  description
        0,                                                        //  Min
        127,                                            //  Max
        0xff,                                           //  NoValue
        MPF_STATE,                                      //  Flags
        0                                                         //  default
};
```

```
// lfo2
CMachineParameter const paraLFO2Dest =
{
        pt_byte,                                        //  type
        "LFO2 Dest",
        "Low Frequency Oscillator 2 Destination",                       //  description
        0,                                                      //  Min
        15,                                             //  Max
        0xff,                                           //  NoValue
        MPF_STATE,                                      //  Flags
        0                                                       //  default
};

CMachineParameter const paraLFO2Wave =
{
        pt_byte,                                        //  type
        "LFO2 Wave",
        "Low Frequency Oscillator 2 Waveform",                          //  description
        0,                                                      //  Min
        4,                                              //  Max
        0xff,                                           //  NoValue
        MPF_STATE,                                      //  Flags
        0                                                       //  default
};

CMachineParameter const paraLFO2Freq =
{
        pt_byte,                                        //  type
        "LFO2 Freq",
        "Low Frequency Oscillator 2 Frequency",                         //  description
         0,                                                     //  Min
        127,                                            //  Max
        0xff,                                           //  NoValue
        MPF_STATE,                                      //  Flags
        0                                                       //  default
};

CMachineParameter const paraLFO2Amount =
{
        pt_byte,                                        //  type
        "LFO2 Amount",
        "Low Frequency Oscillator 2 Amount",                            //  description
        0,                                                      //  Min
        127,                                            //  Max
        0xff,                                           //  NoValue
        MPF_STATE,                                      //  Flags
        0                                                       //  default
};

CMachineParameter const paraLFO1PhaseDiff =
{
        pt_byte,                                        //  type
        "LFO1 Ph Diff",
        "Low Frequency Oscillator 1 Phase Difference: 00h =0°  40h=180°  7Fh=357°",
        0,                                                      //  Min
        127,                                            //  Max
        0xff,                                           //  NoValue
        MPF_STATE,                                      //  Flags
        0x40
};
CMachineParameter const paraLFO2PhaseDiff =
{
        pt_byte,                                        //  type
        "LFO2 Ph Diff",
        "Low Frequency Oscillator 2 Phase Difference: 00h=0°  40h=180°  7Fh=357°",
        0,                                                      //  Min
        127,                                            //  Max
        0xff,                                           //  NoValue
        MPF_STATE,                                      //  Flags
        0x40
};


// ATTRIBUTES

CMachineAttribute const attrLFO1ScaleOsc1 =
{
        "LFO1 Oscillator1 Scale",
            0,
        127,
        127
};

CMachineAttribute const attrLFO1ScalePW1 =
{
        "LFO1 PulseWidth1 Scale",
            0,
        127,
        127
};

CMachineAttribute const attrLFO1ScaleVolume =
{
        "LFO1 Volume Scale",
            0,
        127,
        127
};

CMachineAttribute const attrLFO1ScaleCutoff =
{
        "LFO1 Cutoff Scale",
            0,
        127,
        127
};

CMachineAttr ibute const attrLFO2ScaleOsc2 =
{
        "LFO2 Oscillator2 Scale",
            0,
        127,
        127
};

CMachineAttribute const attrLFO2ScalePW2 =
{
        "LFO2 PulseWidth2 Scale",
            0,
        127,
        127
};

CMachineAttribute con st attrLFO2ScaleMix =
{
        "LFO2 Mix Scale",
        0,
        127,
        127
};

CMachineAttribute const attrLFO2ScaleReso =
{
        "LFO2 Resonance Scale",
            0,
        127,
        127
};


CMachineAttribute const *pAttributes[] =
{
        &attrLFO1ScaleOsc1,
        &attrLFO1ScalePW1,
        &attrLFO1ScaleVolume,
        &attrLFO1ScaleCutoff,
        &attrLFO2ScaleOsc2,
        &attrLFO2ScalePW2,
        &attrLFO2ScaleMix,
        &attrLFO2ScaleReso
};



CMachineParameter const *p Parameters[] = {
                &paraWave1,
        &paraPulseWidth1,
        &paraWave2,
        &paraPulseWidth2,
        &paraDetuneSemi,
        &paraDetuneFine,
        &paraSync,
        &paraMixType,
        &paraMix,
        &paraSubOscWave,
        &paraSubOscVol,

        &paraPEGAttackTime,
        &paraPEGDecayTime,
        &paraPEnvMod,
        &paraGlide,

        &paraAEGAttackTime,
        &paraAEGSustainTime,
        &paraAEGReleaseTime,

        &paraFilterType,
        &paraCutoff,
        &paraResonance,
        &paraFEGAttackTime,
        &paraFEGSustainTime,
        &paraFEGReleaseTime,
        &paraFEnvMod,

        // LFO 1
        &paraLFO1Dest,
        &paraLFO1Wave,
        &paraLFO1Freq,
        &paraLFO1Amount,
                & paraLFO1PhaseDiff,
        // LFO 2
        &paraLFO2Dest,
        &paraLFO2Wave,
        &paraLFO2Freq,
        &paraLFO2Amount,
                &paraLFO2PhaseDiff,

        &paraNote,
        &paraVolume,
};

#pragma pack(1)


class gvals
{
public:
        byte Wave1;
```

```
	byte PulseWidth1;
	byte Wave2;
	byte PulseWidth2;
	byte DetuneSemi;
	byte DetuneFine;
	byte Sync;
	byte MixType;
	byte Mix;
	byte SubOscWave;
	byte SubOscVol;
	byte PEGAttackTime;
	byte PEGDecayTime;
	byte PEnvMod;
	byte Glide;

	byte AEGAttackTime;
	byte AEGSustainTime;
	byte AEGReleaseTime;

	byte FilterType;
	byte Cutoff;
	byte Resonance;
	byte FEGAttackTime;
	byte FEGSustainTime;
	byte FEGReleaseTime;
	byte FEnvMod;

	byte LFO1Dest;
	byte LFO1Wave;
	byte LFO1Freq;
	byte LFO1Amount;

		byte LFO1PhaseDiff;
	byte LFO2Dest;
	byte LFO2Wave;
	byte LFO2Freq;
	byte LFO2Amount;
		byte LFO2PhaseDiff;
};

class tvals
{
public:
	byte Note;
	byte Volume;
};

class avals
{
public:
	int LFO1ScaleOsc1;
	int LFO1ScalePW1;
	int LFO1ScaleVolume;
	int LFO1ScaleCutoff;
	int LFO2ScaleOsc2;
	int LFO2ScalePW2;
	int LFO2ScaleMix;
	int LFO2ScaleReso;
};



#pragma pack()

CMachineInfo const MacInfo =
{
	MT_GENERATOR,			//  type
	MI_VERSION,
	0,						//  flags
	1,						//  min tracks
	MAX_TRACKS,			//  max tracks
	35,				//  numGlobalParameters
	2,				//  numTrackParameters
	pParameters,
	8,
	pAttributes,
#ifdef _DEBUG
	"M4 by Makk (Debug build)",			// name
#else
	"M4 by Makk",
#endif
	"M4",					//  short name
	"Makk",					//  author
	NULL
};


class mi;

class CTrack
{
public:
	void Tick(tvals const &tv);
	void Stop();
	void Init();
	void Work(float *psamples, int numsamples);
	inline int Osc();
	inline float VCA();
	inline float CTrack::Filter( float x);
	void NewPhases();
	int MSToSamples(double const ms);

public:
        // ......Osc......
        int Phase1, Phase2, PhaseSub;
        int Ph1, Ph2;
        float center1, center2;
                int c1, c2;
        float PhScale1A, PhScale1B;
        float PhScale2A, PhScale2B;
        int PhaseAdd1, PhaseAdd2;
        float Frequency, FrequencyFrom;
                // Glide
        bool GlideActive;
        float GlideMul, GlideFactor;
        int GlideCount;
        // PitchEnvMod
        bool PitchModActive;
        // PEG ... AD -Hüllkurve
        int PEGState;
        int PEGCount;
        float PitchMul, PitchFactor;
        // random generator... rauschen
        short r1, r2, r3, r4;

        float OldOut; // gegen extreme Knackser/Wertesprünge

        // .........AEG........ ASR -Hüllkurve
        int AEGState;
        int AEGCount;
        int Volume;
        int Amp;
        int AmpAdd;

        // ........Filter..........
        float x1, x2, y1, y2;
        float x241, x242, y241, y242;
        int FEGState;
        int FEGCount;
        float Cut;
        float CutAdd;

        // .........LFOs...........
        int PhLFO1, PhLFO2;

        mi *pmi; // ptr to MachineInterface


};


class mi : public CMachineInterface
{
public:
        mi();
        virtual ~mi();

        virtual void Init(CMachineDataInput * const pi);
        virtual void Tick();
        virtual bool Work(float *psamples, int numsamples, int const mode);
        virtual void SetNum Tracks(int const n) { numTracks = n; }
        virtual void Stop();
        virtual char const *mi::DescribeValue(int const param, int const value);
        void ComputeCoefs( float *coefs, int f, int r, int t);
        // skalefuncs
        inline float s calLFOFreq( int v);
        inline float scalEnvTime( int v);
        inline float scalCutoff( int v);
        inline float scalResonance( float v);
        inline float scalBandwidth( int v);
        inline int mi::MSToSamples(double const ms);
public:

        // OSC
        bool noise1, noise2;
        int SubOscVol;
        float Center1, Center2;
        const short *pwavetab1, *pwavetab2, *pwavetabsub;

        // Filter
        float *coefsTabOffs; // abhängig vom FilterTyp
        int Cutoff, Resona nce;
        bool db24, db18;
        // PEG
        int PEGAttackTime;
        int PEGDecayTime;
        int PEnvMod;
        bool PitchMod;
        // AEG
        int AEGAttackTime;
        int AEGSustainTime;
        int AEGReleaseTime;
        // FEG
        int FEGAttackTime;
        int FEGSustainTime;
        int FEGReleaseTime;
        int FEnvMod;
        // Glide
        bool Glide;
        int GlideTime;
        // Detune
        float DetuneSemi, DetuneFine;
        bool Sync;
        // LFOs

        bool LFO1Noise, LFO2Noise; // andere Frequenz
        bool LFO1Synced,LFO2Synced; // zum Songtempo
        const short *pwavetabLFO1, *pwavetabLFO2;
        int PhaseLFO1, PhaseLFO2;
        int PhaseAddLFO1, PhaseAddLFO2;
        int LFO1Freq, LFO2 Freq;
        int LFO1PhaseDiff, LFO2PhaseDiff;

        // Amounts
        int LFO1Amount, LFO2Amount;
        int LFO1AmountOsc1;
        float LFO1AmountPW1;
        int LFO1AmountVolume;
        int LFO1AmountCutoff;
        int LFO2AmountOsc2;
        float LFO2AmountPW2;
        int LFO2AmountMix;
        int LFO2AmountReso;

        float TabSizeDivSampleFreq;
        int numTracks;
        CTrack Tracks[MAX_TRACKS];

        // LFO
        // 1
        bool LFO_Osc1;
        bool LFO_PW1;
        bool LFO_Amp;
        bool LFO_Cut;
        // 2
        bool LFO_Osc2;
        bool LFO_PW2;
        bool LFO_Mix;
    bool LFO_Reso;
        // OscMix
        int Bal1, Bal2;
        int MixType;

        avals aval; // attributes
        gvals gval; // gl obals
        tvals tval[MAX_TRACKS]; // track -vals


};


DLL_EXPORTS

// Skalierungsmethoden
inline float mi::scalCutoff( int v)
{
        return (float)(pow( (v+5)/(127.0+5), 1.7)*13000+30);
}
inline float mi::scalResonance( float v)
{
        return (f loat)(pow( v/127.0, 4)*150+0.1);
}
inline float mi::scalBandwidth( int v)
{
        return (float)(pow( v/127.0, 4)*4+0.1);
}

inline float mi::scalLFOFreq( int v)
{
        return (float)((pow( (v+8)/(116.0+8), 4) -0.000017324998565270)*40.00072);
}

inline float mi::scalEnvTime( int v)
{
        return (float)(pow( (v+2)/(127.0+2), 3)*10000);
}

//////////////////////////////////////////////////////
// CTRACK METHODEN
//////////////////////////////////////////////////////

inline int mi::MSToSamples(double  const ms)
{
        return (int)(pMasterInfo ->SamplesPerSec * ms * (1.0 / 1000.0)) + 1; // +1 wg. div durch 0
}


void CTrack::Stop()
{
        AEGState = EGS_NONE;
}

void CTrack::Init()
{
        AEGState = EGS_NONE;
                FEGState = EGS_NONE;
                PEGState = EGS_NONE;
        r1=26474; r2=13075; r3=18376; r4=31291; // randomGenerator
        Phase1 = Phase2 = Ph1 = Ph2 = PhaseSub = 0; // Osc starten neu
        x1 = x2 = y1 = y2 = 0; //Filter
        x241 = x242 = y241 = y242 = 0; / /Filter
        OldOut = 0;
                Amp = 0;
                AEGCount =  -1;
                FEGCount =  -1;
                PEGCount =  -1;
                center1 = pmi ->Center1;
                center2 = pmi ->Center2;
                PhScale1A = 0. 5/center1;
                PhScale1B = 0.5/(1 -center1);
                PhScale2A = 0.5/center2;
                PhScale2B = 0.5/(1 -center2);
                c1 = center1*0x8000000;
                c2 = center2*0x8000000;
                GlideActive = fals e;
                PitchModActive = false;
                Volume = paraVolume.DefValue << 20;

}

void CTrack::Tick( tvals const &tv)
{
        if( tv.Volume != paraVolume.NoValue)
                Volume = tv.Volume << 20;

        if( tv.Note != paraNote .NoValue) { // neuer wert
                if( (tv.Note >= NOTE_MIN) && (tv.Note <= NOTE_MAX)) { // neue note gesetzt
                        FrequencyFrom = Frequency;
                        Frequency = (float)(16.3516*pow(2,((tv.Note>>4)*12+(tv.Note&0x0f  )-1)/12.0));

                        if( pmi ->Glide) {
                                GlideActive = true;
                                if( Frequency > FrequencyFrom)
                                        GlideMul = (float)pow( 2, 1.0/pmi  ->GlideTime);
                                else
                                        GlideMul = (float)pow( 0.5, 1.0/pmi  ->GlideTime);
                                GlideFactor = 1;
                                GlideCount = (int)(log( Frequency/FrequencyFrom  )/log(GlideMul));
                        }
                        else
                                GlideActive = false;

                        // trigger envelopes neu an...
                        // Amp
                        AEGState = EGS_ATTA CK;
                        AEGCount = pmi ->AEGAttackTime;
                        AmpAdd = Volume/pmi ->AEGAttackTime;
                        Amp = 0; //AmpAdd; // fange bei 0 an
                        // Pitch
                        if( pmi ->PitchMod) {
                                PitchModActive = true;
                                PEGState = EGS_ATTACK;
                                PEGCount = pmi ->PEGAttackTime;
                                PitchMul = (float)pow( pow( 1.01, pmi  ->PEnvMod), 1.0/pmi->PEGAttackTime);
                                PitchFactor = 1.0;
                        }
                        else
                                PitchModActive = false;

                        // Filter
                        FEGState =  EGS_ATTACK;
                        FEGCount = pmi ->FEGAttackTime;
                        CutAdd = ((float)pmi ->FEnvMod)/pmi->FEGAttackTime;
                        Cut = 0.0; // fange bei 0 an


                } else
                        if( tv.Note  == NOTE_OFF)
                                AEGState = EGS_NONE; // note aus

        }

        if( GlideActive) {
                PhaseAdd1 = (int)(FrequencyFrom*pmi ->TabSizeDivSampleFreq*0x10000);
                PhaseAdd2 = (int)(FrequencyFrom*pmi ->DetuneSemi*pmi->DetuneFine*pmi->TabSizeDivSampleFreq*0x10000);
        }
        else {
                PhaseAdd1 = (int)(Frequency*pmi ->TabSizeDivSampleFreq*0x10000);
                PhaseAdd2 = (int)(Frequency*pmi ->DetuneSemi*pmi->DetuneFine*pmi->TabSizeD ivSampleFreq*0x10000);
        }

}


inline int CTrack::Osc()
{
        int o, o2;
        int B1, B2;

        if( pmi->LFO_Mix) { // LFO -MIX
                B2 = pmi ->Bal2 + ((pmi->pwavetabLFO2[((unsigned)PhLFO2)>>21]*pmi ->LFO2AmountMix)>>15);
                        if( B2<0)
                                B2 = 0;
                        else
                                if( B2>127)
                                        B2 = 127;
                        B1 = 127 -B2;

                        // osc 1
                        if( pmi ->noise1) {
                                short t = r1+r2+r3+r4;
                                r1=r2; r2=r3; r3=r4; r4=t;
                                o = (t*B1)>>7;
                        }
                         else
                                o = (pmi ->pwavetab1[(unsigned)Ph1>>16]*B1)>>7;

                        // osc2
                        if( pmi ->noise2) {
                                short u = r1+r2+r3+r4;
                                r1=r2; r2 =r3; r3=r4; r4=u;
                                o2 = (u*B2)>>7;
                        }
                        else
                                o2 = (pmi ->pwavetab2[(unsigned)Ph2>>16]*B2)>>7;
                }
                else { // kein LFO
                        // osc1
                        if( pmi ->noise1) {
                                short t = r1+r2+r3+r4;
                                r1=r2; r2=r3; r3=r4; r4=t;
                                o = (t*pmi ->Bal1)>>7;
                        }
                        else
                                o = (pmi ->pwavetab1[(unsigned)Ph1>>16]*pmi ->Bal1)>>7;

                        // osc2
                        if( pmi ->noise2) {
                                short u = r1+r2+r3+r4;
                                r1=r2; r2=r3; r3=r4; r4=u;
                                o2 = (u*pmi ->Bal2)>>7;
                        }
                        else
                                o2 = (pmi ->pwavetab2[(unsigned)Ph2>>16]*pmi ->Bal2)>>7;

                }

        // PhaseDependentMixing

                switch( pmi ->MixType)
        {
        case 0: //ADD
                o += o2;
                break;
        case 1: // ABS
                o = (abs(o -o2)<<1)-0x8000;
                br eak;
        case 2: // MUL
                o *= o2;
                                o >>= 15;
                break;
        case 3: // highest amp
                if( abs(o) < abs(o2))
                        o = o2;
                break;
        case 4 : // lowest amp
                if( abs(o) > abs(o2))
                        o = o2;
                break;
        case 5: // AND
                o &= o2;
                break;
        case 6: // OR
                o |= o2;
                break;
        case 7: // XOR
                o ^= o2;
                break;
        }
        return o + ((pmi ->pwavetabsub[PhaseSub>>16]*pmi ->SubOscVol)>>7);
}

inline float CTrack::VCA()
{
        // EG...
        if( !AEGCount --)
                switch( ++AEGState )
                {
                case EGS_SUSTAIN:
                        AEGCount = pmi ->AEGSustainTime;
                        Amp = Volume;
                        AmpAdd = 0;
                        break;
                case EGS_RELEASE:
                        AEGCount = pmi ->AEGReleaseTime;
                        AmpAdd =  -Volume/pmi->AEGReleaseTime;
                        break;
                case EGS_RELEASE + 1:
                        AEGState = EGS_NONE;
                        AEGCoun t = -1;
                        Amp = 0;
                        break;
                }

        Amp +=AmpAdd;

        if( pmi->LFO_Amp) {
                float a =
                  Amp + ((pmi ->pwavetabLFO1[((unsigned)PhLFO1)>>21]*pmi ->LFO1AmountVolum e)<<5);
                if( a<0)
                        a = 0;
                return( a*(1.0/0x8000000));
        }
        else
                return Amp*(1.0/0x8000000);
}


inline float CTrack::Filter( float x)
{
        float y;

        // Envelope
        if( FEGState) {
                if( !FEGCount --)
                        switch( ++FEGState)
                        {
                        case EGS_SUSTAIN:
                                FEGCount = pmi ->FEGSustainTime;
                                Cut = (float)pmi ->FEnvMod;
                                CutAdd = 0.0;
                                break;
                        case EGS_RELEASE:
                                FEGCount = pmi ->FEGReleaseTime;
                                CutAdd = ((float) -pmi->FEnvMod)/pmi ->FEGReleaseTime;
                                break;
                        case EGS_RELEASE + 1:
                                FEGState = EGS_NONE; // false
                                FEGCount =  -1;
                                Cut = 0.0;
                                CutAdd = 0.0;
                                break;
                        }
                Cut += CutAdd;
        }

        // LFO
        // Cut
        int c, r;
        if( pmi ->LFO_Cut)
                c = pmi ->Cutoff + Cut + // Cut = EnvMod
                ((pmi ->pwavetabLFO1[((unsigned)PhLFO1)>>21]*pmi ->LFO1AmountCutoff)>>(7+8));
        else
                c = pmi ->Cutoff + Cut; // Cut = EnvMod
        if( c < 0)
                c = 0;
        else
                if( c > 127)
                        c = 127;
        // Reso
        if( pmi ->LFO_Reso) {
                r = pmi ->Resonance +
                ((pmi ->pwavetabLFO2[((unsigned)PhLFO2)>>21]*pmi ->LFO2AmountReso)>>(7+8)) ;
        if( r < 0)
                r = 0;
        else
                if( r > 127)
                        r = 127;
        }
        else
                r = pmi ->Resonance;


        int ofs = ((c<<7)+r)<<3;
        y = pmi ->coefsTabOffs[ofs]*x +
                pmi ->coefsTabOffs[ofs+1]*x1 +
                pmi ->coefsTabOffs[ofs+2]*x2 +
                pmi ->coefsTabOffs[ofs+3]*y1 +
                pmi ->coefsTabOffs[ofs+4]*y2;

        y2=y1;
        y1=y;
        x2=x1;
        x1=x;
                if ( !pmi ->db24)
                        return y;
                else { // 24 DB
                        float y24 = pmi ->coefsTabOffs[ofs]*y +
                                pmi ->coefsTabOffs[ofs+1]*x241 +
                                pmi ->coefsTabOffs [ofs+2]*x242 +
                                pmi ->coefsTabOffs[ofs+3]*y241 +
                                pmi ->coefsTabOffs[ofs+4]*y242;
                        y242=y241;
                        y241=y24;
                        x242=x241;
                        x241=y;
                        if( !pmi ->db18)
                                return y24;
                        else
                                return (y+y24)*0.5;
                }
}

inline void CTrack::NewPhases()
{
        if( Pi tchModActive) {
                if( GlideActive) {
                        if( pmi ->LFO_Osc1) {
                                float pf = LFOOscTab[(pmi  ->pwavetabLFO1[((unsigned)PhLFO1)>>21]*pmi ->LFO1AmountOsc1>>7) + 0x8000];
                                Phase1 += PhaseAdd1*GlideFactor*PitchFactor*pf;
                                PhaseSub += (PhaseAdd1>>1)*GlideFactor*PitchFactor*pf;
                        }
                        else {
                                Phase1 += PhaseAdd1*GlideFact  or*PitchFactor;
                                PhaseSub += (PhaseAdd1>>1)*GlideFactor*PitchFactor;
                        }
                        if( pmi ->LFO_Osc2)
                                Phase2 += PhaseAdd2*GlideFactor*PitchFactor
                                  *LFOOscTab[(pmi ->pwavetabLFO2[((unsigned)PhLFO2)>>21]*pmi ->LFO2AmountOsc2>>7) + 0x8000];
                        else
                                Phase2 += PhaseAdd2*GlideFactor*PitchFactor;
                        GlideFactor *=  GlideMul;
                        if( !GlideCount --) {
                                GlideActive = false;
                                PhaseAdd1 = (int)(Frequency*pmi  ->TabSizeDivSampleFreq*0x10000);
                                PhaseAdd2 = (int)(F  requency*pmi ->DetuneSemi*pmi ->DetuneFine*pmi ->TabSizeDivSampleFreq*0x10000);
                        }
                }
                else { // kein Glide
                        if( pmi ->LFO_Osc1) {
                                float pf = LFOOscTab[  (pmi->pwavetabLFO1[((unsigned)PhLFO1)>>21]*pmi ->LFO1AmountOsc1>>7) + 0x8000];
                                Phase1 += PhaseAdd1*PitchFactor*pf;
                                PhaseSub += (PhaseAdd1>>1)*PitchFactor*pf;
                        }
                        else {
                                Phase1 += PhaseAdd1*PitchFactor;
                                PhaseSub += (PhaseAdd1>>1)*PitchFactor;
                        }
                        if( pmi ->LFO_Osc2)
                                 Phase2 += PhaseAdd2*PitchFactor
                                  *LFOOscTab[(pmi ->pwavetabLFO2[((unsigned)PhLFO2)>>21]*pmi ->LFO2AmountOsc2>>7) + 0x8000];
                        else
                                Phase2 += PhaseAdd2*PitchFactor;
                }

                PitchFactor *= PitchMul;

                if( !PEGCount --)
                        if( ++PEGState == 2) {// DECAY -PHASE beginnt
                                PEGCount = pmi ->PEGDecayTime;
                                Pi tchMul = pow( pow( 1/1.01, pmi ->PEnvMod), 1.0/pmi ->PEGDecayTime);
                        }
                        else  // AD -Kurve ist zu Ende
                                PitchModActive = false;
        }

        else { // kein PitchMod
                if( GlideActive) {
                        if( pmi ->LFO_Osc1) {
                                float pf = LFOOscTab[(pmi  ->pwavetabLFO1[((unsigned)PhLFO1)>>21]*pmi ->LFO1AmountOsc1>>7) + 0x8000];
                                Phase1 += PhaseAdd1*Glid eFactor*pf;
                                PhaseSub += (PhaseAdd1>>1)*GlideFactor*pf;
                        }
                        else {
                                Phase1 += PhaseAdd1*GlideFactor;
                                PhaseSub += (Ph aseAdd1>>1)*GlideFactor;
                        }
                        if( pmi ->LFO_Osc2)
                                Phase2 += PhaseAdd2*GlideFactor
                                  *LFOOscTab[(pmi ->pwavetabLFO2[((unsigned)PhLFO2)>>21]*pmi ->LFO2AmountOsc2>>7) + 0x8000];
                        else
                                Phase2 += PhaseAdd2*GlideFactor;
                        GlideFactor *= GlideMul;
                        if( !GlideCount --) {
                                GlideActive  = false;
                                PhaseAdd1 = (int)(Frequency*pmi  ->TabSizeDivSampleFreq*0x10000);
                                PhaseAdd2 = (int)(Frequency*pmi  ->DetuneSemi*pmi ->DetuneFine*pmi ->TabSizeDivSampleFreq*0x10000);
                         }
                }
                else {
                        if( pmi ->LFO_Osc1) {
                                float pf = LFOOscTab[(pmi  ->pwavetabLFO1[((unsigned)PhLFO1)>>21]*pmi ->LFO1AmountOsc1>>7) + 0x8000];
                                Pha se1 += PhaseAdd1*pf;
                                PhaseSub += (PhaseAdd1>>1)*pf;
                        }
                        else {
                                Phase1 += PhaseAdd1;
                                PhaseSub += (PhaseAdd1>>1);
                        }
                        if( pmi ->LFO_Osc2)
                                Phase2 += PhaseAdd2
                                  *LFOOscTab[(pmi ->pwavetabLFO2[((unsigned)PhLFO2)>>21]*pmi ->LFO2AmountOsc2>>7) + 0x8000];
                        else
                                Phase2 += PhaseAdd2;
                }
        }


        if( Phase1 & 0xf8000000) { // neuer durchlauf ??
                // PW1
                if( pmi ->LFO_PW1) { //LFO_PW_Mod
                        cente r1 = pmi ->Center1 + (float)pmi ->pwavetabLFO1[((unsigned)PhLFO1)>>21]*
                                                pmi ->LFO1AmountPW1;
                        if( center1 < 0)
                                center1 = 0;
                        else
                                if( center1 > 1)
                                        center1 = 1;
                }
                else  // No LFO
                        center1 = pmi ->Center1;
                PhScale1A = 0.5/center1;
                Ph Scale1B = 0.5/(1 -center1);
                                c1 = center1*0x8000000;
                // PW2
                if( pmi ->LFO_PW2) { //LFO_PW_Mod
                        center2 = pmi ->Center2 + (float)pmi ->pwavetabLFO2[((unsigned)PhLFO2)>>21]*
                                                pmi ->LFO2AmountPW2;
                        if( center2 < 0)
                                center2 = 0;
                        else
                                if( center2 > 1)
                                         center2 = 1;
                }
                else  // No LFO
                        center2 = pmi ->Center2;
                PhScale2A = 0.5/center2;
                PhScale2B = 0.5/(1 -center2);
                                c2 = center2* 0x8000000;

                // SYNC
                if( pmi ->Sync)
                        Phase2 = Phase1; // !!!!!
        }

        Phase1 &= 0x7ffffff;
        Phase2 &= 0x7ffffff;
```

```
        PhaseSub &= 0x7ffffff;

        if( Phase1 < c1)
                Ph1 = Phase1*PhScale1A;
        else
                Ph1 = (Phase1  - c1)*PhScale1B + 0x4000000;

        if( Phase2 < c2)
                Ph2 = Phase2*PhScale2A;
        else
                Ph2 = (Phase2  - c2)*PhScale2B + 0x4000000;

                // LFOs
        PhLFO1 += pmi->PhaseAddLFO1;
        PhLFO2 += pmi->PhaseAddLFO2;
}

void CTrack::Work( float *psamples, int numsamples)
{
        for( int i=0; i<numsamples; i++) {
                if( AEGState) {

                        float o = Osc()*VCA ();
                        *psamples++ = Filter( OldOut + o); // anti knack
                        OldOut = o;
                }
                else
                        *psamples++ = 0;
                NewPhases();
        }
}

//.......................................................
//.................... DESCRIPTION ......................
//......................................................

char const *mi::DescribeValue(int const param, int const value)
{
        static char *MixTypeTab[9] =  {
                "add",
                "difference",
                "mul",
                "highest amp",
                "lowest amp",
                "and",
                "or",
                "xor",
                "random" };

        static char  *LFO1DestTab[16] = {
                "none",
                "osc1",
                "p.width1",
                "volume",
                "cutoff",
                "osc1+pw1", // 12
                "osc1+volume", // 13
                "osc1+cutoff", // 1 4
                "pw1+volume", // 23
                "pw1+cutoff", // 24
                "vol+cutoff", // 34
                "o1+pw1+vol",// 123
                "o1+pw1+cut",// 124
                "o1+vol+cut",// 134
                "pw1+vol+cut",// 234
                "all"// 1234
        };

        static char *LFOWaveTab[5] = {
                "sine",
                "saw",
                "square",
                "triangle",
                "random",
        };

        static char *LFO2DestTab[16] =  {
                "none",
                "osc2",
                "p.width2",
                "mix",
                "resonance",
                "osc2+pw2", // 12
                "osc2+mix", // 13
                "osc2+res", // 14
                "pw2+mi x", // 23
                "pw2+res", // 24
                "mix+res", // 34
                "o2+pw2+mix", // 123
                "o2+pw2+res", // 124
                "o2+mix+res", // 134
                "pw2+mix+res", // 234
                "all" // 1234
        };

        static char *FilterTypeTab[6] = {
                "lowpass24",
                "lowpass18",
                "lowpass12",
                "highpass",
                "bandpass",
                "bandreject" };

#include "waves/wavename.in c"
```

```
	static char txt[16];

	switch(param){
	case 0: // OSC1Wave
	case 2: // OSC2Wave
	case 9: // SubOscWave
				return( wavenames[value]);
				break;
		 case 1: // PW1
	case 3: // PW2
		sprintf(txt, "%u : %u", (int)(value*100.0/127),
								100  -(int)(value*100.0/127));
		break;
	case 4: // semi detune
		if( value == 0x40)
			return "±0 halfnotes";
		else
			if( value > 0x40)
				sprintf( txt, "+%i halfnotes", value  -0x40);
			else
				 sprintf( txt, "%i halfnotes", value -0x40);
		break;
	case 5: // fine detune
		if( value == 0x40)
			return "±0 cents";
		else
			if( value > 0x4 0)
				sprintf( txt, "+%i cents", (int)((value  -0x40)*100.0/63));
			else
				sprintf( txt, "%i cents", (int)((value  -0x40)*100.0/63));
		break;

	case 6: // Sy nc
		if( value == SWITCH_ON)
			return( "on");
		else
			return( "off");
		break;

	case 7: // MixType
				return MixTypeTab[value];
				break;
	case 8: // Mix
		switch( value) {
		case 0:return "Osc1";
		case 127:return "Osc2";
		default: sprintf(txt, "%u%% : %u%%", 100 -(int)(value*100.0/127),
								(int)(value*100.0/127));
		}
		break;

	case 11: // Pitch Env
	case 12: // Pitch Env
	case 15: // Amp Env
	case 16: // Amp Env
	case 17 : // Amp Env
	case 21: // Filter Env
	case 22: // Filter Env
	case 23: // Filter Env
		sprintf( txt, "%.4f sec", scalEnvTime( value)/1000);
		break;

	case 13: // PitchEnvMod
	case 24: // Filt  ENvMod
		sprintf( txt, "%i", value -0x40);
		break;
	case 18: //FilterType
				return FilterTypeTab[value];
				break;
	case 25: // LFO1Dest
				 return LFO1DestTab[value];
				break;
	case 30: // LFO2Dest
				return LFO2DestTab[value];
				break;
	case 26: // LFO1Wave
	case 31: //  LFO2Wave
				return LFOWaveTab[value];
				break;
	case 27: // LFO1Freq
	case 32: // LFO2Freq
		if( value <= 116)
			sprintf( txt, "%.4f HZ", scalLFOFr eq( value));
		else
			sprintf( txt, "%u ticks", 1<<(value  -117));
		break;
		case 29: //LFO1PhaseDiff
		case 34: //LFO2PhaseDiff
			sprintf( txt, "%i°", value *360/128);
			break;
	default: return NULL;
		}
	return txt;
}



/////////////////////////////////////////////////////
// MACHINE INTERFACE METHODEN
/////////////////////////////////////////////////////

mi::mi()
```

```
{
        GlobalVals = &gval;
        TrackVals = tval;
        AttrVals = (int *)&aval;
}

mi::~mi()
{
}


void mi::Init(CMachineDataInput * const pi)
{
        TabSizeDivSampleFreq = (float)(2048.0/pMasterInfo ->SamplesPerSec);

        // Filter
        coefsTabOffs = coefsTab; // LowPass
        Cutoff = paraCutoff.DefValue;
        Resonance = paraResonance.DefValue;
        db24 = db18 = false;
        //PEG
        PEGAttackTime = MSToSamples( scalEnvTime( paraPEGAttackTime.DefValue));
        PEGDecayTime = MSToSamples( scalEnvTime( paraPEGDecayTime.DefValue));
        PEnvMod = 0;
        // FEG
        FEGAttackTime = MSToSamples( scalEnvTime( paraFEGAttackTime.DefValue));
        FEGSustainTime = MSToSamples( scalEnvTime( paraFEGSustainTim  e.DefValue));
        FEGReleaseTime = MSToSamples( scalEnvTime( paraFEGReleaseTime.DefValue));
        FEnvMod = 0;
        // AEG
        AEGAttackTime = MSToSamples( scalEnvTime( paraAEGAttackTime.DefValue));
        AEGSustainTime = MSToSamples( scalEn vTime( paraAEGSustainTime.DefValue));
        AEGReleaseTime = MSToSamples( scalEnvTime( paraAEGReleaseTime.DefValue));


        pwavetab1 = pwavetab2 = pwavetabsub = waves;


        noise1 = noise2 = Sync = false;
        LFO1Noise = LFO2Noise = false;
        LFO1Synced = LFO2Synced = false;

        PhaseLFO1 = PhaseLFO2 = 0;

        pwavetabLFO1 = pwavetabLFO2 = pCB ->GetOscillatorTable( OWF_SINE);
        DetuneSemi = DetuneFine = 1.0;

        PhaseAddLFO1 = PhaseAddLFO2 = 0;

        SubOscVol = pa raSubOscVol.DefValue;

        // PulseWidth
        Center1 = (float)(paraPulseWidth1.DefValue/127.0);
        Center2 = (float)(paraPulseWidth2.DefValue/127.0);
        LFO1Amount = LFO2Amount = 0;
        LFO1PhaseDiff = paraLFO1PhaseDiff.DefValue<<(9+1 6);
        LFO2PhaseDiff = paraLFO1PhaseDiff.DefValue<<(9+16);

        // OscMix
        Bal1 = 127 -paraMix.DefValue;
        Bal2 = paraMix.DefValue;
        MixType = 0;

        LFO_Osc1 = LFO_PW1 = LFO_Amp = LFO_Cut = false;
        LFO_Osc2 = LFO_PW 2 = LFO_Mix = LFO_Reso = false;

        for( int i=0; i<MAX_TRACKS; i++)
        {
                Tracks[i].pmi = this;
                Tracks[i].Init();
        }

        // generate coefsTab
        for( int t=0; t<4; t++)
                for( int f=0 ; f<128; f++)
                        for( int r=0; r<128; r++)
                                ComputeCoefs( coefsTab+(t*128*128+f*128+r)*8, f, r, t);
        // generate LFOOscTab
        for( int p=0; p<0x10000; p++)
                LFOOscTab[p] = pow(  1.00004230724139582, p -0x8000);

}


void mi::Tick()
{
        // Filter
        if( gval.FilterType != paraFilterType.NoValue)
                if( gval.FilterType == 0){ //LP24
                        db18 = false;
                        db24 = true;
                        coefsTabOffs = coefsTab;
                }
                else {
                        if( gval.FilterType == 1){ //LP24
                                db18 = true;
                                db24 = true;
                                coefsTabOffs = coefsTab;
                        }
                        else {
                                db18 = false;
                                db24 = false;
                                coefsTabOffs = coefsTab + (int)(gval.Fi  lterType-2)*128*128*8;
                        }
                }

        if( gval.Cutoff != paraCutoff.NoValue)
                Cutoff = gval.Cutoff;
        if( gval.Resonance != paraResonance.NoValue)
                Resonance = gval.Resonance;

        // FEG
        if( gval.FEGAttackTime != paraFEGAttackTime.NoValue)
                FEGAttackTime = MSToSamples( scalEnvTime( gval.FEGAttackTime));
        if( gval.FEGSustainTime != paraFEGSustainTime.NoValue)
                FEGSustainTime = MSToSampl es( scalEnvTime( gval.FEGSustainTime));
        if( gval.FEGReleaseTime != paraFEGReleaseTime.NoValue)
                FEGReleaseTime = MSToSamples( scalEnvTime( gval.FEGReleaseTime));
        if( gval.FEnvMod != paraFEnvMod.NoValue)
                FEnvMo d = (gval.FEnvMod  - 0x40)<<1;

        // PEG
        if( gval.PEGAttackTime != paraPEGAttackTime.NoValue)
                PEGAttackTime = MSToSamples( scalEnvTime( gval.PEGAttackTime));
        if( gval.PEGDecayTime != paraPEGDecayTime.NoValue)
                PEGDecayTime = MSToSamples( scalEnvTime( gval.PEGDecayTime));

        if( gval.PEnvMod != paraPEnvMod.NoValue) {
                if( gval.PEnvMod  - 0x40 != 0)
                        PitchMod = true;
                else {
                        Pi tchMod = false;
                        for( int i=0; i<numTracks; i++)
                                Tracks[i].PitchModActive = false;
                }
                PEnvMod = gval.PEnvMod  - 0x40;
        }


        if( gval.Mix != paraMix.NoValue)  {
                Bal1 = 127 -gval.Mix;
                Bal2 = gval.Mix;
        }

        if( gval.Glide != paraGlide.NoValue)
                if( gval.Glide == 0) {
                        Glide = false;
                        for( int i=0; i<numTracks;  i++)
                                Tracks[i].GlideActive = false;
                }
                else {
                        Glide = true;
                        GlideTime = gval.Glide*10000000/pMasterInfo  ->SamplesPerSec;
                }


        // SubOsc
        if( gval.SubOscWave != paraSubOscWave.NoValue)
                pwavetabsub = waves + (gval.SubOscWave<<11);


        if( gval.SubOscVol != paraSubOscVol.NoValue)
                SubOscVol = gval.SubOscVol;

        // PW
        if(  gval.PulseWidth1 != paraPulseWidth1.NoValue)
                Center1 = gval.PulseWidth1/127.0;

        if( gval.PulseWidth2 != paraPulseWidth2.NoValue)
                Center2 = gval.PulseWidth2/127.0;

        // Detune
        if( gval.DetuneSemi != par aDetuneSemi.NoValue)
                DetuneSemi = (float)pow( 1.05946309435929526, gval.DetuneSemi  -0x40);
        if( gval.DetuneFine != paraDetuneFine.NoValue)
                DetuneFine = (float)pow( 1.00091728179580156, gval.DetuneFine  -0x40);
        if( gval.Sync != SWITCH_NO)
                if( gval.Sync == SWITCH_ON)
                        Sync = true;
                else
                        Sync = false;


        if( gval.MixType != paraMixType.NoValue)
                MixType = gval.MixType;

        if( gval.Wave1 != paraWave1.NoValue) { // neuer wert
                if( gval.Wave1 == NUMWAVES)
                        noise1 = true;
                else {
                        noise1 = false;
                        pwavetab1 = waves + (gv al.Wave1 << 11);
                }
        }

        if( gval.Wave2 != paraWave2.NoValue) // neuer wert
                if( gval.Wave2 == NUMWAVES)
                        noise2 = true;
                else {
                        noise2 = false;
                        pwavetab2 = waves + (gval.Wave2 << 11);
                }


        // AEG
        if( gval.AEGAttackTime != paraAEGAttackTime.NoValue)
                AEGAttackTime = MSToSamples( scalEnvTime( gval.AEGAttackTime));
        if( gval.AEGS ustainTime != paraAEGSustainTime.NoValue)
```

```
		AEGSustainTime = MSToSamples( scalEnvTime( gval.AEGSustainTime));
	if( gval.AEGReleaseTime != paraAEGReleaseTime.NoValue)
		AEGReleaseTime = MSToSamples( scalEnvTime( gval.AEGRe  leaseTime));

	// ..........LFO............

	// LFO1
	if( gval.LFO1Dest != paraLFO1Dest.NoValue) {
		   LFO_Osc1 = LFO_PW1 = LFO_Amp = LFO_Cut = false;
		   switch( gval.LFO1Dest) {
//		case 0:  ...none
		case 1:
			LFO_Osc1 = true;
			break;
		case 2:
						LFO_PW1 = true;
			break;
		case 3:
			LFO_Amp = true;
			break;
		case 4:
			LFO_Cut = true;
			break;

		case 5: // 12
			LFO_Osc1 = true;
			LFO _PW1 = true;
			break;
		case 6: // 13
			LFO_Osc1 = true;
			LFO_Amp = true;
			break;
		case 7: // 14
						 LFO_Osc1 = true;
						LFO_Cut = true;
			break;
		case 8: // 23
			LFO_PW1 = true;
			LFO_Amp = true;
			br eak;
		case 9: // 24
			LFO_PW1 = true;
			LFO_Cut = true;
			break;
		case 10: // 34
			LFO_Amp = true;
			LFO_Cut = true;
			break;

		case 11: // 123
			LFO_Osc1 = true;
			LFO_PW1 = true;
			LFO_Amp = true;
			break;
		case 12: // 124
			LFO_Osc1 = true;
			LFO_PW1 = true;
			LFO_Cut = true;
			break;
		case 13: // 134
			LFO_Osc1 = true;
			LFO_Amp = true ;
			LFO_Cut = true;
			break;
		case 14: // 234
			LFO_PW1 = true;
			LFO_Amp = true;
			LFO_Cut = true;
			break;
		case 15: // 1234
			LFO_Osc1 = true;
			LFO_PW1 = true;
			LFO_Amp = true;
			LFO_Cut = true;
			break;
			}
		}

	if( gval.LFO1Wave != paraLFO1Wave.NoValue) {
		pwavetabLFO1 = pCB ->GetOscillatorTable( gval.LFO1Wave);
		if( gval.LFO1Wave == OWF_NOISE)
			LFO1Noise = true;
		else
			LFO1Noise = false;
	}


	if( gval.LFO1Freq != paraLFO1Freq.NoValue)
		if( gval.LFO1Freq>116) {
			LFO1Synced = true;
			LFO1Freq = gval.LFO1Freq  - 117;
		 }
		else {
			LFO1Synced = false;
			LFO1Freq = gval.LFO1Freq;
		}

	if( gval.LFO1Amount != paraLFO1Amount.NoValue)
		LFO1Amount = gval.LFO1Amount;


	if( LFO 1Synced)
		if( LFO1Noise) // sample & hold
			PhaseAddLFO1 = (int)(0x200000/(pMasterInfo  ->SamplesPerTick<<LFO1Freq));
```

```
                else
                        PhaseAddLFO1 = (int)((double)0x200000*2048/(pMasterInfo  ->SamplesPerTick<<LFO1Freq));
        else
                if( LFO1Noise) // sample & hold
                        PhaseAddLFO1 = (int)(scalLFOFreq( LFO1Freq)/pMasterInfo  ->SamplesPerSec*0x200000);
                else
                        PhaseAddLFO1 = (i nt)(scalLFOFreq( LFO1Freq)*TabSizeDivSampleFreq*0x200000);

        // LFO2
                if( gval.LFO2Dest != paraLFO2Dest.NoValue) {
                        LFO_Osc2 = LFO_PW2 = LFO_Mix = LFO_Reso = false;
                        switch( gval.LFO2Dest)  {
//              case 0: ...none
                case 1:
                        LFO_Osc2 = true;
                        break;
                case 2:
                        LFO_PW2 = true;
                        break;
                case 3:
                         LFO_Mix = true;
                        break;
                case 4:
                        LFO_Reso = true;
                        break;
                case 5: // 12
                        LFO_Osc2 = true;
                         LFO_PW2 = true;
                        break;
                case 6: // 13
                        LFO_Osc2 = true;
                        LFO_Mix = true;
                        break;
                case 7: // 14
                        LFO_Osc2 = tr ue;
                        LFO_Reso = true;
                        break;
                case 8: // 23
                        LFO_PW2 = true;
                        LFO_Mix = true;
                        break;
                case 9: // 24
                         LFO_PW2 = true;
                        LFO_Reso = true;
                        break;
                case 10: // 34
                        LFO_Mix = true;
                        LFO_Reso = true;
                        break;
                 case 11: // 123
                        LFO_Osc2 = true;
                        LFO_PW2 = true;
                        LFO_Mix = true;
                        break;
                case 12: // 124
                        LFO_Osc2 = true;
                         LFO_PW2 = true;
                        LFO_Reso = true;
                        break;
                case 13: // 134
                        LFO_Osc2 = true;
                        LFO_Mix = true;
                        LFO_Reso = true;
                        break;
                case 14: // 234
                        LFO_PW2 = true;
                        LFO_Mix = true;
                        LFO_Reso = true;
                        break;
                case 15: // 1234
                         LFO_Osc2 = true;
                        LFO_PW2 = true;
                        LFO_Mix = true;
                        LFO_Reso = true;
                        break;
                }
        }

        if( gval.LFO2Wave != paraLFO2Wave .NoValue) {
                pwavetabLFO2 = pCB ->GetOscillatorTable( gval.LFO2Wave);
                if( gval.LFO2Wave == OWF_NOISE)
                        LFO2Noise = true;
                else
                        LFO2Noise = false;
        }

         if( gval.LFO2Freq != paraLFO2Freq.NoValue)
                if( gval.LFO2Freq>116) {
                        LFO2Synced = true;
                        LFO2Freq = gval.LFO2Freq  - 117;
                }
                else {
                        LFO2Syn ced = false;
                        LFO2Freq = gval.LFO2Freq;
                }

        if( gval.LFO2Amount != paraLFO2Amount.NoValue)
                LFO2Amount = gval.LFO2Amount;

        // LFO -Phasen-Differenzen
        if( gval.LFO1PhaseDiff != para LFO1PhaseDiff.NoValue)
                LFO1PhaseDiff = gval.LFO1PhaseDiff << (9+16);
        if( gval.LFO2PhaseDiff != paraLFO2PhaseDiff.NoValue)
                LFO2PhaseDiff = gval.LFO2PhaseDiff << (9+16);
        if( LFO2Synced)
                if( LFO2Noise) // sample & hold
                        PhaseAddLFO2 = (int)(0x200000/(pMasterInfo->SamplesPerTick<<LFO2Freq));
                else
                        PhaseAddLFO2 = (int)((double)0x200000*2048/(pMasterInfo->SamplesPerTick<<LFO2Freq));
        else
                if( LFO2Noise) // sample & hold
                        PhaseAddLFO2 = (int)(scalLFOFreq( LFO2Freq)/pMasterInfo->SamplesPerSec*0x200000);
                else
                        PhaseAddLFO2 = (int)(scalLFOFreq( LFO2Freq)*TabSizeDivSampleFreq*0x200000);

        // skalierte LFO-Amounts
        LFO1AmountOsc1 = (LFO1Amount*aval.LFO1ScaleOsc1)>>7;
        LFO1AmountPW1 = (LFO1Amount*aval.LFO1ScalePW1/(128.0*127.0*0x8000));
        LFO1AmountVolume = (LFO1Amount*aval.LFO1ScaleVolume)>>7;
        LFO1AmountCutoff = (LFO1Amount*aval.LFO1ScaleCutoff)>>7;
        LFO2AmountOsc2 = (LFO2Amount*aval.LFO2ScaleOsc2)>>7;
        LFO2AmountPW2 = (LFO2Amount*aval.LFO2ScalePW2/(128.0*127.0*0x8000));
        LFO2AmountMix = (LFO2Amount*aval.LFO2ScaleMix)>>7;
        LFO2AmountReso = (LFO2Amount*aval.LFO2ScaleReso)>>7;


        // TrackParams durchgehen
        for (int i=0; i<numTracks; i++)
                Tracks[i].Tick( tval[i]);
}


bool mi::Work(float *psamples, int numsamples, int const)
{
        bool gotsomething = false;

        for ( int i=0; i<numTracks; i++) {
                if ( Tracks[i].AEGState) {
                        Tracks[i].PhLFO1 = PhaseLFO1 + i*LFO1PhaseDiff;
                        Tracks[i].PhLFO2 = PhaseLFO2 + i*LFO2PhaseDiff;
                        if ( !gotsomething) {
                                Tracks[i].Work( psamples, numsamples);
                                gotsomething = true;
                        }
                        else {
                                float *paux = pCB->GetAuxBuffer();
                                Tracks[i].Work( paux, numsamples);
                                DSP_Add(psamples, paux, numsamples);
                        }
                }
        }
        PhaseLFO1 += PhaseAddLFO1*numsamples;
        PhaseLFO2 += PhaseAddLFO2*numsamples;
        return gotsomething;
}


void mi::Stop()
{
        for( int i=0; i<numTracks; i++)
                Tracks[i].Stop();
}

void mi::ComputeCoefs( float *coefs, int freq, int r , int t)
{
        float omega = 2*PI*scalCutoff(freq)/pMasterInfo->SamplesPerSec;
    float sn = (float)sin( omega);
    float cs = (float)cos( omega);
    float alpha;
        if( t<2)
                alpha = (float)(sn / scalResonance( r *(freq+70)/(127.0+70)));
        else
                alpha = (float)(sn * sinh( scalBandwidth( r) * omega/sn));

        float a0, a1, a2, b0, b1, b2;

        switch( t) {
        case 0: // LP
                b0 =  (1 - cs)/2;
                b1 =   1 - cs;
                b2 =  (1 - cs)/2;
                a0 =   1 + alpha;
                a1 =  -2*cs;
                a2 =   1 - alpha;
                break;
        case 1: // HP
                b0 =  (1 + cs)/2;
                b1 = -(1 + cs);
                b2 =  (1 + cs)/2;
                a0 =   1 + alpha;
                a1 =  -2*cs;
                a2 =   1 - alpha;
                break;
        case 2: // BP
                b0 =   alpha;
                b1 =   0;
                b2 =  -alpha;
                a0 =   1 + alpha;
                a1 =  -2*cs;
                a2 =   1 - alpha;
                break;
        case 3: // BR
                b0 =   1;
                b1 =  -2*cs;
                b2 =   1;
                a0 =   1 + alpha;
                a1 =  -2*cs;
                a2 =   1  - alpha;
                break;
        }

        coefs[0] = b0/a0;
        coefs[1] = b1/a0;
        coefs[2] = b2/a0;
        coefs[3] =  -a1/a0;
        coefs[4] =  -a2/a0;
}
```

```
#include <string.h>
#include <stdlib.h>
#include <assert.h>
#include <math.h>
#include "../../MachineInterface.h"
#include "../../auxbus/auxbus.h"

CMachineParameter const paraDummy = { pt_byte, "Dummy", "Dummy", 0, 127, 255, MPF_STATE, 0 };

CMachineParameter const *pParameters[] =
{
          // global
          &paraDummy
};


#pragma pack(1)

class gvals
{
public:
          byte dummy;
};


#pragma pack()

CMachineInfo const MacInfo =
{
          MT_EFFECT,                              // type
          MI_VERSION,
          0,                                                // flags
          0,                                                // min tracks
          0,                                                // max tracks
          1,                                                // numGlobalParameters
          0,                                                // numTrackParameters
          pParameters,
          0,
          NULL,
#ifdef _DEBUG
          "Jeskola Multiplier (Debug build)",               // name
#else
          "Jeskola Multiplier",                                               // name
#endif
          "Multiplier",                                     // short name
          "Oskari Tammelin",                                                  // author
          "Set Channel"
};


class mi : public CMachineInterface
{
public:
          mi();
          virtual ~mi();

          virtual void Init(CMachineDataInput * const pi);
          virtual void Tick();
          virtual bool Work(float *psamples, int numsamples, int const mode);

          virtual void Command(int const i);
          virtual void Save(CMachineDataOutput * const po);

          void DisconnectAux();

private:



private:
          int Channel;


          gvals gval;

};

DLL_EXPORTS

mi::mi()
{
          GlobalVals = &gval;
}

mi::~mi()
{
          AB_Disconnect(this);
}

void cb(void *user)
{
          mi *pmi = (mi *)user;
          pmi->DisconnectAux();
}


#define VERSION                1

void mi::Init(CMachineDataInput * const pi)
{
          if (pi != NULL)
```

```
	{
		byte ver;
		pi->Read(ver);
		if (ver == VERSION)
		{
			pi->Read(Channel);

			if (Channel != -1)
				AB_ConnectOutput(Channel, MacIn fo.ShortName, cb, this);
		}

	}
	else
	{
		Channel = -1;
	}
}

void mi::Save(CMachineDataOutput * const po)
{
	po->Write((byte)VERSION);
	po->Write(Channel);
}


void mi::Tick()
{
}


bool mi::Work(float *psamples, int numsamples, int const mode)
{
	if (Channel != -1 && mode & WM_READ)
	{
		float *paux = pCB ->GetAuxBuffer();		// note: AuxBuffer and AuxBus are not related in any way

		AB_Receive(Channel, paux, numsamples);

		do
		{

			double i = *psamples * (1.0 / 32768.0);
			double a = *paux++ * (1.0  / 32868.0);

			double o = i * a;

			if (o < -1.0)
				o = -1.0;
			else if (o > 1.0)
				o = 1.0;

			*psamples++ = (float)(o * 32768.0);

		} while(--numsamples);

		return true;
	}
	else
	{
		// no aux connected or no input, pass signal thru unmo dified
		return mode & WM_READ;
	}

}



void mi::DisconnectAux()
{
	MACHINE_LOCK;

	Channel = -1;
}


void mi::Command(int const i)
{
	if (i == 0)
		AB_ShowEditor(NULL, &Channel, MacInfo.ShortName, cb, this);
}
```

```
#include <string.h>
#include <stdlib.h>
#include <assert.h>
#include <math.h>
#include "../MachineInterface.h"
#include "../nr_lib/nr_lib.h"

CMachineParameter const paraDetail =
{
          pt_byte,                                  // type
          "Detail",
          "Detail",                                 // description
          1,                                                            // MinValue
          16,                                                 // MaxValue
          0,                                                            // NoValue
          MPF_STATE,                                          // Flags
          1
};

CMachineParameter const paraSeekRate =
{
          pt_word,                                  // type
          "Rate",
          "Change Rate (half life in ms)",                              // description
          1,                                                            // MinValue
          0xFFFF,                                             // MaxValue
          0,                                                            // NoValue
          MPF_STATE,                                          // Flags
          125
};

CMachineParameter const paraInit =
{
          pt_word,                                  // type
          "Initial",
          "Level [initialise] (0=-200%, 8000=0%, FFFE=~200%)",         // description
          0,                                                            // MinValue
          0xfffe,                                             // MaxValue
          0xffff,                                             // NoValue
          MPF_STATE,                                          // Flags
          0x8000
};

CMachineParameter const paraTarget =
{
          pt_word,                                  // type
          "Target",
          "Level [target] (0=-200%, 8000=0%, FFFE=~200%)",   // description
          0,                                                            // MinValue
          0xfffe,                                             // MaxValue
          0xffff,                                             // NoValue
          MPF_STATE,                                          // Flags
          0x8000
};

CMachineParameter const *pParameters[] =
{
          // global
          &paraDetail,
          &paraSeekRate,
          &paraInit,
          &paraTarget
};

#pragma pack(1)

class gvals
{
public:
          byte detail;
          word seek_rate;
          word initial;
          word target;
};

#pragma pack()

CMachineInfo const MacInfo =
{
          MT_EFFECT,                      // type
          MI_VERSION,
          0,                                        // flags
          0,                                        // min tracks
          0,                                        // max tracks
          4,                                        // numGlobalParameters
          0,                                        // numTrackParameters
          pParameters,
          0,
          NULL,
#ifdef _DEBUG
          "Ninereeds Discretize (Debug build)",     // name
#else
          "Ninereeds Discretize",                                       // name
#endif
          "Discrete",                     // short name
          "Steve Horne",         // author
          NULL
};


class mi : public CMachineInterface
```

```
{
public:
          mi();
          virtual ~mi();

          virtual void Init(CMac hineDataInput * const pi);
          virtual void Tick();
          virtual bool Work(float *psamples, int numsamples, int const mode);

private:



private:
          int   Detail;

          c_Decay_Simple *f_Decay;

//        float AttackA;
//        float AttackB;

//        float Target;
//        float Current;

          gvals gval;

};

DLL_EXPORTS

mi::mi()
{
          GlobalVals = &gval;
          f_Decay = NULL;
//        AttrVals = (int *)&aval;
}

mi::~mi()
{
          if (f_Decay != NULL)  {  delete f_Decay;  }
}

void mi::Init(CMachineDataInput * const pi)
{
          Detail = 1;

          f_Decay = new c_Dec ay_Simple (pMasterInfo ->SamplesPerSec);

          f_Decay->Tick (125, 0, 0);

//        AttackA = pow(2.0,  -1000.0 / (((double) pMasterInfo ->SamplesPerSec) * 125.0));
//        AttackB = 1.0  - AttackA;

//        Target  = 0.0;
//        Current = 0.0;
}

void mi::Tick()
{
          if (gval.detail != paraDetail.NoValue)
          {
                    Detail = gval.detail;
          }

          f_Decay->Tick (gval.seek_rate, gval.target, gval.initial);
/*
          if (gval.seek_rate != paraSeekRate.NoValue)
          {
                    AttackA = pow(2.0,  -1000.0 / (((double) pMasterInfo ->SamplesPerSec) * ((float) gval.seek_ rate)));
                    AttackB = 1.0  - AttackA;
          }

          if (gval.initial != paraInit.NoValue)
          {
                    Current = ((float) (gval.initial  - 0x8000)) / 16384.0;
                    Target  = Current;
          }

          if (gval.target != paraTarget.NoValue)
          {
                    Target = ((float) (gval.target  - 0x8000)) / 163 84.0;
          }
*/
}


bool mi::Work(float *psamples, int numsamples, int const mode)
{
          if (mode == WM_WRITE || mode == WM_NOIO)
                    return false;

          if (mode == WM_READ)
                    return true;

          float Tmp;
          float Tmp2;
          float Tmp3;

          Tmp3 = ((float) (Detail + Detail));

  if (f_Decay->Is_Static ())
  {
    float l_Level = f_Decay ->Current_Level ();
    if (l_Level != 1.0)
    {
//      l_Level -= 1.0;

      do
      {
        Tmp = ((((*psamples) / 65536.0) + 0.5) * Tmp3) + 0.5;

        Tmp = (float) ((int) Tmp);

        Tmp = ((Tmp / Tmp3) - 0.5) * 65536.0;
        Tmp2 = (*psamples) - Tmp;

        (*psamples) += (Tmp2 * l_Level);

        psamples++;

      } while (--numsamples);
    }
  }
  else
  {
          float *l_Aux_Buf = pCB->GetAuxBuffer ();

    f_Decay->Overwrite (l_Aux_Buf, numsamples);

    do
    {
      Tmp = ((((*psamples) / 65536.0) + 0.5) * Tmp3) + 0.5;

      Tmp = (float) ((int) Tmp);

      Tmp = ((Tmp / Tmp3) - 0.5) * 65536.0;
      Tmp2 = (*psamples) - Tmp;

//      (*psamples) += (Tmp2 * (*l_Aux _Buf - 1.0));
      (*psamples) += (Tmp2 * (*l_Aux_Buf));

      psamples++;
      l_Aux_Buf++;

    } while (--numsamples);
  }

          return true;
}
```

```
#include <string.h>
#include <stdlib.h>
#include <assert.h>
#include <math.h>
#include "../MachineInterface.h"

#include "../nr_lib/nr_lib.h"


NR_LIB_ATTACK_PAR     (paraSeekRate, "Fade Speed", "Fade Speed (half life in ms)")
NR_LIB_AMPLITUDE16_PAR(paraInit,     "Initial",    "Volume [initialise] (0=0%, 8000=100%, FFFE=~200%)")
NR_LIB_AMPLITUDE16_PAR(paraTarget,   "Target",     "Volume [target]     (0=0%, 8000=100%, FFFE=~200%)")


CMachineParameter const *pParameters[] =
{
          // global
          &paraSeekRate,
          &paraInit,
          &paraTarget
};

#pragma pack(1)

class gvals
{
public:
          word seek_rate;
          word initial;
          word target;
};

#pragma pack()

CMachineInfo const MacInfo =
{
          MT_EFFECT,                              // type
          MI_VERSION,
          0,                                                // flags
          0,                                                // min tracks
          0,                                                // max tracks
          3,                                                // numGlobalParameters
          0,                                                // numTrackParameters
          pParameters,
          0,
          NULL,
#ifdef _DEBUG
          "Ninereeds Fade v1.0(Debug build)",               // name
#else
          "Ninereeds Fade v1.0",                                              // name
#endif
          "Fade",                       // short name
          "Steve Horne",                // author
          NULL
};


class mi : public CMachineInterface
{
public:
          mi();
          virtual ~mi();

          virtual void Init(CMachineDataInput * const pi);
          virtual void Tick();
          virtual bool Work(float *psamples, int numsamples, int const mode);

private:



private:
    c_Decay_Simple  *f_Level;

          gvals gval;

};

DLL_EXPORTS

mi::mi()
{
          f_Level = NULL;

          GlobalVals = &gval;
//        AttrVals = (int *)&aval;
}

mi::~mi()
{
          if (f_Level != NULL)  {  delete f_Level;  }
}

void mi::Init(CMachineDataInput * const pi)
{
          f_Level = new c_Decay_Simple (pMasterInfo ->SamplesPerSec);
}

void mi::Tick()
{
          f_Level->Tick (gval.seek_rate, gval.target, gval.initial);
}


bool mi::Work(float *psamples, int numsamples, int const mode)
{
          if ((mode & WM_READ) == 0)  {  return false;  }

          if (mode == WM_READ)
                    return true;

          return f_Level->Multiply (psamples, numsamples);
}
```

```
//
//  To Do...
//
//    - Broadcast Rx
//      - Handle detune in twin note command
//      - Update property sheet (if visible) after command received
//
//    - Envelopes
//      - Allow a softy-like envelope to be configured
//      - Allow envelope to vary with note pitch  - eg longer decay for low notes
//
//    - Filters
//      - Add a property sheet page for filter configuration
//      - Implement butterworth filters
//      - Add a filter configuration command
//      - Add filter parameters to program data
//
//    - Arpegiateur
//      - ?
//

#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include <windows.h>
#include <commctrl.h>

#include "../dsplib/dsplib.h"
#include "../dsplib/auxbus.h"
#include "../Ninereeds Broadcast Lib/Ninereeds Broadcast Lib.h"
#include "../nr_lib/nr_lib.h"

#include "resource.h"

#pragma optimize ("a", on)

#define MAX_TRACKS   8

float const oolog2 = 1.0 / log(2);

NR_LIB_NOTE_PAR                  (paraNoteInit,    "Note",                        "Note"
                               )
NR_LIB_NOTE_PAR                  (paraNote,                  "Bend Note",         "Bend Note"
                               )
NR_LIB_AMPLITUDE_PAR   (paraVolume,                "Volume",            "Volume (0=0%, 80=100%, FE=~200%)", 0x80)


CMachineParameter const *pParameters[] =
{
          // global

          // track
          &paraNoteInit,
          &paraNote,
          &paraVolume
};

#pragma pack(1)

//class gvals
//{
//public:
//        word bend;
//};

class tvals
{
public:
          byte noteinit;
          byte note;
          byte volume;
};

#pragma pack()

CMachineInfo const MacInfo =
{
          MT_GENERATOR, MI_VERSION, 0,                                // type, version, flags
          1, MAX_TRACKS,                           // min, max tracks
          0, 3, pParameters,             // num globalpars, num trackpars, *pars
          0,    NULL,                                      // num attribs, *attribs
          "Ninereeds NRS04", "NRS04", "Steve Horne",         // name, short name, author
          "Edit Ninereeds NRS04"                                                          // command menu
};

class mi;

class CTrack
{
public:
          void Tick(tvals const &tv);
          void Stop();
          void Reset();
          bool Generate(float *psamples, int numsamples);

public:

          c_ADSR_Track   *f_Level;
          c_ADSR2_Track  *f_Mod_Level;
          c_Pitch_Track  *f_Pitch;
          c_Frq_To_Phase *f_Frq_To _Phase;
          c_Frq_To_Phase *f_Frq_To_Phase2;
          c_Waveform     *f_Waveform;  //  This just duplicates the pointer from mi::f_Waveform
          c_Fractal      *f_Fractal;   //  This just duplicates the pointer from mi::f_Fractal

          c_Frq_To_Phase *f_Mod_Frq_To_Phase;
          c_Waveform     *f_Mod_Waveform;
          c_Fractal      *f_Mod_Fractal;

          float f_Mod_Buffer [MAX_BUFFER_LENGTH];  //  Used for modulator

          float f_Buffer  [MAX_BUFFER_LENGTH];  //  Used for amplitude
          float f_Buffer2 [MAX_BUFFER_LENGTH];  //  Used for modulator env  elope and twin note

          CTrack (c_ADSR_Global  *p_Level_Global,
          c_ADSR2_Global *p_Mod_Level_Global,
          c_Pitch_Global *p_Pitch_Global     )
          {
                    f_Level            = new c_ADSR_Track  (*p_Level_Global);
                    f_Mod_Level        = new c_ADSR2_Tra ck (*p_Mod_Level_Global);
                    f_Pitch            = new c_Pitch_Track (*p_Pitch_Global);
                    f_Frq_To_Phase     = new c_Frq_To_Phase;
                    f_Frq_To_Phase2    = new c_Frq_To_Phase;
    f_Mod_Frq_To_Phase = new c_Frq_To_Phase;

                    f_Level->Set_Volume (1.0);  //  Def ault note amplitude
          }

          ~CTrack (void)
          {
                    delete f_Level;
                    delete f_Mod_Level;
                    delete f_Pitch;
                    delete f_Frq_To_Phase;
                    delete f_Frq_To_Phase2;
          }

          mi *pmi;
};

#define MAX_PROGRAM_NAME 32
#define MAX_PROGRAMS     32
#define MAX_PROGRAM_IDS  32

struct c_Program
{
          char  f_Name [MAX_PROGRAM_NAME];

          byte  f_Waveform;
          byte  f_Depth;         float f_Effect_Hi;       float f_Effect_Lo;
          byte  f_Mod_Waveform;
          byte  f_Mod_Depth;         float f_Mod_Effect_Hi;       float f_Mod_Effect_Lo;
          float f_Atta ck;        float f_Decay;           float f_Sustain;           float f_Release;
          float f_Bend;
          bool  f_Enable_Twin;   bool  f_Fractal_Before;  float f_Detune_Semitones;  float f_Twin_Amp;
          int   f_Channel;
          bool  f_Ringmod_Main;  bool  f_Ringmod_Twin;
          bool  f_FM_Enable;     float f_FM_Level;   bool f_FM_Internal;
          bool  f_AM_Enable;     float f_AM_Level;   bool f_AM_Internal;
          bool  f_PWM_Enable;    float f_PWM_Level;  bool f_PWM_Internal;
  int   f_Mod_Frq_Type;  float f_Mod_Frq;

          float f_Mod_Attack;    float f_Mod_Decay;       float f_Mod_Sustain;       float f_Mod_Release;
          float f_Mod_Start;     float f_Mod_Peak;        float f_Mod_End;


          void Get_From_Current (mi *pmi);
          void Set_As_Current   (mi *pmi);

          static int Find (mi *pmi, char *p_Name);

          static void Fill_Program_Combo (mi *pmi, HWND hDlg);
          static void Fill_ID_Combo      (mi *pmi, HWND hDlg, int f_Curr_Program);
          static void Fill_ID_List       (mi *pmi, HWND hDlg);
};

struct c_V2_Program
{
          char  f_Name [MAX_PROGRAM_NAME];

          byte  f_Wavefo rm;
          byte  f_Depth;         float f_Effect_Hi;       float f_Effect_Lo;
          byte  f_Mod_Waveform;
          byte  f_Mod_Depth;         float f_Mod_Effect_Hi;       float f_Mod_Effect_Lo;
          float f_Attack;        float f_Decay;           float f_Sustain;           float   f_Release;
          float f_Bend;
          bool  f_Enable_Twin;   bool  f_Fractal_Before;  float f_Detune_Semitones;  float f_Twin_Amp;
          int   f_Channel;
          bool  f_Ringmod_Main;  bool  f_Ringmod_Twin;
          bool  f_FM_Enable;     float f_FM_Level;   bool f_FM_Internal;
          bool  f_AM_Enable;     float f_AM_Level;   bool f_AM_Internal;
          bool  f_PWM_Enable;    float f_PWM_Level;  bool f_PWM_Internal;
  int   f_Mod_Frq_Type;  float f_Mod_Frq;

  void Copy_To_Current (c_Program &p);
};

void c_V2_Program::Copy_To_Current (c_Program &p )
{
  memcpy (p.f_Name, f_Name, MAX_PROGRAM_NAME);

          p.f_Waveform = f_Waveform;
          p.f_Depth = f_Depth;
  p.f_Effect_Hi = f_Effect_Hi;
  p.f_Effect_Lo = f_Effect_Lo;
```

```
          p.f_Attack = f_Attack;
  p.f_Decay = f_Decay;
  p.f_Sustain = f_Sustain;
  p.f_Release = f_Release;
          p.f_Bend = f_Bend;
          p.f_Enable_Twin = f_Enable_Twin;
  p.f_Fractal_Before = f_Fractal_Before;
  p.f_Detune_Semitones = f_Detune_Semitones;
  p.f_Twin_Amp = f_Twin_Amp;
          p.f_Channel = f_Channel;
          p.f_Ringmod_Main = f_Ringmod_Main;
  p.f_Ringmod_Twin = f_Ringmod_Twin;
          p.f_FM_Enable = f_FM_Enable;
  p.f_FM_Level = f_FM_Level;
          p.f_AM_Enable = f_AM_Enable;
  p.f_AM_Level = f_AM_Level;
          p.f_PWM_Enable = f_PWM_Enable;
  p.f_PWM_Level = f_PWM_Level;

          p.f_Mod_Waveform = f_Mod_Waveform;
          p.f_Mod_Depth = f_Mod_Depth;
  p.f_Mod_Effect_Hi = f_Mod_Effect_Hi;
  p.f_Mod_Effect_Lo = f_Mod_Effect_Lo;
  p.f_FM_Internal = f_FM_Internal;
  p.f_AM_Internal = f_AM_Internal;
  p.f_PWM_Internal = f_PWM_Internal;
  p.f_Mod_Frq_Type = f_Mod_Frq_Type;
  p.f_Mod_Frq = f_Mod_Frq;

          p.f_Mod_Attack  = 0.2;  //  These defaults for back compatability rather than usefulness
  p.f_Mod_Decay   = 0.2;
  p.f_Mod_Sustain = 1.0;
  p.f_Mod_Release = 0.2;
          p.f_Mod_Start   = 1.0;
  p.f_Mod_Peak    = 1.0;
  p.f_Mod_End     = 1.0;
}

struct c_V1_Program
{
          char  f_Name [MAX_PROGRAM_NAME];

          byte  f_Waveform;
          byte  f_Depth;        float f_Effect_Hi;       float f_Effect_Lo;
          float f_Attack;       float f_Decay;           float f_Sustain;          float f_Release;
          float f_Bend;
          bool  f_Enable_Twin;   bool  f_Fractal_Before;  float f_Detune_Semitones;  float f_Twin_Amp;
          int   f_Channel;
          bool  f_Ringmod_Main;  bool  f_Ringmod_Twin;
          bool  f_FM_Enable;     float f_FM_Level;
          bool  f_AM_Enable;     float f_AM_Level;
          bool  f_PWM_Enable;    float f_PWM_Level;

  void Copy_To_Current (c_Program &p);
};

void c_V1_Program::Copy_To_Current (c_Program &p)
{
  memcpy (p.f_Name, f_Name, MAX_PROGRAM_NAME);

          p.f_Waveform = f_Waveform;
          p.f_Depth = f_Depth;
  p.f_Effect_Hi = f_Effect_Hi;
  p.f_Effect_Lo = f_Effect_Lo;
          p.f_Attack = f_Attack;
  p.f_Decay = f_Decay;
  p.f_Sustain = f_Sustain;
  p.f_Release = f_Release;
          p.f_Bend = f_Bend;
          p.f_Enable_Twin = f_Enable_Twin;
  p.f_Fractal_Before = f_Fractal_Before;
  p.f_Detune_Semitones = f_Detune_Semitones;
  p.f_Twin_Amp = f_Twin_Amp;
          p.f_Channel = f_Channel;
          p.f_Ringmod_Main = f_Ringmod_Main;
  p.f_Ringmod_Twin = f_Ringmod_Twin;
          p.f_FM_Enable = f_FM_Enable;
  p.f_FM_Level = f_FM_Level;
          p.f_AM_Enable = f_AM_Enable;
  p.f_AM_Level = f_AM_Level;
          p.f_PWM_Enable = f_PWM_Enable;
  p.f_PWM_Level = f_PWM_Level;

          p.f_Mod_Waveform = 0;
          p.f_Mod_Depth = 0;
  p.f_Mod_Effect_Hi = 0.0;
  p.f_Mod_Effect_Lo = 0.0;
  p.f_FM_Internal = false;
  p.f_AM_Internal = false;
  p.f_PWM_Internal = false;
  p.f_Mod_Frq_Type =  0;
  p.f_Mod_Frq = 0.0;

          p.f_Mod_Attack  = 0.2;
  p.f_Mod_Decay   = 0.2;
  p.f_Mod_Sustain = 1.0;
  p.f_Mod_Release = 0.2;
          p.f_Mod_Start   = 1.0;
  p.f_Mod_Peak    = 1.0;
  p.f_Mod_End     = 1.0;
}


//
```

```
//  Note multiple inheritence  - easiest way to handle listener
//

class mi : public CMachineInterface, public c_Broadcast_Listener_CB
{
public:
          mi ();
          virtual ~mi();

          virtual void Init(CMachineDataInput  * const pi);
          virtual void Save(CMachineDataOutput * const po);

          virtual void Tick();
          virtual bool Work(float *psamples, int numsamples, int const mode);
          virtual void SetNumTracks(int const n);
          virtual void Stop();

  virtual void Command(int const i);

          virtual void Transpose_Notes  (int p_Semitones);

          virtual void All_Notes_Off    (void);
          virtual void All_Notes_Ignore (void);

  virtual void Set_Control           (int p_ID, int p_Value);


          virtual void Tick2 (void);
          virtual void Tick2 (int p_Channel, c_Broadcast_Params *p_Command);
          virtual bool Work2 (float *psamples, int numsamples, int const mode, int offset);

          void DisconnectAux ();

public:
          //  Do the following all need to be pointers?

          c_ADSR_Global  *f_Level;
          c_ADSR2_Global *f_Mod_Level;
          c_Pitch_Global *f_Pitch;
          c_Waveform     *f_Waveform;
          c_Fractal      *f_Fractal;
          c_Waveform     *f_Mod_Waveform;
          c_Fractal      *f_Mod_Fractal;

          //  Twin note parameters

          bool  f_Enable_Twin;
          bool  f_Fractal_Before;
          float f_Detune_Semitones;
          float f_Twin_Amp;

          int   f_Channel;  //  Auxbus Connection Channel

          bool  f_Listen_Enabled;  //  For Broadcast Commands
          int   f_Listen_Channel;

          bool  f_Ringmod_Main;
          bool  f_Ringmod_Twin;
          bool  f_FM_Enable;
          bool  f_AM_Enable;
          bool  f_PWM_Enable;
          bool  f_FM_Internal;
          bool  f_AM_Internal;
          bool  f_PWM_Internal;
          float f_FM_Level;
          float f_AM_Level;
          float f_PWM_Level;
  int   f_Mod_Frq_Type;
  float f_Mod_Frq;

          int       f_Num_Programs;
          c_Program f_Programs     [MAX_PROGRAMS];
          int       f_Program_IDs  [MAX_PROGRAM_IDS];  //  Max program ID to f_Programs index

          float  f_Aux_Buf_Full [MAX_BUFFER_LENGTH];  //  Used for AuxBus input
          float *f_Aux_Buffer;

          int numTracks;
          CTrack *Tracks[MAX_TRACKS];

          tvals tval     [MAX_TRACKS];
//        gvals gval;

};

void c_Program::Get_From_Current (mi *pmi)
{
          f_Waveform  = pmi->f_Waveform->Get_Waveform ();

          f_Depth     = pmi->f_Fractal->Get_Depth     ();
          f_Effect_Hi = pmi->f_Fractal->Get_Effect_Hi ();
          f_Effect_Lo = pmi->f_Fractal->Get_Effect_Lo ();

          f_Mod_Waveform  = pmi->f_Mod_Waveform->Get_Waveform ();

          f_Mod_Depth     = pmi->f_Mod_Fractal->Get_Depth     ();
          f_Mod_Effect_Hi = pmi->f_Mod_Fractal->Get_Effect_Hi ();
          f_Mod_Effect_Lo = pmi->f_Mod_Fractal->Get_Effect_Lo ();

  f_Mod_Frq_Type  = pmi->f_Mod_Frq_Type;
  f_Mod_Frq       = pmi->f_Mod_Frq;

          f_Attack    = pmi->f_Level->Get_Attack  ();
          f_Decay     = pmi->f_Level->Get_Decay   ();
          f_Sustain   = pmi->f_Level->Get_Sustain ();
          f_Release   = pmi->f_Level->Get_Release ();

          f_Mod_Attack    = pmi->f_Mod_Level->Get_Attack  ();
          f_Mod_Decay     = pmi->f_Mod_Level->Get_Decay   ();
          f_Mod_Sustain   = pmi->f_Mod_Level->Get_Sustain ();
          f_Mod_Release   = pmi->f_Mod_Level->Get_Release ();
          f_Mod_Start     = pmi->f_Mod_Level->Get_Start   ();
          f_Mod_Peak      = pmi->f_Mod_Level->Get_Peak    ();
          f_Mod_End       = pmi->f_Mod_Level->Get_End     ();

          f_Bend      = pmi->f_Pitch->Get_Attack  ();

          f_Enable_Twin      = pmi->f_Enable_Twin;
          f_Fractal_Before   = pmi->f_Fractal_Before;
          f_Detune_Semitones = pmi->f_Detune_Semitones;
          f_Twin_Amp         = pmi->f_Twin_Amp;

          f_Channel          = pmi->f_Channel;

          f_Ringmod_Main      = pmi->f_Ringmod_Main;
          f_Ringmod_Twin     = pmi->f_Ringmod_Twin;

          f_FM_Enable        = pmi->f_FM_Enable;
          f_FM_Internal      = pmi->f_FM_Internal;
          f_FM_Level         = pmi->f_FM_Level;

          f_AM_Enable        = pmi->f_AM_Enable;
          f_AM_Internal      = pm i->f_AM_Internal;
          f_AM_Level         = pmi->f_AM_Level;

          f_PWM_Enable       = pmi->f_PWM_Enable;
          f_PWM_Internal     = pmi->f_PWM_Internal;
          f_PWM_Level        = pmi->f_PWM_Level;
}

void c_Program::Set_As_Current   (mi *pmi)
{
          pmi->f_Waveform->Set_Waveform (f_Waveform);

          pmi->f_Fractal->Set_Depth     (f_Depth);
          pmi->f_Fractal->Set_Effect_Hi (f_Effect_Hi);
          pmi->f_Fractal->Set_Effect_Lo (f_Effect_Lo);

          pmi->f_Mod_Waveform->Set_Waveform (f_Mod_Waveform);

          pmi->f_Mod_Fractal->Set_Depth     (f_Mod_Depth );
          pmi->f_Mod_Fractal->Set_Effect_Hi (f_Mod_Effect_Hi);
          pmi->f_Mod_Fractal->Set_Effect_Lo (f_Mod_Effect_Lo);

  pmi->f_Mod_Frq_Type  = f_Mod_Frq_Type;
  pmi->f_Mod_Frq       = f_Mod_Frq;

          pmi->f_Level->Set_Attack  (f_Attack);
          pmi->f_Level->Set_Decay   (f_Decay);
          pmi->f_Level->Set_Sustain (f_Sustain);
          pmi->f_Level->Set_Release (f_Release);

          pmi->f_Mod_Level->Set_Attack  (f_Mod_Attack);
          pmi->f_Mod_Level->Set_Decay   (f_Mod_Decay);
          pmi->f_Mod_Level->Set_Sustain (f_Mod_Sustain);
          pmi->f_Mod_Level->Set_Release (f_Mod_Release);
          pmi->f_Mod_Level->Set_Start   (f_Mod_Start);
          pmi->f_Mod_Level->Set_Peak    (f_Mod_Peak);
          pmi->f_Mod_Level->Set_End     (f_Mod_End);

          pmi->f_Pitch->Set_Attack  (f_Bend);

          pmi->f_Enable_Twin      = f_Enable_Twin;
          pmi->f_Fractal_Before   = f_Fractal_Before;
          pmi->f_Detune_Semitones = f_Detune_Semitones;
          pmi->f_Twin_Amp         = f_Twin_Amp;

          pmi->f_Channel          = f_Channel;

          pmi->f_Ringmod_Main     = f_Ringmod_Main;
          pmi->f_Ringmod_Twin     = f_Ringmod_Twin;

          pmi->f_FM_Enable        = f_FM_Enable;
          pmi->f_FM_Internal      = f_FM_Internal;
          pmi->f_FM_Level         = f_FM_Level;

          pmi->f_AM_Enable        = f_AM_Enable;
          pmi->f_AM_Internal      = f_AM_Internal;
          pmi->f_AM_Level         = f_AM_Level;

          pmi->f_PWM_Enable        = f_PWM_Enable;
          pmi->f_PWM_Internal     = f_PWM_Internal;
          pmi->f_PWM_Level        = f_PWM_Level;
}

int c_Program::Find (mi *pmi, char *p_Name)
{
          for (int i = 0; i < pmi->f_Num_Programs; i++)
          {
                    if (strcmp (p_Name, pmi->f_Programs [i].f_Name) == 0)
                    {
                              return i;
                    }
          }

          return -1;
}
```

```
void c_Program::Fill_Program_Combo (mi *pmi, HWND hDlg)
{
          SendMessage (GetDlgItem (hDlg, IDC_COMBO_PROG_NAME), CB_RESETCONTENT, 0, 0L);

          for (int i = 0; i < pmi ->f_Num_Programs; i++)
          {
                    SendMessage (GetDlgItem (hDl g, IDC_COMBO_PROG_NAME),
                                                       CB_ADDSTRING, 0, (LONG) (LPSTR) pmi ->f_Programs[i].f_Name);
          }
}

void c_Program::Fill_ID_Combo (mi *pmi, HWND hDlg, int f_Curr_Program)
{
          char l_Buf [80];

          SendMessage (GetDlgItem (hDlg, IDC_COMBO_PRG_IDS), CB_RESETCONTENT  , 0, 0L);

          for (int i = 0; i < MAX_PROGRAM_IDS; i++)
          {
                    if (pmi->f_Program_IDs [i] == f_Curr_Program)
                    {
                              sprintf (l_Buf, "%d", i);

                              SendMessage (GetDlgItem (hDlg, IDC_COMBO_PRG_IDS),
                                                                 CB_ADDSTRING, 0, (LONG) (LPSTR) l_Buf);
                    }
          }
}

void c_Program::Fill_ID_List (mi *pmi, HWND hDlg)
{
          char l_Buf [80];

          ListView_DeleteAllItems (GetDlgItem (hDlg, IDC_LIST_PROG_IDS));

          LV_ITEM l_Item;

          l_Item.mask       = LVIF_TEXT | LVIF_PARAM;
          l_Item.iItem      = 0;
          l_Item.iSubItem   = 0;
          l_Item.state       = 0;
          l_Item.stateMask  = 0;
          l_Item.pszText    = NULL;
          l_Item.cchTextMax = 32;
          l_Item.iImage     = 0;
          l_Item.lParam     = 0;

          for (int i = 0; i < MAX_PROGRAM_IDS; i++)
          {
                    sprintf (l_Buf, "%d", i);

                    l_Item.iItem    = i;
                    l_Item.lParam   = i;
                    l_Item.pszText  = l_Buf;
                    l_Item.iSubItem = 0;

                    ListView_InsertItem (GetDlgItem (hDlg, IDC_LIST_PROG_IDS), &l_Item);

                    l_Item.iItem    = i;
                    l_Item.lParam   = i;
                    l_Item.pszText  = pmi ->f_Programs [pmi ->f_Program_IDs [i]].f_Name;
                    l_Item.iSubItem = 1;

                    ListView_InsertItem (GetDlgItem (hDlg, IDC_LIST_PROG_IDS), &l_Item);
          }
}


DLL_EXPORTS
NR_STD_GENERATOR_STOP

void mi::SetNumTracks(int const n)
{
          for (int i = numTracks; i < n; i++)
          {
                    Tracks[i]->Reset ();
          }

          numTracks = n;
}


//  Note that although Work itself exists mainly to redirect things to the
//  broadcast library stuff, it can do some things itself  - in this case,
//  it retrieves the AuxBus input (if needed).
//
//  Work2, however, does the real signal generation.

bool mi::Work (f loat *psamples, int numsamples, int const mode)
{
          if (((mode & WM_WRITE) != 0) && (f_Channel !=  -1))
          {
                    AB_Receive (f_Channel, f_Aux_Buf_Full, numsamples);
          }

          return Handle_Work (psamples, numsamples, mode);
}


bool mi::Work2(float *psamples, int num samples, int const mode, int offset)
{
//        if ((mode & WM_WRITE) == 0)
//        {
//                  return false;
//        }

          bool gotsomething = false;
          bool l_Temp;

          f_Aux_Buffer = f_Aux_Buf_Full + offset;

          float *paux = pCB ->GetAuxBuffer();

          for (int c = 0; c < numTracks; c++ )
          {
                    if (gotsomething)
                    {
                              l_Temp = Tracks[c] ->Generate(paux, numsamples);

                              gotsomething |= l_Temp;

                              if (l_Temp)
                              {
                                        DSP_Add(psamples, paux, numsamples);
                              }
                    }
                    else
                    {
                              gotsomething |= Tracks[c] ->Generate(psamples, numsamples);
                    }
          }

          return gotsomething;
}

mi::mi()
{
          f_Level    = NULL;
          f_Mod_Level = NULL;
          f_Pitch    = NULL;
          f_Waveform = NULL;
          f_Fractal  = NULL;
          f_Mod_Waveform = NULL;
          f_Mod_Fractal  = NULL;

          f_Enable_Twin      = true;
          f_Fractal_Before   = true;
          f_Detune_Semitones = 0.1;
          f_Twin_Amp         = 1.0;

          f_Channel = -1;

          f_Listen_Enabled = false;
          f_Listen_Channel = 0;

          f_Ringmod_Main = false;
          f_Ringmod_Twin = false;
          f_FM_Enable    = false;
          f_FM_Internal  = true;
          f_FM_Level     = 0.0;
          f_AM_Enable    = fa lse;
          f_AM_Internal  = true;
          f_AM_Level     = 0.0;
          f_PWM_Enable   = false;
          f_PWM_Internal = true;
          f_PWM_Level    = 0.0;

  f_Mod_Frq_Type = 0;
  f_Mod_Frq      = 0.0;

          f_Num_Programs = 0;

          for (int i = 0; i < MAX_TRACKS; i++)
          {
                    Tracks [i] = NULL;
          }

          for (i = 0; i < MAX_PROGRAM_IDS; i++)
          {
                    f_Program_IDs [i] =  -1;
          }

//        GlobalVals = &gval;
          GlobalVals = NULL;
          TrackVals  = tval;
          AttrVals   = NULL;
}

mi::~mi()
{
          AB_Disconnect (this);

          if (f_Listen_Enabled)
          {
                    NR_Stop_Listening (f_Listen_Ch annel, this);
          }

          for (int i = 0; i < MAX_TRACKS; i++)
          {
                    delete Tracks [i];
          }

          delete f_Waveform;
          delete f_Pitch;
          delete f_Level;
          delete f_Mod_Level;
          delete f_Fractal;
          delete f_Mod_Waveform;
          delete f_Mod_Fractal;
}

void mi::DisconnectAux()
{
          MACHINE_LOCK;

          f_Channel = -1;
}

void cb(void *user)  //  Auxbus disconnection callback
{
          mi *pmi = (mi *)user;
          pmi->DisconnectAux();
}

void mi::Init(CMachineDataInput * const pi)
{
          f_Level     = new c_ADSR_Global;
          f_Mod_Level = new c_ADSR2_Global;
  f_Pitch     = new c_Pitch_Global (pMasterInfo ->SamplesPerSec);
          f_Waveform  = new c_Waveform;
          f_Fractal   = new c_Fractal;
          f_Mod_Waveform = new c_Waveform;
          f_Mod_Fractal  = new c_Fractal;

          f_Level->Set_Samples_Per_Sec     (pMasterInfo ->SamplesPerSec);
          f_Mod_Level->Set_Samples_Per_Sec (pMasterInfo ->SamplesPerSec);

  for (int i = 0; i < MAX_TRACKS; i++)
          {
                    Tracks [i] = new CTrack (f_Level, f_Mod_Level, f_Pitch);

                    Tracks [i]->pmi = this;
                    Tracks [i]->f_Waveform     = f_Waveform;
                    Tracks [i]->f_Fractal      = f_Fractal;
                    Tracks [i]->f_Mod_Waveform = f_Mod_Waveform;
                    Tracks [i]->f_Mod_Fractal  = f_Mod_Fractal;

                    Tracks [i]->Reset();
          }

          if (pi != NULL)
          {
                    word l_Version;
                    pi->Read (l_Version);

                    while (l_Version != 0x0000)
                    {
                              switch (l_Version)
                              {
                                        case 0x0101 :
                                        {
                                                  byte  l_Waveform;   pi ->Read (l_Waveform );  f_Waveform ->Set_Waveform (l_Waveform);

                                                  f_Waveform->Set_Optimise_Enable  (false);
                                                  f_Waveform->Set_Optimise_Quality (NR_LIB_WAVEFORM_MAX_QUALITY);

                                                  break;
                                        }
                                        case 0x0102 :
                                        {
                                                  byte  l_Waveform;   pi ->Read (l_Waveform );  f_Waveform ->Set_Waveform         (l_Waveform);
                                                  bool  l_Optimise;   pi ->Read (l_Optimise );  f_Waveform ->Set_Optimise_Enable  (l_Optimise);
                                                  int   l_Quality;    pi ->Read (l_Quality  );  f_Waveform ->Set_Optimise_Quality (l_Quality );

          break;
        }

                                        case 0x0201 :
                                        {
                                                  byte  l_Depth;      pi ->Read (l_Depth    );  f_Fractal ->Set_Depth     (l_Depth);
                                                  float l_Effect_Hi;  pi ->Read (l_Effect_Hi);  f_F ractal->Set_Effect_Hi (l_Effect_Hi);
                                                  float l_Effect_Lo;  pi ->Read (l_Effect_Lo);  f_Fractal ->Set_Effect_Lo (l_Effect_Lo);

                                                  f_Fractal->Set_Optimise_Enable  (false);
                                                  f_Fractal->Set_Optimise_Quality (NR_LIB_FRACTAL_MAX_QUALITY);

                                                  break;
                                        }
                                        case 0x0202 :
                                        {
                                                  byte  l_Depth;      pi ->Read (l_Depth    );  f_Fractal ->Set_Depth     (l_Depth);
                                                  float l_Effect_Hi;  pi ->Read (l_Effect_Hi);  f_Fractal ->Set_Effect_Hi (l_Effect_Hi);
                                                  float l_Effect_Lo;  pi ->Read (l_Effect_Lo);  f_ Fractal->Set_Effect_Lo (l_Effect_Lo);
                                                  bool  l_Optimise;   pi ->Read (l_Optimise );  f_Fractal ->Set_Optimise_Enable  (l_Optimise);
                                                  int   l_Quality;    pi ->Read (l_Quality  );  f_Fractal ->Set_Optimise_Quality (l_Quality );

                                                  break;
                                        }
                                        case 0x0301 :
                                        {
                                                  float l_Attack;     pi ->Read (l_Attack   );  f_Level ->Set_Attack      (l_Attack);
                                                  float l_Decay;      pi ->Read (l_Decay    );  f_Level ->Set_Decay       (l_Decay);
                                                  float l_Sustain;    pi ->Read (l_Sustain  );  f_Level ->Set_Sustain     (l_Sustain);
                                                  float l_Release;    pi ->Read (l_Release  );  f_Level ->Set_Release     (l_Release);

                                                  break;
                                        }
                                        case 0x0401 :
                                        {
                                                  float l_Bend;       pi ->Read (l_Bend     );  f_Pitch ->Set_Attack      (l_Bend);

                                                  break;
                                        }
                                        case 0x0501 :
                                        {
                                                  pi->Read (f_Enable_Twin     );
                                                  pi->Read (f_Fractal_Before  );
                                                  pi->Read (f_Detune_Semitones);
                                                  f_Twin_Amp = 1.0;

                                                  break;
                                        }
                                        case 0x0502 :
                                        {
                                                  pi->Read (f_Enable_Twin     );
                                                  pi->Read (f_Fractal_Before  );
                                                  pi->Read (f_Detune_Semitones);
                                                  pi->Read (f_Twin_Amp        );

                                                  break;
                                        }
                                        case 0x0601 :
                                        {
                                                  pi->Read (f_Channel         );
                                                  pi->Read (f_Ringmod_Main       );
                                                  pi->Read (f_Ringmod_Twin       );
                                                  pi->Read (f_FM_Level        );

                                                  f_FM_Enable = (f_FM_Level > 0.0);

                                                  f_AM_Enable = false;
                                                  f_AM_Level  = 0.25;

                                                  f_PWM_Enable = false;
                                                  f_PWM_Level  = 0.25;

         f_FM_Internal  = false;
         f_AM_Internal  = false;
         f_PWM_Internal = false;

                                                  break;
                                        }
                                        case 0x0602 :
                                        {
                                                  pi->Read (f_Channel         );
                                                  pi->Read (f_Ringmod_Main       );
                                                  pi->Read (f_Ringmod_Twin       );
                                                  pi->Read (f_FM_Enable       );
                                                  pi->Read (f_FM_Level        );

                                                  f_AM_Enable = false;
                                                  f_AM_Level  = 0.25;

                                                  f_PWM_Enable = false;
                                                  f_PWM_Level  = 0.25;

         f_FM_Internal  = false;
         f_AM_Internal  = false;
         f_PWM_Internal = false;

                                                  break;
                                        }
                                        case 0x0603 :
                                        {
                                                  pi->Read (f_Channel         );
                                                  pi->Read (f_Ringmod_Main       );
                                                  pi->Read (f_Ringmod_Twin       );
                                                  pi->Read (f_FM_Enable       );
                                                  pi->Read (f_FM_Level        );
                                                  pi->Read (f_AM_Enable       );
                                                  pi->Read (f_AM_Level        );

                                                  f_PWM_Enable = false;
                                                  f_PWM_Level  =  0.25;
         f_FM_Internal  = false;
         f_AM_Internal  = false;
         f_PWM_Internal = false;

                                                  break;
                                        }
                                        case 0x0604 :
                                        {
                                                  pi->Read (f_Channel         );
                                                  pi->Read (f_Ringmod_Main       );
                                                  pi->Read (f_Ringmod_Twin       );
                                                  pi->Read (f_FM_Enable       );
                                                  pi->Read (f_FM_Level        );
                                                  pi->Read (f_AM_Enable       );
                                                  pi->Read (f_AM_Level        );
                                                  pi->Read (f_PWM_Enable      );
                                                  pi->Read (f_PWM_Level       );

         f_FM_Internal  = false;
         f_AM_Internal  = false;
         f_PWM_Internal = false;

                                                  break;
                                        }
                                        case 0x0605 :
                                        {
                                                  pi->Read (f_Channel         );
                                                  pi->Read (f_Ringmod_Main       );
                                                  pi->Read (f_Ringmod_Twin       );
                                                  pi->Read (f_FM_Enable       );
                                                  pi->Read (f_FM_Internal     );
                                                  pi->Read (f_FM_Level        );
                                                  pi->Read (f_AM_Enable       );
```

```
                                                  pi->Read (f_AM_Internal     );
                                                  pi->Read (f_AM_Level        );
                                                  pi->Read (f_PWM_Enable      );
                                                  pi->Read (f_PWM_Internal    );
                                                  pi->Read (f_PWM_Level       );

                                                  break;
                                        }
                                        case 0x0701 :
                                        {
                                                  pi->Read (f_Listen_Enabled  );
                                                  pi->Read (f_Listen_Channel    );

                                                  break;
                                        }
                                        case 0x0801 :
                                        {
          c_V1_Program l_Temp;

                                                  pi->Read (f_Num_Programs);

                                                  for (int i = 0; i < f_Num_Programs; i++)
                                                  {
                                                            pi->Read (&l_Temp, sizeof (c_V1_Program));

            l_Temp.Copy_To_Current (f_Programs [i]);
                                                  }

                                                  break;
                                        }
                                        case 0x0802 :
                                        {
          c_V2_Program l_Temp;

                                                  pi->Read (f_Num_Programs);

                                                  for (int i = 0; i < f_Num_Programs; i++)
                                                  {
                                                            pi->Read (&l_Temp, sizeof (c_V2_Program));

            l_Temp.Copy_To_Current (f_Programs [i]);
                                                  }

                                                  break;
                                        }
                                        case 0x0803 :
                                        {
                                                  pi->Read (f_Num_Programs);

                                                  for (int i = 0 ; i < f_Num_Programs; i++)
                                                  {
                                                            pi->Read (&f_Programs [i], sizeof (c_Program));
                                                  }

                                                  break;
                                        }
                                        case 0x0901 :
                                        {
                                                  for (int i = 0; i < MAX_PROGRAM_IDS; i++)
                                                  {
                                                            pi->Read (f_Program_IDs [i]);
                                                  }

                                                  break;
                                        }
                                        case 0x0A01 :
                                        {
          Read_Controls (pi);

          break;
        }
                                        case 0x0B01 :
                                        {
                                                  byte  l_Waveform;   pi ->Read (l_Waveform );  f_Mod_Waveform ->Set_Waveform
(l_Waveform);
                                                  bool  l_Optimise;   pi ->Read (l_Optimise );  f _Mod_Waveform ->Set_Optimise_Enable
(l_Optimise);
                                                  int   l_Quality;    pi ->Read (l_Quality  );  f_Mod_Waveform ->Set_Optimise_Quality (l_Quality
);

          break;
        }

                                        case 0x0C01 :
                                        {
                                                  byte  l_Depth;      pi ->Read (l_Depth    );  f_M od_Fractal->Set_Depth     (l_Depth);
                                                  float l_Effect_Hi;  pi ->Read (l_Effect_Hi);  f_Mod_Fractal ->Set_Effect_Hi (l_Effect_Hi);
                                                  float l_Effect_Lo;  pi ->Read (l_Effect_Lo);  f_Mod_Fractal ->Set_Effect_Lo (l_Effect_Lo);
                                                  bool  l_Optimise;   pi ->Read (l_Optimise );  f_Mod_Fractal ->Set_Optimise_Enable  (l_Optimise);
                                                  int   l_Quality;    pi ->Read (l_Quality  );  f_Mod_Fractal ->Set_Optimise_Quality (l_Quality );

                                                  break;
                                        }
                                        case 0x0D01 :
                                        {
                                                  pi->Read (f_Mod_Frq_Type);
                                                  pi->Read (f_Mod_Frq     );

                                                  break;
                                        }
                                        case 0x0E01 :
                                        {
                                                  float l_Attack;     pi ->Read (l_Attack   );  f_Mod_Level ->Set_Attack      (l_Attack);
                                                  float l_Decay;      pi ->Read (l_Decay    );  f_Mod_Level ->Set_Decay       (l_Decay);
                                                  float l_Sustain;    pi ->Read (l_Sustain  );  f_Mod_Level ->Set_Sustain     (l_Sustain);
```

```
                                                        float l_Release;    pi ->Read (l_Release  );  f_Mod_Level ->Set_Release     (l_Release);
                                                        float l_Start;      pi ->Read (l_Start    );  f_Mod_Level ->Set_Start       (l_St art);
                                                        float l_Peak;       pi ->Read (l_Peak     );  f_Mod_Level ->Set_Peak        (l_Peak);
                                                        float l_End;        pi ->Read (l_End      );  f_Mod_Level ->Set_End         (l_End);

                                                        break;
                                              }
                                    }

                                    pi->Read (l_Version);
                          }
                }

                Set_Master_Inf o (pMasterInfo);

                if (f_Channel != -1)
                {
                          AB_ConnectOutput (f_Channel, MacInfo.ShortName, cb, this);
                }

                if (f_Listen_Enabled)
                {
                          NR_Start_Listening (f_Listen_Channel, this);  //  Note  - ignoring possibility of error
                }
      }

      void mi::Save(CMachineDataO utput * const po)
      {
                //  Each functional block is identified by a code which indicates the type of data and
                //  the version number, thus allowing extensions to be added with the minimum of hassle.
                //
                //  A size value for each block may be a good idea, t o allow unknown blocks to be skipped.
                //  This will ensure forward compatability.

                if (po != NULL)
                {
                          po->Write ((word) 0x0102);              //  Waveform version 2
                          po->Write ((byte) f_Waveform ->Get_Waveform         ());
          po->Write (       f_Waveform ->Get_Optimise_Enable  ());
          po->Write (       f_Waveform ->Get_Optimise_Quality ());

                          po->Write ((word) 0x0202);              //  Fractal version 2
                          po->Write ((byte) f_Fractal ->Get_Depth     ());
                          po->Write (       f_Fractal ->Get_Effect_Hi ());
                          po->Write (       f_Fr actal->Get_Effect_Lo ());
          po->Write (       f_Fractal ->Get_Optimise_Enable  ());
          po->Write (       f_Fractal ->Get_Optimise_Quality ());

                          po->Write ((word) 0x0301);              //  Envelope version 1
                          po->Write (       f_Level ->Get_Attack  ());
                          po->Write (       f_Level ->Get_Decay   ());
                          po->Write (       f_Level ->Get_Sustain ());
                          po->Write (       f_Level ->Get_Release ());

                          po->Write ((word) 0x0401);              //  Pitch bend version 1
                          po->Write (       f_Pitch ->Get_Attack  ());

                          po->Write ((word) 0x0502) ;             //  Twin note version 1
                          po->Write (       f_Enable_Twin     );
                          po->Write (       f_Fractal_Before  );
                          po->Write (       f_Detune_Semitones);
                          po->Write (       f_Twin_Amp        );

                          po->Write ((word) 0x0605);              //  AuxBus Effects
                          po->Write (       f_Channel         );
                          po->Write (       f_Ringmod_Main    );
                          po->Write (       f_Ringmod_Twin    );
                          po->Write (       f_FM_Enable       );
                          po->Write (       f_FM_Internal     );
                          po->Write (       f_FM_Level        );
                          po->Write (       f_AM_ Enable       );
                          po->Write (       f_AM_Internal     );
                          po->Write (       f_AM_Level        );
                          po->Write (       f_PWM_Enable      );
                          po->Write (       f_PWM_Internal    );
                          po->Write (       f_PWM_Level       );

                          po->Write ((word) 0x0701);              //  Broadcast Rx
                          po->Write (       f_Listen_Enabled  );
                          po->Write (       f_Listen_Channel  );

                          po->Write ((word) 0x0803);              //  Program Bank
                          po->Write (       f_Num_Programs    );

                          for (int i = 0; i < f_Num_Programs; i++)
                          {
                                    po->Write (&f_Program s [i], sizeof (c_Program));
                          }

                          po->Write ((word) 0x0901);              //  Program ID Mappings

                          for (i = 0; i < MAX_PROGRAM_IDS; i++)
                          {
                                    po->Write (f_Program_IDs [i]);
                          }

                          po->Write ((word) 0x0A01);              //  Control Mappings
          Write_Controls (po);

                          po->Write ((word) 0x0B01);              //  Modulator Waveform version 1
                          po->Write ((byte) f_Mod_Waveform ->Get_Waveform         ());
          po->Write (       f_Mod_Waveform ->Get_Optimise_Enable  ());
          po->Write (       f_Mod_Waveform ->Get_Optimise_Quality ());
```

```
                    po->Write ((word) 0x0C01);              //  Modulator Fractal version 1
                    po->Write ((byte) f_Mod_Fractal->Get_Depth     ());
                    po->Write (       f_Mod_Fractal->Get_Effect_Hi ());
                    po->Write (       f_Mod_Fractal->Get_Effect_Lo ());
    po->Write (       f_Mod_Fractal->Get_Optimise_Enable  ());
    po->Write (       f_Mod_Fractal->Get_Optimise_Quality ());

                    po->Write ((word) 0x0D01);              //  Modulator Frq Settings
                    po->Write (       f_Mod_Frq_Type);
                    po->Write (       f_Mod_Frq     );

                    po->Write ((word) 0x0E01);              //  Modulator Envelope version 1
                    po->Write (       f_Mod_Level->Get_Attack  ());
                    po->Write (       f_Mod_Level->Get_Decay   ());
                    po->Write (       f_Mod_Level->Get_Sustain ());
                    po->Write (       f_Mod_Level->Get_Release ());
                    po->Write (       f_Mod_Level->Get_Start   ());
                    po->Write (       f_Mod_Level->Get_Peak    ());
                    po->Write (       f_Mod_Level->Get_End     ());

                    po->Write ((word) 0x0000);              //  Terminator
          }
}

void mi::Tick()
{
          Handle_Tick ();
}

void mi::Tick2 (void)
{
          for (int c = 0; c < numTracks; c++)
          {
                    Tracks[c]->Tick(tval[c]);
          }
}

void mi::Transpose_Notes  (int p_Semitones)
{
  int l_Note;

  for (int i = 0; i < MAX_TRACKS; i++)
  {
    if (   (tval [i].noteinit != NOTE_NO )
        && (tval [i].noteinit != NOTE_OFF))
    {
      l_Note  = ((tval [i].noteinit >> 4) * 12) + (tval [i].noteinit & 0x0F);
      l_Note += p_Semitones;

      if ((l_Note >= NOTE_MIN) && (l_Note <= NOTE_MAX))
      {
        tval [i].noteinit = ((l_Note / 12) << 4) + (l_Note % 12);
      }
      else
      {
        tval [i].noteinit = NOTE_OFF;
      }
    }

    if (   (tval [i].note != NOTE_NO )
        && (tval [i].note != NOTE_OFF))
    {
      l_Note  = ((tval [i].note >> 4) * 12) + (tval [i].note & 0x0F);
      l_Note += p_Semitones;

      if ((l_Note >= NOTE_MIN) && (l_Note <= NOTE_MAX))
      {
        tval [i].note = ((l_Note / 12) << 4) + (l_Note % 12);
      }
      else
      {
        tval [i].noteinit = NOTE_OFF;
        tval [i].note     = NOTE_NO;
      }
    }
  }
}

void mi::All_Notes_Off    (void)
{
  for (int i = 0; i < MAX_TRACKS; i++)
  {
    tval [i].noteinit = NOTE_OFF;
  }
}

void mi::All_Notes_Ignore (void)
{
  for (int i = 0; i < MAX_TRACKS; i++)
  {
    if (   (tval [i].noteinit != NOTE_NO )
        && (tval [i].noteinit  != NOTE_OFF))
    {
      tval [i].noteinit = NOTE_NO;
    }

    tval [i].note = NOTE_NO;
  }
}

void mi::Set_Control (int p_ID, int p_Value)
{
  switch (p_ID)
  {
    case 0 :  //  Volume
    {
      int k = p_Value >> 8;

      if (k > 0xFE)  {  k = 0x FE;  }

      for (int i = 0; i < MAX_TRACKS; i++)
      {
        tval [i].volume = k;
      }

      break;
    }
    case 1 :  //  Attack
    {
      float l_Temp = 10000.0 * (((float) p_Value) / 65534.0);
      f_Level->Set_Attack (l_Temp);

      break;
    }
    case 2 :  //  Decay
    {
      float l_Temp = 10000.0 * (((float) p_Value) / 65534.0);
      f_Level->Set_Decay (l_Temp);

      break;
    }
    case 3 :  //  Sustain
    {
      float l_Temp = (((float) p_Value) / 65534.0);
      f_Level->Set_Sustain (l_Temp);

      break;
    }
    case 4 :  //  Release
    {
      float l_Temp = 10000.0 * (((float) p_Value) / 65534.0);
      f_Level->Set_Release (l_Temp);

      break;
    }
    case 5 :  //  Bend Rate
    {
      float l_Temp = 10000.0 *  (((float) p_Value) / 65534.0);
      f_Pitch->Set_Attack (l_Temp);

      break;
    }
    case 6 :  //  Fractal Effect Low
    {
      float l_Effect = 9.0 * (((float) p_Value) / 65534.0);
      f_Fractal->Set_Effect_Lo (l_Effect);

      break;
    }
    case 7 :  //  Fractal Effect High
    {
      float l_Effect = 9.0 * (((float) p_Value) / 65534.0);
      f_Fractal->Set_Effect_Hi (l_Effect);

      break;
    }
    case 8 :  //  Modulator Fractal Effect Low
    {
      float l_Effect = 9.0 * (((float)  p_Value) / 65534.0);
      f_Fractal->Set_Effect_Lo (l_Effect);

      break;
    }
    case 9 :  //  Modulator Fractal Effect High
    {
      float l_Effect = 9.0 * (((float) p_Value) / 65534.0);
      f_Fractal->Set_Effect_Hi (l_Effect);

      break;
    }
  }
}

void mi::Tick2 (int p_Channel, c_Broadcast_Params *p_Command)
{
          switch (p_Command->p_Command)
          {
          //  Commands 0x01 to ? will be standardised (may have 0x40 -0x7F as macros of some form?)
          case 0x01 :  //  Program Select
                    {
                              if (   (p_Command->p_Byte_Param1                 >= 0               )
                                        && (p_Command->p_Byte_Param1                 <  MAX_PROGRAM_IDS)
                                        && (f_Program_IDs [p_Command->p_Byte_Param1] != -1             ))
                              {
                                        f_Programs [f_Program_IDs [p_Command ->p_Byte_Param1]].Set_As_Current (this);
                              }

                              break;
                    }

          //  Setting a convention where commands 0x80 upwards are usable for specific purposes
          //  by each machine, and need not be user configurable, will allow easy setup of basic
          //  commands. They will, however, ne ed to be documented for each machine  - and there
          //  will be a need to select a channel to listen to.
          case 0x80 :
                    {
                              //  Set waveform from p_Byte_Param1
```

```
                if (   (p_Command ->p_Byte_Param1 >= 1                   )
                        && (p_Command ->p_Byte_Param1 <=  NR_LIB_HIGH_WAVEFORM))
                {
                        f_Waveform->Set_Waveform (p_Command ->p_Byte_Param1);
                }

                break;
        }
case 0x81 :
        {
                //  Set fractal

                if (   (p_Command ->p_Byte_Param1 >=  0)
                        && (p_Command ->p_Byte_Param1 <= 10))
                {
                        f_Fractal->Set_Depth (p_Command ->p_Byte_Param1);
                }

                if (p_Command ->p_Word_Param1 != 0xFFFF)
                {
                        float l_Effect = 9.0 * (((float) p_Command ->p_Word_Param1) / 65534.0);
                        f_Fractal->Set_Effect_Lo (l_Effect);
                }

                if (p_Command ->p_Word_Param2 != 0xFFFF)
                {
                        float l_Effect = 9.0 * (((float) p_Command ->p_Word_Param2) / 65534.0);
                        f_Fractal->Set_Effect_Hi (l_Effect);
                }

                break;
        }
case 0x82 :
        {
                //  Set ADSR

                if (p_Command ->p_Word_Param1 != 0xFFFF)
                {
                        float l_Temp = 10000.0 * (((float) p_ Command->p_Word_Param1) / 65534.0);
                        f_Level->Set_Attack (l_Temp);
                }

                if (p_Command ->p_Word_Param2 != 0xFFFF)
                {
                        float l_Temp = 10000.0 * (((float) p_Command ->p_Word_Param2) / 65534.0);
                        f_Level->Set_Decay (l_Temp);
                }

                if (p_Command ->p_Word_Param3 != 0xFFFF)
                {
                        float l_Temp = (((float) p_Command ->p_Word_Param3) / 65534.0);
                        f_Level->Set_Sustain (l_Temp);
                }

                if (p_Command ->p_Word_Param4 != 0xFFFF)
                {
                        float l_Temp = 10000.0 * (((float) p_Command ->p_Word_Param4) / 6 5534.0);
                        f_Level->Set_Release (l_Temp);
                }

                break;
        }
case 0x83 :
        {
                //  Set Pitch Bend Rate

                if (p_Command ->p_Word_Param1 != 0xFFFF)
                {
                        float l_Temp = 10000.0 * (((float) p_Command ->p_Word_Param1) / 65534.0);
                        f_Pitch->Set_Attack (l_Temp);
                }

                break;
        }
case 0x84 :
        {
                //  Set Twin Note

                if (p_Command ->p_Byte_Param1 != 0xFF)
                {
                        f_Enable_Twin = (p_Command ->p_Byte_Param1 != 0);
                }

                if (p_Command ->p_Byte_Param2 != 0xFF)
                {
                        f_Fractal_Before = (p_Command ->p_Byte_Param2 != 0);
                }

                if (p_Command ->p_Word_Param2 != 0xFFFF)
                {
                        f_Twin_Amp = (((float) p_Command ->p_Word_Param1) / 65534.0);
                }

                break;
        }
case 0x85 :
        {
                //  Set Ring Modulations

                if (p_Command ->p_Byte_Param1 != 0xFF)
                {
                        f_Ringmod_Main = (p_Command ->p_Byte_Param1 != 0);
                }
```

```
                    if (p_Command->p_Byte_Param2 != 0xFF)
                    {
                              f_Ringmod_Twin = (p_Command->p_Byte_Param2 != 0);
                    }

                    break;
          }
case 0x86 :
          {
                    //  Set Frequency Modulation

                    if (p_Command->p_Byte_Param1 != 0xFF)
                    {
                              f_FM_Enable = (p_Command->p_Byte_Param1 != 0);
                    }

                    if (p_Command->p_Byte_Param2 != 0xFF)
                    {
                              f_FM_Internal = (p_Command->p_Byte_Param2 != 0);
                    }

                    if (p_Command->p_Word_Param1 != 0xFFFF)
                    {
                              f_FM_Level = 10.0 * (((float) p_Command->p_Word_Param1) / 65534.0);
                    }

                    break;
          }
case 0x87 :
          {
                    //  Set Amplitude Modulation

                    if (p_Command->p_Byte_Param1 != 0xFF)
                    {
                              f_AM_Enable = (p_Command->p_Byte_Param1 != 0);
                    }

                    if (p_Command->p_Byte_Param2 != 0xFF)
                    {
                              f_AM_Internal = (p_Command->p_Byte_Param2 != 0);
                    }

                    if (p_Command->p_Word_Param1 != 0xFFFF)
                    {
                              f_AM_Level = (((float) p_Command->p_Word_Param1) / 65534.0);
                    }

                    break;
          }
case 0x88 :
          {
                    //  Set Pulse Width Modulation

                    if (p_Command->p_Byte_Param1 != 0xFF)
                    {
                              f_PWM_Enable = (p_Command->p_Byte_Param1 != 0);
                    }

                    if (p_Command->p_Byte_Param2 != 0xFF)
                    {
                              f_PWM_Internal = (p_Command->p_Byte_Param2 != 0);
                    }

                    if (p_Command->p_Word_Param1 != 0xFFFF)
                    {
                              f_PWM_Level = (((float) p_Command->p_Word_Param1) / 65534.0);
                    }

                    break;
          }
case 0x89 :
          {
                    //  Set modulator waveform from p_Byte_Param1

                    if (   (p_Command->p_Byte_Param1 >= 1                    )
                              && (p_Command->p_Byte_Param1 <= NR_LIB_HIGH_WAVEFORM))
                    {
                              f_Mod_Waveform->Set_Waveform (p_Command->p_Byte_Param1);
                    }

                    break;
          }
case 0x8A :
          {
                    //  Set modulator fractal

                    if (   (p_Command->p_Byte_Param1 >=  0)
                              && (p_Command->p_Byte_Param1 <= 10))
                    {
                              f_Mod_Fractal->Set_Depth (p_Command->p_Byte_Param1);
                    }

                    if (p_Command->p_Word_Param1 != 0xFFFF)
                    {
                              float l_Effect = 9.0 * (((float) p_Command->p_Word_Param1) / 65534.0);
                              f_Mod_Fractal->Set_Effect_Lo (l_Effect);
                    }

                    if (p_Command->p_Word_Param2 != 0xFFFF)
                    {
                              float l_Effect = 9.0 * (((float) p_Command->p_Word_Param2) / 65534.0);
                              f_Mod_Fractal->Set_Effect_Hi (l_Effect);
                    }
```

```
                              break;
                    }
          }
}

void CTrack::Reset()
{
          //  Should probably reset the tracks envelope here
}

void CTrack::Tick(tvals  const &tv)
{
          if (tv.volume != paraVolume.NoValue)
          {
                    f_Level->Set_Volume (((float) tv.volume) / ((float) 0x80));
          }

          if (tv.noteinit == NOTE_OFF)
          {
                    f_Level->Stop_Note     ();
                    f_Mod_Level->Stop_Note ();
          }
          else if (tv.noteinit != NOTE_NO)
          {
                    f_Level->Start_Note     ();
                    f_Mod_Level->Start_Note ();
          }

          f_Pitch->Tick (tv.note,     tv.noteinit);
}

void CTrack::Stop()
{
          f_Level->Stop     ();
          f_Mod_Level->Stop ();
          f_Pitch->Stop     ();
}

//  Apply_Std_Mod handles amplitude and frequency modulat ion, depending on the
//  contents of the buffer it is applied to.

void Apply_Std_Mod (float *p_Dest, float *p_Src, int p_Num_Samples, float p_Level)
{
  if (p_Level != 0.0)
  {
    if (p_Level == 1.0)
    {
      while (p_Num_Samples-- > 0)
      {
        *(p_Dest++) *= (*(p_Src++) + 1.0);
      }
    }
    else
    {
      while (p_Num_Samples-- > 0)
      {
        *(p_Dest++) *= ((*(p_Src++) * p_Level) + 1.0);
      }
    }
  }
}

//  PWM needs a different algorithm

void Apply_PWM (float *p_Dest, flo at *p_Src, int p_Num_Samples, float p_Level)
{
  if (p_Level != 0.0)
  {
    if (p_Level == 1.0)
    {
      while (p_Num_Samples-- > 0)
                              {
                                        if (*p_Dest < PI)
                                        {
                                                  *p_Dest *= (*p_Src + 1.0) * 0.5;
                                        }
                                        else
                                        {
                                                  *p_Dest = ((*p_Dest  - 1.0) * (1.0 - (*p_Src)) * 0.5) + 1.0;
                                        }

                                        p_Dest++; p_Src++;
                              }
    }
    else
    {
      while (p_Num_Samples-- > 0)
                              {
                                        if (*p_Dest < PI)
                                        {
                                                  *p_Dest *= ((*p_Src * p_Level) + 1.0) * 0.5;
                                        }
                                        else
                                        {
                                                  *p_Dest = ((*p_Dest  - 1.0) * (1.0 - (*p_Src *  -p_Level)) * 0.5) + 1.0;
                                        }

                                        p_Dest++; p_Src++;
                              }
    }
  }
}

//
```

```
//  The following Generate function is starting to accumulate special cases
//
//  For readability (and possibly a small effeciency gain) it may be bette  r
//  to have several generate functions.
//

bool CTrack::Generate(float *psamples, int numsamples)
{
          //  Derive envelope

          if (f_Level->Is_Static ())
          {
                    if (f_Level->Current_Level () == 0.0)
                    {
                              return false;
                    }
          }

  //  Evaluate and save repeat  conditions

  bool l_Mod_Level_Zero = (f_Mod_Level ->Is_Static () && (f_Mod_Level->Current_Level () == 0.0));

  bool l_FM_Internal  = (pmi ->f_FM_Enable  && pmi->f_FM_Internal  && (!l_Mod_Level_Zero));
  bool l_AM_Internal  = (pmi ->f_AM_Enable  && pmi->f_AM_Internal  && (!l_Mod_Level_Zero));
  bool l_PWM_Internal = (pmi ->f_PWM_Enable && pmi->f_PWM_Internal && (!l_Mod_Level_Zero));

  bool l_FM_Needed    = (pmi ->f_FM_Enable  && ((pmi->f_FM_Internal  && (!l_Mod_Level_Zero)) || (pmi ->f_Channel != -1)));
  bool l_AM_Needed    = (pmi ->f_AM_Enable  && ((pmi->f_AM_Internal  && (!l_Mod_Level_Zero)) || (pmi ->f_Channel != -1)));
  bool l_PWM_Needed   = (pmi ->f_PWM_Enable && ((pmi->f_PWM_Internal && (!l_Mod_Level_Zero)) || (pmi ->f_Channel != -1)));

  bool l_Twin_Needed        = pmi->f_Enable_Twin;
  bool l_Twin_Mix_First     = l_Twin_Needed && (!pmi ->f_Fractal_Before);
  bool l_Twin_Fractal_First = l_Twin_Needed && ( pmi ->f_Fractal_Before);

  bool l_Ringmod_Main_Needed = pmi ->f_Ringmod_Main && (pmi->f_Channel != -1);
  bool l_Ringmod_Twin_Needed = pmi ->f_Ringmod_Twin && (pmi->f_Channel != -1) && l_Twin_Needed;

          //  Both pitch and amplitude envelopes might benefit from more frequent static
          //  checks, allowing an envelope to become static in the middle of a block. Howe  ver,
          //  tuning of the minimum amplitude might be better.

          f_Pitch->Overwrite (psamples, numsamples);

  //  Derive modulator  - this is quick and dirty at present  - much more optimisation needed

  if (l_FM_Internal || l_AM_Internal || l_PWM_Internal )
  {
    float l_Amp = 1.0;

    switch (pmi->f_Mod_Frq_Type)
    {
      case 0 :  l_Amp = pow (2.0,  pmi ->f_Mod_Frq       );  break;
      case 1 :  l_Amp = pow (2.0,  -pmi->f_Mod_Frq       );  break;
      case 2 :  l_Amp = pow (2.0,  pmi ->f_Mod_Frq / 12.0);  break;
      case 3 :  l_Amp = pow (2.0,  -pmi->f_Mod_Frq / 12.0);  break;
      case 4 :  l_Amp = pmi ->f_Mod_Frq;                     break;
      case 5 :  if (pmi ->f_Mod_Frq > 0.0)  {  l_Amp = 1.0 / pmi ->f_Mod_Frq;  }  break;
    }

    if (l_Mod_ Level_Zero)
    {
      DSP_Zero (f_Mod_Buffer, numsamples);
    }
    else
    {
      if (f_Mod_Level ->Is_Static ())  //  Optimise fractal and envelope for constant envelope level
      {
        f_Mod_Frq_To_Phase ->Process   (f_Mod_Buffer, psamples, num samples, l_Amp);
        f_Mod_Waveform ->Process       (f_Mod_Buffer, numsamples);
        f_Mod_Fractal ->Process_Static (f_Mod_Buffer, f_Mod_Level ->Current_Level (), numsamples);
        f_Mod_Level ->Multiply         (f_Buffer2, numsamples);
      }
      else
      {
        f_Mod_Level ->Overwrite (f_Buffer2, numsamples);

        f_Mod_Frq_To_Phase ->Process (f_Mod_Buffer, psamples, numsamples, l_Amp);
        f_Mod_Waveform ->Process     (f_Mod_Buffer, numsamples);
        f_Mod_Fractal ->Process      (f_M od_Buffer, f_Buffer2, numsamples);
        NR_DSP_Multiply             (f_Mod_Buffer, f_Buffer2, 32768.0, numsamples);
      }
    }
  }

          if (l_FM_Needed)
          {
    float *b;

    if (pmi->f_FM_Internal)
    {
      b = f_Mod_Buffer;
    }
    else
    {
      b = pmi->f_Aux_Buffer;
    }

    Apply_Std_Mod (psamples, b, numsamples, pmi ->f_FM_Level / 32768.0);
          }

          if (l_Twin_Needed)
          {
                    f_Frq_To_Phase2 ->Process (f_Buffer2, psamples, numsamples, pow(2.0, pmi ->f_Detune_Semitones / 12.0));
```

```
                if (l_PWM_Needed)
                {
  float *b;

  if (pmi->f_PWM_Internal)
  {
    b = f_Mod_Buffer;
  }
  else
  {
    b = pmi->f_Aux_Buffer;
  }

  Apply_PWM (f_Buffer2, b, numsamples, pmi->f_PWM_Level / 32768.0);
                }

                f_Waveform->Process      (f_Buffer2, numsamples);

                if (l_Ringmod_Twin_Needed)
                {
                          NR_DSP_Multiply (f_Buffer2, pmi->f_Aux_Buffer, 1.0 / 32768.0, numsamples);
                }
      }

      f_Frq_To_Phase->Process (psamples, numsamples);

      if (l_PWM_Needed)
      {
                float *b;

if (pmi->f_PWM_Internal)
{
  b = f_Mod_Buffer;
}
else
{
  b = pmi->f_Aux_Buffer;
}

Apply_PWM (psamples, b, numsamples, pmi->f_PWM_Level / 32768.0);
      }

      f_Waveform->Process     (psamples, numsamples);

      if (l_Ringmod_Main_Needed)
      {
                NR_DSP_Multiply (psamples, pmi->f_Aux_Buffer, 1.0 / 32768.0, numsamples);
      }

      if (l_Twin_Mix_First)
      {
                DSP_Add (psamples, f_Buffer2, numsamples, pmi->f_Twin_Amp);
                DSP_Amp (psamples, numsamples, 0.5);       //  Scaling necessary for fractal to work
      }

      //  Apply envelope and Fractal, and normalise amplitude

      if (f_Level->Is_Static ())
      {
                float l_Level = f_Level->Current_Level ();

                f_Fractal->Process_Static (psamples,  l_Level, numsamples);

                if (l_Twin_Fractal_First)
                {
                          f_Fractal->Process_Static (f_Buffer2, l_Level, numsamples);

                          DSP_Add (psamples, f_Buffer2, numsamples, pmi->f_Twin_Amp);
                          DSP_Amp (psamples, numsamples, l_Level * 16384.0);
                }
                else
                {
                          DSP_Amp (psamples, numsamples, l_Level * 32768.0);
                }
      }
      else
      {
                f_Level->Overwrite (f_Buffer, numsamples);

                if (f_Fractal->Is_Static ())
                {
                          //  If the fractal is static the modulator is irrelevant, so it may as well be zero
                          f_Fractal->Process_Static (psamples,  0.0, numsamples);

                          if (l_Twin_Fractal_First)
                          {
                                    f_Fractal->Process_Static (f_Buffer2, 0.0, numsamples);
                          }
                }
                else
                {
                          f_Fractal->Process (psamples,  f_Buffer, numsamples);

                          if (l_Twin_Fractal_First)
                          {
                                    f_Fractal->Process (f_Buffer2, f_Buffer, numsamples);
                          }
                }

                if (l_Twin_Fractal_First)
                {
                          DSP_Add (psamples, f_Buffer2, numsamples, pmi->f_Twin_Amp);

                          NR_DSP_Multiply (psamples, f_Buffer, (float) 16384.0, numsamples);
```

```
                    }
                    else
                    {
                              NR_DSP_Multiply (psamples, f_Buffer, (float) 32768.0, numsamples);
                    }
          }

          if (l_AM_Needed)
          {
    float *b;

    if (pmi->f_AM_Internal)
    {
      b = f_Mod_Buffer;
    }
    else
    {
      b = pmi->f_Aux_Buffer;
    }

    Apply_Std_Mod (psamples, b, numsamples, pmi ->f_AM_Level / 32768.0);
          }

          return true;
}

HINSTANCE dllInstance;

mi   *g_mi;

HWND g_Propsheet_HWnd [6];

BOOL WINAPI DllMain( HANDLE hModule, DWORD fdwreason, LPVOID lpReserved)
{
    switch(fdwreason)
          {
    case DLL_PROCESS_ATTACH:
          {
                    // The DLL is being mapped into process's address space
                    //  Do any required initialization on a  per application basis, return FALSE if failed
                    dllInstance=(HINSTANCE) hModule;

                    g_mi = NULL;

                    for (int i = 0; i < 6; i++)
                    {
                              g_Propsheet_HWnd [i] = NULL;
                    }

                    break;
          }
    case DLL_THREAD_ATTACH:
                    // A thread is created. Do any required init ialization on a per thread basis
                    break;
    case DLL_THREAD_DETACH:
                    // Thread exits with  cleanup
                    break;
    case DLL_PROCESS_DETACH:
              // The DLL unmapped from process's address space. Do necessary cleanup
                    break;
    }

          return TRUE;
}

#define M_CHECK_RADIO(p)  CheckRadioButton (hDlg, IDC_SINE, IDC_DBL_TRIANGLE2, p)

#define M_FROM_SLIDER(p) pow(10.0, (((float) (p)) / 3000.0) + (2.0 / 3.0))
#define M_TO_SLIDER(p)   ((log10((float) (p))  - (2.0 / 3.0)) * 3000.0)

//
//  Control handling macro s, to make it easier to move stuff from one page to
//  another or to have duplicate pages for similar types of data...
//

//  Logarithmic scale sliders range 1.0..10000.0 : Edit control notify

#define M_LOG_SLIDER_EDIT_NOTIFY(p_Slider_ID,p_Edit_ID,p_Des t)                                    \
                              case p_Edit_ID :                                                        \
                              {                                                                       \
                                        char  l_Buf [80];                                                     \
                                                                                                    \
                                        GetDlgItemText (hDlg, p_Edit_ID, l_Buf, 32);                          \
                                                                                                    \
                                        float l_Temp = atof (l_Buf) * 1000.0;                                 \
                                                                                                    \
                                        if ((l_Temp >= 1.0) && (l_Temp <= 10000.0))                           \
                                        {                                                                     \
                                                  p_Dest;                                                             \
                                                  SendDlgItemMessage (hDlg, p_Slider_ID, TBM_SETPOS, TRUE, M_TO_SLIDER (l_Temp));        \
                                        }                                                                     \
                                                                                                    \
                                        return 1;                                                             \
                              }

//  Slider notify

#define M_LOG_S LIDER_SLIDE_NOTIFY(p_Slider_ID,p_Edit_ID,p_Dest)                                    \
                    if ((HWND) lParam == GetDlgItem (hDlg, p_Slider_ID))                                \
                    {                                                                                   \
                              float l_Temp = M_FROM_SLIDER (SendDlgItemMessage (hDlg, p_Slider_ID, TBM_GETPOS, 0, 0));      \
                                                                                                     \
                              p_Dest;                                                                          \
                                                                                                     \
                              char  l_Buf [80];                                                       \
                                                                                         \
                              sprintf (l_Buf, "%f", l_Temp / 1000.0);  SetDlgItemText (hDlg, p_Edit_ID, l_Buf);              \
                    }


//  Linear scale sliders range 0.0..1.0  : Edit control notify

#define M_LIN_SLIDER_EDIT_NOTIFY(p_Slider_ID,p_Edit_ID,p_Dest)                                \
                              case p_Edit_ID :                                                    \
                              {                                                                   \
                                        char  l_Buf [80];                                                 \
                                                                                         \
                                        GetDlgItemText (hDlg, p_Edit_ID, l_Buf, 32);                              \
                                                                                         \
                                        float l_Temp = atof (l_Buf);                                      \
                                                                                         \
                                        if ((l_Temp >= 0.0) && (l_Temp <= 1.0))                           \
                                        {                                                                 \
                                                  p_Dest;                                                         \
                                                  SendDlgItemMessage (hDlg, p_Slider_ID, TBM_SETPOS, TRUE, (int) (l_Temp * 10000.0))  ;     \
                                        }                                                                 \
                                                                                         \
                                        return 1;                                                         \
                              }

//  Slider notify

#define M_LIN_SLIDER_SLIDE_NOTIFY(p_Slider_ID,p_Edit_ID,p_Dest)                                \
                    if ((HWND) lParam == GetDlgItem (hDlg, p_Slider_ID))                              \
                    {                                                                                 \
                              float l_Temp = ((float) SendDlgItemMessage (hDlg, p_Slider_ID, TBM_GETPOS, 0, 0)) / 10000.0;   \
                                                                                         \
                              p_Dest;                                                                 \
                                                                                         \
                              char  l_Buf [80];                                                       \
                                                                                         \
                              sprintf (l_Buf, "%f", l_Temp);  SetDlgItemText (hDlg, p_ Edit_ID, l_Buf);                     \
                    }



void Update_Waveform_Grp (HWND hDlg, c_Waveform *p_Waveform)
{
          switch (p_Waveform->Get_Waveform ())
          {
          case  1 :  M_CHECK_RADIO (IDC_SINE         );  break;
          case  2 :  M_CHECK_RADIO (IDC_TRIANGLE     );  br eak;
          case  3 :  M_CHECK_RADIO (IDC_DBL_TRIANGLE );  break;
          case  4 :  M_CHECK_RADIO (IDC_HEX          );  break;
          case  5 :  M_CHECK_RADIO (IDC_SQUARE       );  break;
          case  6 :  M_CHECK_RADIO (IDC_RAMP         );  break;
          case  7 :  M_CHECK_RADIO (IDC_ ALT_RAMP     );  break;
          case  8 :  M_CHECK_RADIO (IDC_BUMPS        );  break;
          case  9 :  M_CHECK_RADIO (IDC_DBL_BUMPS    );  break;
          case 10 :  M_CHECK_RADIO (IDC_ALT_BUMPS    );  break;
          case 11 :  M_CHECK_RADIO (IDC_ALT_BUMPS2   );  break;
          case 12 :   M_CHECK_RADIO (IDC_SQUARE_BUMPS );  break;
          case 13 :  M_CHECK_RADIO (IDC_DBL_BUMPS2   );  break;
          case 14 :  M_CHECK_RADIO (IDC_DBL_TRIANGLE2);  break;
          }
}

void Update_Waveform_Dlg (mi *pmi, HWND hDlg)
{
  Update_Waveform_Grp (hDlg, pmi->f_Waveform);

          CheckDlgButton (hDlg, IDC_CHECK_ENABLE_TWIN,    pmi ->f_Enable_Twin   );
          CheckDlgButton (hDlg, IDC_CHECK_FRACTAL_BEFORE, pmi ->f_Fractal_Before);

          SendDlgItemMessage (hDlg, IDC_SLIDER_DEPTH,     TBM_SETRANGE, TRUE, MAKELONG(0,   10));
          SendDlgItemMessage ( hDlg, IDC_SLIDER_EFFECT_HI, TBM_SETRANGE, TRUE, MAKELONG(0, 9000));
          SendDlgItemMessage (hDlg, IDC_SLIDER_EFFECT_LO, TBM_SETRANGE, TRUE, MAKELONG(0, 9000));
          SendDlgItemMessage (hDlg, IDC_SLIDER_DETUNE,    TBM_SETRANGE, TRUE, MAKELONG(  -10000, 10000));
          SendDlgItemMessage (hDlg, IDC_SLIDER_TWIN_AMP,  TBM_SETRANGE, TRUE, MAKELONG(0,      10000));

          {
                    int   l_Depth     = pmi->f_Fractal->Get_Depth ();
                    float l_Effect_Hi = pmi->f_Fractal->Get_Effect_Hi ();
                    float l_Effect_Lo = pmi->f_Fractal->Get_Effect_Lo ( );

                    SendDlgItemMessage (hDlg, IDC_SLIDER_DEPTH,     TBM_SETPOS, TRUE, l_Depth             );
                    SendDlgItemMessage (hDlg, IDC_SLIDER_EFFECT_HI, TBM_SETPOS, TRUE, l_Effect_Hi * 1000.0);
                    SendDlgItemMessage (hDlg, IDC_SLIDER_EFFECT_LO, TBM_SETPOS, TRUE, l_  Effect_Lo * 1000.0);
                    SendDlgItemMessage (hDlg, IDC_SLIDER_DETUNE,    TBM_SETPOS, TRUE, pmi  ->f_Detune_Semitones * 10000.0 / 12.0);
                    SendDlgItemMessage (hDlg, IDC_SLIDER_TWIN_AMP,  TBM_SETPOS, TRUE, pmi  ->f_Twin_Amp * 10000.0);

                    char  l_Buf [80];

                    sprintf (l_Buf, "%d", l_Depth    );  SetDlgItemText (hDlg, IDC_EDIT_DEPTH,     l_Buf);
                    sprintf (l_Buf, "%f", l_Effect_Hi);  SetDlgItemText (hDlg, IDC_EDIT_EFFECT_HI, l_Buf);
                    sprintf (l_Buf, "%f", l_Effect_Lo);  SetDlgItemText (hDlg, IDC_EDIT_EFFECT_LO, l_B  uf);
                    sprintf (l_Buf, "%f", pmi->f_Detune_Semitones);  SetDlgItemText (hDlg, IDC_EDIT_DETUNE, l_Buf);
                    sprintf (l_Buf, "%f", pmi->f_Twin_Amp);  SetDlgItemText (hDlg, IDC_EDIT_TWIN_AMP, l_Buf);
          }
}

void Update_Mod_Waveform_Dlg (mi *pmi, HWND hDlg)
{
  Update_Waveform_Grp (hDlg, pmi->f_Mod_Waveform);

          SendDlgItemMessage (hDlg, IDC_SLIDER_DEPTH,     TBM_SETRANGE, TRUE, MAKELONG(0,    10));
          SendDlgItemMessage (hDlg, IDC_SLIDER_EFFECT_HI, TBM_SETRANGE, TRUE, MAKELONG(0,  9000));
          SendDlgItemMessage (hDlg,  IDC_SLIDER_EFFECT_LO, TBM_SETRANGE, TRUE, MAKELONG(0,  9000));
          SendDlgItemMessage (hDlg, IDC_SLIDER_MODFRQ,    TBM_SETRANGE, TRUE, MAKELONG(0, 24000));

          {
                    int   l_Depth     = pmi ->f_Mod_Fractal ->Get_Depth ();
                    float l_Effect_Hi = pmi ->f_Mod_Fractal ->Get_Effect_Hi ();
                    float l_Effect_Lo = pmi ->f_Mod_Fractal ->Get_Effect_Lo ();

                    SendDlgItemMessage (hDlg, IDC_SLIDER_DEPTH,     TBM_SETPOS, TRUE, l_Depth                );
                    SendDlgItemMessage (hDlg, IDC_SLIDER_EFFECT_HI, TBM_SETPOS, TRUE, l_Effect_Hi    *   1000.0);
                    SendDlgItemMessage (hDlg, IDC_SLIDER_EFFECT_LO, TBM_SETPOS, TRUE, l_Effect_Lo    * 1000.0);
                    SendDlgItemMessage (hDlg, IDC_SLIDER_MODFRQ,    TBM_SETPOS, TRUE, pmi  ->f_Mod_Frq * 1000.0);

                    char  l_Buf [80];

                    sprintf (l_Buf, "%d", l_Depth        );  SetDlgItemText (hDlg, IDC_EDIT_DEPTH,     l_Buf);
                    sprintf (l_Buf, "%f", l_Effect_Hi   );  SetDlgItemText (hDlg, IDC_EDIT_EFFECT_HI, l_Buf);
                    sprintf (l_Buf, "%f", l_Effect_Lo   );  SetDlgItemText (hDlg, IDC_EDIT_EFFECT_LO, l_Buf);
                    sprintf (l_Buf , "%f", pmi ->f_Mod_Frq);  SetDlgItemText (hDlg, IDC_EDIT_MODFRQ,    l_Buf);
          }

  switch (pmi ->f_Mod_Frq_Type)
  {
    case 0 : CheckRadioButton (hDlg, IDC_RADIO_RAISE_OCTAVES, IDC_RADIO_DIVIDE_FRQ, IDC_RADIO_RAISE_OCTAVES  );  break;
    case 1 : CheckRad ioButton (hDlg, IDC_RADIO_RAISE_OCTAVES, IDC_RADIO_DIVIDE_FRQ, IDC_RADIO_LOWER_OCTAVES  );  break;
    case 2 : CheckRadioButton (hDlg, IDC_RADIO_RAISE_OCTAVES, IDC_RADIO_DIVIDE_FRQ, IDC_RADIO_RAISE_SEMITONES);  break;
    case 3 : CheckRadioButton (hDlg,  IDC_RADIO_RAISE_OCTAVES, IDC_RADIO_DIVIDE_FRQ, IDC_RADIO_LOWER_SEMITONES);  break;
    case 4 : CheckRadioButton (hDlg, IDC_RADIO_RAISE_OCTAVES, IDC_RADIO_DIVIDE_FRQ, IDC_RADIO_MULTIPLY_FRQ   );  break;
    case 5 : CheckRadioButton (hDlg, IDC_RADIO_RAISE_ OCTAVES, IDC_RADIO_DIVIDE_FRQ, IDC_RADIO_DIVIDE_FRQ     );  break;
  }
}

void Update_Aux_Bus_Dlg (mi *pmi, HWND hDlg)
{
          CheckDlgButton (hDlg, IDC_CHECK_RINGMOD_MAIN, pmi ->f_Ringmod_Main);
          CheckDlgButton (hDlg, IDC_CHECK_RINGMOD_TWIN, pmi ->f_Ringmod_Twin );
          CheckDlgButton (hDlg, IDC_CHECK_FM_ENABLE,    pmi ->f_FM_Enable );
          CheckDlgButton (hDlg, IDC_CHECK_AM_ENABLE,    pmi ->f_AM_Enable );
          CheckDlgButton (hDlg, IDC_CHECK_PWM_ENABLE,   pmi ->f_PWM_Enable);
          CheckDlgButton (hDlg, IDC_CHECK_FM_INTERNAL,  pmi ->f_FM_Internal);
          CheckDlgButton (hDlg, IDC_CHECK_AM_INTERNAL,  pmi ->f_AM_Internal );
          CheckDlgButton (hDlg, IDC_CHECK_PWM_INTERNAL, pmi ->f_PWM_Internal);
  CheckDlgButton (hDlg, IDC_CHECK_WAV_OPT,      pmi ->f_Waveform ->Get_Optimise_Enable ());
  CheckDlgBu tton (hDlg, IDC_CHECK_FRACT_OPT,    pmi ->f_Fractal ->Get_Optimise_Enable ());

          SendDlgItemMessage (hDlg, IDC_SLIDER_FM_LEVEL,  TBM_SETRANGE, TRUE, MAKELONG(0,      10000));
          SendDlgItemMessage (hDlg, IDC_SLIDER_AM_LEVEL,  TBM_SETRANGE, TRUE, MAKELONG(0,        10000));
          SendDlgItemMessage (hDlg, IDC_SLIDER_PWM_LEVEL, TBM_SETRANGE, TRUE, MAKELONG(0,      10000));

          {
                    SendDlgItemMessage (hDlg, IDC_SLIDER_FM_LEVEL,  TBM_SETPOS, TRUE, M_TO_SLIDER(pmi  ->f_FM_Level  *  1000.0));
                    SendDlgItemMessage (hDlg, IDC_SLI DER_AM_LEVEL,  TBM_SETPOS, TRUE, M_TO_SLIDER(pmi ->f_AM_Level  * 10000.0));
                    SendDlgItemMessage (hDlg, IDC_SLIDER_PWM_LEVEL, TBM_SETPOS, TRUE, M_TO_SLIDER(pmi  ->f_PWM_Level * 10000.0));

                    char  l_Buf [80];

                    sprintf (l_Buf, "%f", pmi ->f_FM_Level);  SetDlg ItemText (hDlg, IDC_EDIT_FM_LEVEL,  l_Buf);
                    sprintf (l_Buf, "%f", pmi ->f_AM_Level);  SetDlgItemText (hDlg, IDC_EDIT_AM_LEVEL,  l_Buf);
                    sprintf (l_Buf, "%f", pmi ->f_PWM_Level); SetDlgItemText (hDlg, IDC_EDIT_PWM_LEVEL, l_Buf);
    sprintf (l_Buf, "%d",  pmi->f_Waveform ->Get_Optimise_Quality ());  SetDlgItemText (hDlg, IDC_EDIT_WAV_OPT,  l_Buf);
    sprintf (l_Buf, "%d", pmi ->f_Fractal ->Get_Optimise_Quality ());  SetDlgItemText (hDlg, IDC_EDIT_FRACT_OPT,  l_Buf);
          }
}

void Update_Envel_Dlg (mi *pmi, HWND  hDlg)
{
          SendDlgItemMessage (hDlg, IDC_SLIDER_ATTACK,  TBM_SETRANGE, TRUE, MAKELONG(0, 10000));
          SendDlgItemMessage (hDlg, IDC_SLIDER_DECAY,   TBM_SETRANGE, TRUE, MAKELONG(0, 10000));
          SendDlgItemMessage (hDlg, IDC_SLIDER_SUSTAIN, TBM_SETRANGE, TRUE, MAKEL  ONG(0, 10000));
          SendDlgItemMessage (hDlg, IDC_SLIDER_RELEASE, TBM_SETRANGE, TRUE, MAKELONG(0, 10000));
          SendDlgItemMessage (hDlg, IDC_SLIDER_BEND,    TBM_SETRANGE, TRUE, MAKELONG(0, 10000));

          {
                    float l_Attack  = pmi ->f_Level->Get_Attack  ();
                    float l_Decay   = pmi ->f_Level->Get_Decay   ();
                    float l_Sustain = pmi ->f_Level->Get_Sustain ();
                    float l_Release = pmi ->f_Level->Get_Release ();
                    float l_Bend    = pmi ->f_Pitch->Get_Attack  ();

                    SendDlgItemMessage (hDlg, IDC_SLIDER_ATTACK,  TBM_SETPOS, TRUE,  M_TO_SLIDER (l_Attack) );
                    SendDlgItemMessage (hDlg, IDC_SLIDER_DECAY,   TBM_SETPOS, TRUE, M_TO_SLIDER (l_Decay ) );
                    SendDlgItemMessage (hDlg, IDC_SLIDER_SUSTAIN, TBM_SETPOS, TRUE, l_Sustain *  10000.0   );
                    SendDlgItemMessage (hDlg, IDC_SLIDER_RELEAS E, TBM_SETPOS, TRUE, M_TO_SLIDER (l_Release));
                    SendDlgItemMessage (hDlg, IDC_SLIDER_BEND,    TBM_SETPOS, TRUE, M_TO_SLIDER (l_Bend   ));

                    char  l_Buf [80];

                    sprintf (l_Buf, "%f", l_Attack  / 1000.0);  SetDlgItemText (hDlg, IDC_EDIT_ATTACK,  l_Buf);
                    sprintf (l_Buf, "%f", l_Decay   / 1000.0);  SetDlgItemText (hDlg, IDC_EDIT_DECAY,   l_Buf);
                    sprintf (l_Buf, "%f", l_Sustain         );  SetDlgItemText (hDlg, IDC_EDIT_SUSTAIN, l_Buf);
                    sprintf (l_Buf, "%f", l_Release / 1000.0);  SetDlgItemText (hDlg, I  DC_EDIT_RELEASE, l_Buf);
                    sprintf (l_Buf, "%f", l_Bend    / 1000.0);  SetDlgItemText (hDlg, IDC_EDIT_BEND,    l_Buf);
          }
}

void Update_Mod_Envel_Dlg (mi *pmi, HWND hDlg)
{
          SendDlgItemMessage (hDlg, IDC_SLIDER_ATTACK,  TBM_SETRANGE, TRUE, MAKELONG(0, 100  00));
          SendDlgItemMessage (hDlg, IDC_SLIDER_DECAY,   TBM_SETRANGE, TRUE, MAKELONG(0, 10000));
          SendDlgItemMessage (hDlg, IDC_SLIDER_SUSTAIN, TBM_SETRANGE, TRUE, MAKELONG(0, 10000));
          SendDlgItemMessage (hDlg, IDC_SLIDER_RELEASE, TBM_SETRANGE, TRUE, MAKELON  G(0, 10000));
          SendDlgItemMessage (hDlg, IDC_SLIDER_START,   TBM_SETRANGE, TRUE, MAKELONG(0, 10000));
          SendDlgItemMessage (hDlg, IDC_SLIDER_PEAK,    TBM_SETRANGE, TRUE, MAKELONG(0, 10000));
          SendDlgItemMessage (hDlg, IDC_SLIDER_END,     TBM_SETRANGE, TRUE,  MAKELONG(0, 10000));
```

```
        {
                float l_Attack  = pmi->f_Mod_Level->Get_Attack  ();
                float l_Decay   = pmi->f_Mod_Level->Get_Decay   ();
                float l_Sustain = pmi->f_Mod_Level->Get_Sustain ();
                float l_Release = pmi->f_Mod_Level->Get_Release ();
                float l_Start   = pmi->f_Mod_Level->Get_Start   ();
                float l_Peak    = pmi->f_Mod_Level->Get_Peak    ();
                float l_End     = pmi->f_Mod_Level->Get_End     ();

                SendDlgItemMessage (hDlg, IDC_SLIDER_ATTACK,  TBM_SETPOS, TRUE, M_TO_SLIDER (l_Attack) );
                SendDlgItemMessage (hDlg, IDC_SLIDER_DECAY,   TBM_SETPOS, TRUE, M_TO_SLIDER (l_Decay ) );
                SendDlgItemMessage (hDlg, IDC_SLIDER_SUSTAIN, TBM_SETPOS, TRUE, l_Sustain *  10000.0   );
                SendDlgItemMessage (hDlg, IDC_SLIDER_RELEASE, TBM_SETPOS, TRUE, M_TO_SLIDER (l_Release));
                SendDlgItemMessage (hDlg, IDC_SLIDER_START,   TBM_SETPOS, TRUE, l_Start   *  10000.0   );
                SendDlgItemMessage (hDlg, IDC_SLIDER_PEAK,    TBM_SETPOS, TRUE, l_Peak    *  10000.0   );
                SendDlgItemMessage (hDlg, IDC_SLIDER_END,     TBM_SETPOS, TRUE, l_End     *  10000.0   );

                char  l_Buf [80];

                sprintf (l_Buf, "%f", l_Attack  / 1000.0);  SetDlgItemText (hDlg, IDC_EDIT_ATTACK,  l_Buf);
                sprintf (l_Buf, "%f", l_Decay   / 1000.0);  SetDlgItemText (hDlg, IDC_EDIT_DECAY,   l_Buf);
                sprintf (l_Buf, "%f", l_Sustain         );  SetDlgItemText (hDlg, IDC_EDIT_SUSTAIN, l_Buf);
                sprintf (l_Buf, "%f", l_Release / 1000.0);  SetDlgItemText (hDlg, IDC_EDIT_RELEASE, l_Buf);
                sprintf (l_Buf, "%f", l_Start           );  SetDlgItemText (hDlg, IDC_EDIT_START,   l_Buf);
                sprintf (l_Buf, "%f", l_Peak            );  SetDlgItemText (hDlg, IDC_EDIT_PEAK,    l_Buf);
                sprintf (l_Buf, "%f", l_End             );  SetDlgItemText (hDlg, IDC_EDIT_END,     l_Buf);
        }
}

void Update_All_Dialogs (void)  //  Exclude the Rx Broadcast page
{
        if (g_Propsheet_HWnd [0] != NULL)
        {
                Update_Waveform_Dlg (g_mi, g_Propsheet_HWnd [0]);
        }

        if (g_Propsheet_HWnd [1] != NULL)
        {
                Update_Aux_Bus_Dlg (g_mi, g_Propsheet_HWnd [1]);
        }

        if (g_Propsheet_HWnd [2] != NULL)
        {
                Update_Envel_Dlg (g_mi, g_Propsheet_HWnd [2]);
        }

        if (g_Propsheet_HWnd [3] != NULL)
        {
                Update_Mod_Waveform_Dlg (g_mi, g_Propsheet_HWnd [3]);
        }

        if (g_Propsheet_HWnd [4] != NULL)
        {
                Update_Mod_Envel_Dlg (g_mi, g_Propsheet_HWnd [4]);
        }
}

BOOL APIENTRY MainDialog (HWND hDlg, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
        switch (uMsg)
        {
        case WM_INITDIALOG :
                g_Propsheet_HWnd [0] = hDlg;

                Update_Waveform_Dlg (g_mi, hDlg);

                return 1;

        case WM_COMMAND :
                switch (LOWORD (wParam))
                {
                        //  The CheckRadioButton calls here shouldn't be necessary, but maybe I've hit some
                        //  obscure limit.
                        //
                        //  It might be better to use a combobox or listbox  - it will certainly be easier to
                        //  extend in the future, and could save space on screen when multiple oscilators   are
                        //  added (modulators etc).

                    case IDC_SINE          :  M_CHECK_RADIO (IDC_SINE         );  g_mi  ->f_Waveform->Set_Waveform ( 1);  return 1;
                        case IDC_TRIANGLE      :  M_CHECK_RADIO (IDC_TRIANGLE     );  g_mi  ->f_Waveform->Set_Waveform ( 2);  return 1;
                        case IDC_DBL_TRIANGLE  :  M_CHECK_RADIO (IDC_DBL_TRIANGLE );  g_mi  ->f_Waveform->Set_Waveform ( 3);  return 1;
                        case IDC_HEX           :  M_CHECK_RADIO (IDC_HEX          );  g_mi  ->f_Waveform->Set_Waveform ( 4);  return 1;
                        case IDC_SQUARE        :  M_CHECK_RADIO (IDC_SQUARE       );  g_mi  ->f_Waveform->Set_Waveform ( 5);  return 1;
                        case IDC_RAMP          :  M_CHECK_RADIO (IDC_RAMP         );  g_mi  ->f_Waveform->Set_Waveform ( 6);  return 1;
                        case IDC_ALT_RAMP      :  M_CHECK_RADIO (IDC_ALT_RAMP     );  g_mi  ->f_Waveform->Set_Waveform ( 7);  return 1;
                        case IDC_BUMPS         :  M_CHECK_RADIO (IDC_BUMPS        );  g_mi  ->f_Waveform->Set_Waveform ( 8);  return 1;
                        case IDC_DBL_BUMPS     :  M_CHECK_RADIO (IDC_DBL_BUMPS    );  g_mi  ->f_Waveform->Set_Waveform ( 9);  return 1;
                        case IDC_ALT_BUMPS     :  M_CHECK_RADIO (IDC_ALT_BUMPS    );  g_mi  ->f_Waveform->Set_Waveform (10);  return 1;
                        case IDC_ALT_BUMPS2    :  M_CHECK_RADIO (IDC_ALT_BUMPS2   );  g_mi  ->f_Waveform->Set_Waveform (11);  return 1;
                        case IDC_SQUARE_BUMPS  :  M_CHECK_RADIO (IDC_SQUARE_BUMPS );  g_mi  ->f_Waveform->Set_Waveform (12);  return 1;
                        case IDC_DBL_BUMPS2    :  M_CHECK_RADIO (IDC_DBL_BUMPS2   );  g_mi  ->f_Waveform->Set_Waveform (13);  return 1;
                        case IDC_DBL_TRIANGLE2 :  M_CHECK_RADIO (IDC_DBL_TRIANGLE2);  g_mi  ->f_Waveform->Set_Waveform (14);  return 1;

                        case IDC_EDIT_DEPTH :
                        {
                                char  l_Buf [80];

                                GetDlgItemText (hDlg, IDC_EDIT_DEPTH, l_Buf, 32);

                                int l_Depth = atoi (l_Buf);

                                if ((l_Depth >= 0) && (l_Depth <= 10))
                                {
```

```
                                                g_mi->f_Fractal->Set_Depth (l_Depth);
                                                SendDlgItemMessage (hDlg, IDC_SLIDER_DEPTH, TBM_SETPOS, TRUE, l_Depth);
                                        }

                                        return 1;
                                }
                                case IDC_EDIT_EFFECT_HI :
                                {
                                        char  l_Buf [80];

                                        GetDlgItemText (hDlg, IDC_ED IT_EFFECT_HI, l_Buf, 32);

                                        float l_Effect_Hi = atof (l_Buf);

                                        if ((l_Effect_Hi >= 0.0) && (l_Effect_Hi <= 9.0))
                                        {
                                                g_mi->f_Fractal->Set_Effect_Hi (l_Effect_Hi);
                                                SendDlgItemMessage (hDlg, IDC_SLIDER_EFFECT_HI, TBM_SETPOS, TRUE, (int) (l  _Effect_Hi *
1000.0));
                                        }

                                        return 1;
                                }
                                case IDC_EDIT_EFFECT_LO :
                                {
                                        char  l_Buf [80];

                                        GetDlgItemText (hDlg, IDC_EDIT_EFFECT_LO, l_Buf, 32);

                                        float l_Effect_Lo = atof (l_Buf);

                                        if ((l_Effect_Lo >= 0.0) && (l_Effect_Lo <= 9.0) )
                                        {
                                                g_mi->f_Fractal->Set_Effect_Lo (l_Effect_Lo);
                                                SendDlgItemMessage (hDlg, IDC_SLIDER_EFFECT_LO, TBM_SETPOS, TRUE, (int) (l_Effect_Lo *
1000.0));
                                        }

                                        return 1;
                                }
                                case IDC_CHECK_ENABLE_TWIN :
                                {
                                        g_mi->f_Enable_Twin = (SendD lgItemMessage (hDlg, IDC_CHECK_ENABLE_TWIN, BM_GETCHECK, 0, 0) ==
BST_CHECKED);

                                        return 1;
                                }
                                case IDC_CHECK_FRACTAL_BEFORE :
                                {
                                        g_mi->f_Fractal_Before = (SendDlgItemMessage (hDlg, IDC_CHECK_FRACTAL_BEFORE, BM_GETCHECK, 0, 0) ==
BST_CHECKED) ;

                                        return 1;
                                }
                                case IDC_EDIT_DETUNE :
                                {
                                        char  l_Buf [80];

                                        GetDlgItemText (hDlg, IDC_EDIT_DETUNE, l_Buf, 32);

                                        float l_Detune = atof (l_Buf);

                                        if ((l_Detune >=  -12.0) && (l_Detune <= 12.0))
                                        {
                                                g_mi->f_Detune_Semitones =  l_Detune;
                                                SendDlgItemMessage (hDlg, IDC_SLIDER_DETUNE, TBM_SETPOS, TRUE, (int) (l_Detune * 10000.0 /
12.0));
                                        }

                                        return 1;
                                }
                                case IDC_EDIT_TWIN_AMP :
                                {
                                        char  l_Buf [80];

                                        GetDlgItemText (hDlg, IDC_EDIT_TWIN_AMP, l_Buf, 32);

                                        float l_Twin_Amp = atof (l_Buf);

                                        if ((l_Twin_Amp >= 0.0) && (l_Twin_Amp <= 1.0))
                                        {
                                                g_mi->f_Twin_Amp = l_Twin_Amp;
                                                SendDlgItemMessage (hDlg, IDC_SLIDER_TWIN_AMP, TBM_SETPOS, TRUE, (int) (l_Twin_Amp *
10000.0));
                                        }

                                        return 1;
                                }
                        }

                        break;

                case WM_HSCROLL :
                        if ((HWND) lParam == GetDlgItem (hDlg, IDC_SLIDER_DEPTH))
                        {
                                int   l_Depth     = SendDlgItemMessage (hDlg, IDC_SLIDER_DEPTH, TBM_GETPOS, 0, 0);

                                g_mi->f_Fractal->Set_Depth (l_Depth);

                                char  l_Buf [80];

                                sprintf (l_Buf, "%d", l_Depth    );  SetDlgItemText (hDlg, IDC_EDIT_DEPTH,     l_Buf);
                        }
                        else if ((HWND) lParam == GetDlgItem (hDlg, IDC_SLIDER_EFFECT_HI))
```

```
                    {
                              float l_Effect_Hi = ((float) SendDlgItemMessage (hDlg, IDC_SLIDER_EFFECT_HI, TBM_GETPOS, 0,   0)) / 1000.0;

                              g_mi->f_Fractal->Set_Effect_Hi (l_Effect_Hi);

                              char  l_Buf [80];

                              sprintf (l_Buf, "%f", l_Effect_Hi);  SetDlgItemText (hDlg, IDC_EDIT_EFFECT_HI, l_Buf);
                    }
                    else if ((HWND) lParam == GetDlgItem (hDlg, IDC_SLIDER_EFFECT_LO))
                    {
                              float l_Effect_Lo = ((float) SendDlgItemMessage (hDlg, IDC_SLIDER_EFFECT_LO, TBM_GETPOS, 0, 0)) / 1000.0;

                              g_mi->f_Fractal->Set_Effect_Lo (((float) SendDlgItemMessage (hDlg, IDC_SLIDER_EFFECT_LO, TBM_GETPOS, 0, 0)) /
1000.0);

                              char  l_Buf [80];

                              sprintf (l_Buf, "%f", l_Effect_Lo);  SetDlgItemText (hDlg, IDC_EDIT_EFFECT_LO, l_Buf);
                    }
                    else if ((HWND) lParam == GetDlgItem (hDlg, IDC_SLIDER_DETUNE))
                    {
                              g_mi->f_Detune_Semitones = ((float) SendDlgItemMessage (hDlg, IDC_SLIDER_DETUNE, TBM_GETPOS  , 0, 0)) * 12.0 /
10000.0;

                              char  l_Buf [80];

                              sprintf (l_Buf, "%f", g_mi ->f_Detune_Semitones);  SetDlgItemText (hDlg, IDC_EDIT_DETUNE, l_Buf);
                    }
                    else if ((HWND) lParam == GetDlgItem (hDlg, IDC_SLIDER_TWIN_AMP))
                    {
                              g_mi->f_Twin_Amp = ((float)  SendDlgItemMessage (hDlg, IDC_SLIDER_TWIN_AMP, TBM_GETPOS, 0, 0)) / 10000.0;

                              char  l_Buf [80];

                              sprintf (l_Buf, "%f", g_mi ->f_Twin_Amp);  SetDlgItemText (hDlg, IDC_EDIT_TWIN_AMP, l_Buf);
                    }

                    break;

          }

          return 0;
}

BOOL APIENTRY ModWaveDialog  (HWND hDlg, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
          switch (uMsg)
          {
          case WM_INITDIALOG :
                    g_Propsheet_HWnd [3] = hDlg;

                    Update_Mod_Waveform_Dlg (g_mi, hDlg);

                    return 1;

          case WM_COMMAND :
                    switch (LOWORD (wParam))
                    {
                              //  The CheckRadioButt on calls here shouldn't be necessary, but maybe I've hit some
                              //  obscure limit.
                              //
                              //  It might be better to use a combobox or listbox  - it will certainly be easier to
                              //  extend in the future, and could save space on screen when multiple osci  lators are
                              //  added (modulators etc).

              case IDC_SINE          :  M_CHECK_RADIO (IDC_SINE         );  g_mi  ->f_Mod_Waveform ->Set_Waveform ( 1);  return 1;
                              case IDC_TRIANGLE      :  M_CHECK_RADIO (IDC_TRIANGLE     );  g_mi  ->f_Mod_Waveform->Set_Waveform ( 2);  return 1;
                              case IDC_DBL_TRIANGLE  :  M_CHECK_RADIO (IDC_DBL_TRIANGLE );  g_mi  ->f_Mod_Waveform->Set_Waveform ( 3);  return 1;
                              case IDC_HEX           :  M_CHECK_RADIO (IDC_HEX          );  g_mi  ->f_Mod_Waveform->Set_Waveform ( 4);  return 1 ;
                              case IDC_SQUARE        :  M_CHECK_RADIO (IDC_SQUARE       );  g_mi  ->f_Mod_Waveform->Set_Waveform ( 5);  return 1;
                              case IDC_RAMP          :  M_CHECK_RADIO (IDC_RAMP         );  g_mi  ->f_Mod_Waveform->Set_Waveform ( 6);  return 1;
                              case IDC_ALT_RAMP       :  M_CHECK_RADIO (IDC_ALT_RAMP     );  g_mi ->f_Mod_Waveform->Set_Waveform ( 7);  return 1;
                              case IDC_BUMPS         :  M_CHECK_RADIO (IDC_BUMPS        );  g_mi  ->f_Mod_Waveform->Set_Waveform ( 8);  return 1;
                              case IDC_DBL_BUMPS     :  M_CHECK_RADIO  (IDC_DBL_BUMPS    );  g_mi ->f_Mod_Waveform->Set_Waveform ( 9);  return 1;
                              case IDC_ALT_BUMPS     :  M_CHECK_RADIO (IDC_ALT_BUMPS    );  g_mi  ->f_Mod_Waveform->Set_Waveform (10);  return 1;
                              case IDC_ALT_BUMPS2    :  M_CHECK_RADIO (IDC_ALT_BUMPS2   );   g_mi->f_Mod_Waveform->Set_Waveform (11);  return 1;
                              case IDC_SQUARE_BUMPS  :  M_CHECK_RADIO (IDC_SQUARE_BUMPS );  g_mi  ->f_Mod_Waveform->Set_Waveform (12);  return 1;
                              case IDC_DBL_BUMPS2    :  M_CHECK_RADIO (IDC_DBL_BUMPS2   );  g_mi  ->f_Mod_Waveform->Set_Waveform (13);  return 1;
                              case IDC_DBL_TRIANGLE2 :  M_CHECK_RADIO (IDC_DBL_TRIANGLE2);  g_mi  ->f_Mod_Waveform->Set_Waveform (14);  return 1;

      case IDC_RADIO_RAISE_OCTAVES :
        g_mi->f_Mod_Frq_Type = 0;
        CheckRadioButton (hDlg, IDC_ RADIO_RAISE_OCTAVES, IDC_RADIO_DIVIDE_FRQ, IDC_RADIO_RAISE_OCTAVES  );
        break;
      case IDC_RADIO_LOWER_OCTAVES :
        g_mi->f_Mod_Frq_Type = 1;
        CheckRadioButton (hDlg, IDC_RADIO_RAISE_OCTAVES, IDC_RADIO_DIVIDE_FRQ, IDC_RADIO_LOWER_OCTA  VES  );
        break;
      case IDC_RADIO_RAISE_SEMITONES :
        g_mi->f_Mod_Frq_Type = 2;
        CheckRadioButton (hDlg, IDC_RADIO_RAISE_OCTAVES, IDC_RADIO_DIVIDE_FRQ, IDC_RADIO_RAISE_SEMITONES  );
        break;
      case IDC_RADIO_LOWER_SEMITONES  :
        g_mi->f_Mod_Frq_Type = 3;
        CheckRadioButton (hDlg, IDC_RADIO_RAISE_OCTAVES, IDC_RADIO_DIVIDE_FRQ, IDC_RADIO_LOWER_SEMITONES  );
        break;
      case IDC_RADIO_MULTIPLY_FRQ :
        g_mi->f_Mod_Frq_Type = 4;
        CheckRadioButton  (hDlg, IDC_RADIO_RAISE_OCTAVES, IDC_RADIO_DIVIDE_FRQ, IDC_RADIO_MULTIPLY_FRQ  );
        break;
      case IDC_RADIO_DIVIDE_FRQ :
        g_mi->f_Mod_Frq_Type = 5;
        CheckRadioButton (hDlg, IDC_RADIO_RAISE_OCTAVES, IDC_RADIO_DIVIDE_FRQ, IDC_RADIO_DIV  IDE_FRQ  );
        break;

                        case IDC_EDIT_DEPTH :
                        {
                                char  l_Buf [80];

                                GetDlgItemText (hDlg, IDC_EDIT_DEPTH, l_Buf, 32);

                                int l_Depth = atoi (l_Buf);

                                if ((l_Depth >= 0) && (l_Depth <= 10))
                                {
                                        g_mi->f_Mod_Fractal->Set_Depth (l_Depth);
                                        SendDlgItemMessage (hDlg, IDC_SLIDER_DEPTH, TBM_SETPOS, TRUE, l_Depth);
                                }

                                return 1;
                        }
                        case IDC_EDIT_EFFECT_HI :
                        {
                                char  l_Buf [80];

                                GetDlgItemText (hDlg, IDC_EDIT_EFFECT_HI, l_Buf, 32);

                                float l_Effect_Hi = atof  (l_Buf);

                                if ((l_Effect_Hi >= 0.0) && (l_Effect_Hi <= 9.0))
                                {
                                        g_mi->f_Mod_Fractal->Set_Effect_Hi (l_Effect_Hi);
                                        SendDlgItemMessage (hDlg, IDC_SLIDER_EFFECT_HI, TBM_SETPOS, TRUE, (int) (l_Effect_Hi *
1000.0));
                                }

                                return 1;
                        }
                        case IDC_EDIT_EFFECT_LO :
                        {
                                char  l_Buf [80];

                                GetDlgItemText (hDlg, IDC_EDIT_EFFECT_LO, l_Buf, 32);

                                float l_Effect_Lo = atof (l_Buf);

                                if ((l_Effect_Lo >= 0.0) && (l_Effect_Lo <= 9.0))
                                {
                                        g_mi->f_Mod_Fractal->Set_Effect_Lo (l_Effect_Lo);
                                        SendDlgItemMessage (hDlg, IDC_SLIDER_EFFECT_LO, TBM_SETPOS, TRUE, (int) (l_Effect_Lo *
1000.0));
                                }

                                return 1;
                        }
                        case IDC_EDIT_MODFRQ :
                        {
                                char  l_Buf [80];

                                GetDlgItemText (hDlg, IDC_EDIT_MODFRQ, l_Buf, 32);

                                float l_Mod_Frq = atof (l_Buf);

                                if ((l_Mod_Frq >= 0.0) && (l_Mod_Frq <= 24.0))
                                {
                                        g_mi->f_Mod_Frq = l_Mod_Frq;
                                        SendDlgItemMessage (hDlg, IDC_SLIDER_MODFRQ, TBM_SETPOS, TRUE, (int) (l_Mod_Frq * 1000.0));
                                }

                                return 1;
                        }
                }

                break;

         case WM_HSCROLL :
                if ((HWND) lParam == GetDlgItem (hDlg, IDC_SLIDER_DEPTH))
                {
                        int   l_Depth     = SendDlgItemMessage (hDlg, IDC_SLIDER_DEPTH, TBM_GETPOS, 0, 0);

                        g_mi->f_Mod_Fractal->Set_Depth (l_Depth);

                        char  l_Buf [80];

                        sprintf (l_Buf, "%d", l_Depth    );  SetDlgItemText (hDlg, IDC_EDIT_DEPTH,     l_Buf);
                }
                else if ((HWND) lParam == GetDlgItem (hDlg, IDC_SLIDER_EFFECT_HI))
                {
                        float l_Effect_Hi = ((float) SendDlgItemMessage (hDlg, IDC_SLIDER_EFFECT_HI, TBM_GETPOS, 0, 0)) / 10  00.0;

                        g_mi->f_Mod_Fractal->Set_Effect_Hi (l_Effect_Hi);

                        char  l_Buf [80];

                        sprintf (l_Buf, "%f", l_Effect_Hi);  SetDlgItemText (hDlg, IDC_EDIT_EFFECT_HI, l_Buf);
                }
                else if ((HWND) lParam == GetDlgItem (hDlg, IDC_SLIDER_EFFECT_LO))
                {
                        float l_Effect_Lo = ((float) SendDlgItemMessage (hDlg, IDC_SLIDER_EFFECT_LO, TBM_GETPOS, 0, 0)) / 1000.0;

                        g_mi->f_Mod_Fractal->Set_Effect_Lo (((float) SendDlgItemMessage (hDlg, IDC_SLIDER_EFFECT_LO, TBM_GETPOS, 0, 0)) /
1000.0);

                        char  l_Buf [80];

                        sprintf (l_Buf, "%f", l_Effect_Lo);  SetDlgItemText (hDlg, IDC_EDIT_EFFECT_LO, l_Buf);
                }
```

```
                    else if ((HWND) lParam == GetDlgItem (hDlg, IDC_SLIDER_MODFRQ))
                    {
                              g_mi->f_Mod_Frq = ((float) SendDlgItemMessage (hDlg, IDC_SLIDER_MODFRQ, TBM_GETPOS, 0, 0)) /   1000.0;

                              char  l_Buf [80];

                              sprintf (l_Buf, "%f", g_mi ->f_Mod_Frq);  SetDlgItemText (hDlg, IDC_EDIT_MODFRQ, l_Buf);
                    }

                    break;

          }

          return 0;
}

BOOL APIENTRY AuxBusDialog (HWND hDlg, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
          switch (uMsg)
          {
          case WM_INITDIALOG :
                    g_Propsheet_HWnd [1] = hDlg;

                    Update_Aux_Bus_Dlg (g_mi, hDlg);

                    return 1;

          case WM_COMMAND :
                    switch (LOWORD (wParam))
                    {
      case IDC_CHECK_WAV_OPT :
      {
                                        bool l_Enabled = (SendDlgItemMessage (hDlg, IDC_CHECK_WAV_OP  T, BM_GETCHECK, 0, 0) == BST_CHECKED);

        g_mi ->f_Waveform ->Set_Optimise_Enable (l_Enabled);

        return 1;
      }
      case IDC_EDIT_WAV_OPT :
      {
                                        char  l_Buf [80];

                                        GetDlgItemText (hDlg, IDC_EDIT_WAV_OPT, l_Buf, 32);

                                        int l_Quality = atoi (l_Buf);

        if ((l_Quality > 1) && (l_Quality <= NR_LIB_WAVEFORM_MAX_QUALITY))
        {
          g_mi ->f_Waveform ->Set_Optimise_Quality (l_Quality);
        }

        return 1;
      }
      case IDC_CHECK_FRACT_OPT :
      {
                                        bool l_Enabled = (SendDlgItemMessage (hDlg, IDC_CHECK_FRACT_OPT, BM_GETCHECK, 0, 0) == BST_CHECKED);

        g_mi ->f_Fractal ->Set_Optimise_Enable (l_Enabled);

        return 1;
      }
      case IDC_EDIT_FRACT_OPT :
      {
                                        char  l_Buf [80];

                                        GetDlgItemText (hDlg, IDC_EDIT_FRACT_OPT, l_Buf, 32);

                                        int l_Quality = atoi (l_Buf);

        if ((l_Quality > 1) && (l_Quality <= NR_LIB_FRACTAL_MAX_QUALITY))
        {
          g_mi ->f_Fractal ->Set_Optimise_Quality (l_Quality);
        }

        return 1 ;
      }
                              case IDC_CHECK_RINGMOD_MAIN :
                              {
                                        g_mi->f_Ringmod_Main = (SendDlgItemMessage (hDlg, IDC_CHECK_RINGMOD_MAIN, BM_GETCHECK, 0, 0) ==
BST_CHECKED);

                                        return 1;
                              }
                              case IDC_CHECK_RINGMOD_TWIN :
                              {
                                        g_mi->f_Ringmod_Twin = (SendDlgI temMessage (hDlg, IDC_CHECK_RINGMOD_TWIN, BM_GETCHECK, 0, 0) ==
BST_CHECKED);

                                        return 1;
                              }
                              case IDC_CHECK_FM_ENABLE :
                              {
                                        g_mi->f_FM_Enable = (SendDlgItemMessage (hDlg, IDC_CHECK_FM_ENABLE, BM_GETCHECK, 0, 0) == BST_CHECKED);

                                        return 1;
                              }
                              case IDC_CHECK_AM_ENABLE :
                              {
                                        g_mi->f_AM_Enable = (SendDlgItemMessage (hDlg, IDC_CHECK_AM_ENABLE, BM_GETCHECK, 0, 0) == BST_CHECKED);

                                        return 1;
                              }
                              case IDC_CHECK_PWM_ENABLE :
                              {
```

```
                                        g_mi->f_PWM_Enable = (SendDlgItemMessage (hDlg,  IDC_CHECK_PWM_ENABLE, BM_GETCHECK, 0, 0) ==
BST_CHECKED);

                                        return 1;
                              }
                              case IDC_CHECK_FM_INTERNAL :
                              {
                                        g_mi->f_FM_Internal = (SendDlgItemMessage (hDlg, IDC_CHECK_FM_INTERNAL, BM_GETCHECK, 0, 0) ==
BST_CHECKED);

                                        return 1;
                              }
                              case IDC_CHECK_AM_INTERNAL :
                              {
                                        g_mi->f_AM_Internal = (SendDlgItemMessage (hDlg, IDC_CHECK_AM_INTERNAL, BM_GETCHECK, 0, 0) ==
BST_CHECKED);

                                        return 1;
                              }
                              case IDC_CHECK_PWM_INTERNAL :
                              {
                                        g_mi->f_PWM_Internal = (SendDlgItemMessage (hDlg, IDC_ CHECK_PWM_INTERNAL, BM_GETCHECK, 0, 0) ==
BST_CHECKED);

                                        return 1;
                              }
                              case IDC_SET_CHANNEL :
                              {
                                        AB_ShowEditor(NULL, &g_mi ->f_Channel, MacInfo.ShortName, cb, g_mi);

                                        return 1;
                              }
                              case IDC_DISCONNECT :
                              {
                                        g_mi->f_Channel = -1;

                                        AB_Disconnect (g_mi);

                                        return 1;
                              }
                              case IDC_EDIT_FM_LEVEL :
                              {
                                        char  l_Buf [80];

                                        GetDlgItemText (hDlg, IDC_EDIT_FM_LEVEL, l_Buf, 32);

                                        float l_FM_Level = atof (l_Buf);

                                        if ((l_FM_Level >= 0.0) && (l_FM_Level <= 10.0))
                                        {
                                                  g_mi->f_FM_Level = l_FM_Level;
                                                  SendDlgItemMessage (hDlg, IDC_SLIDER_FM_LEVEL, TBM_SETPOS, TRUE, (int) M_TO_SLIDER (l_FM_Level
* 1000.0));
                                        }

                                        return 1;
                              }
                              case IDC_EDIT_AM_LEVEL :
                              {
                                        char  l_Buf [80];

                                        GetDlgItemText (hDlg,  IDC_EDIT_AM_LEVEL, l_Buf, 32);

                                        float l_AM_Level = atof (l_Buf);

                                        if ((l_AM_Level >= 0.0) && (l_AM_Level <= 1.0))
                                        {
                                                  g_mi->f_AM_Level = l_AM_Level;
                                                  SendDlgItemMessage (hDlg, IDC_SLIDER_AM_LEVEL, TBM_SETPOS, TRUE, (int) M_TO_SLIDER (l_  AM_Level
* 10000.0));
                                        }

                                        return 1;
                              }
                              case IDC_EDIT_PWM_LEVEL :
                              {
                                        char  l_Buf [80];

                                        GetDlgItemText (hDlg, IDC_EDIT_PWM_LEVEL, l_Buf, 32);

                                        float l_PWM_Level = atof (l_Buf);

                                        if ((l_PWM_Level >= 0.0) && (l_PWM_Level <= 1.0))
                                        {
                                                  g_mi->f_PWM_Level = l_PWM_Level;
                                                  SendDlgItemMessage (hDlg, IDC_SLIDER_PWM_LEVEL, TBM_SETPOS, TRUE, (int) M_TO_SLIDER
(l_PWM_Level * 10000.0));
                                        }

                                        return 1;
                              }
                    }

                    break;

          case WM_HSCROLL :
                    if ((HWND) lParam == GetDlgItem (h Dlg, IDC_SLIDER_FM_LEVEL))
                    {
                              g_mi->f_FM_Level = M_FROM_SLIDER (SendDlgItemMessage (hDlg, IDC_SLIDER_FM_LEVEL, TBM_GETPOS, 0, 0)) / 1000.0;

                              char  l_Buf [80];

                              sprintf (l_Buf, "%f", g_mi ->f_FM_Level);  SetDlgItemText (hDlg, IDC_EDIT_FM_LEVEL, l_B uf);
                    }
                    else if ((HWND) lParam == GetDlgItem (hDlg, IDC_SLIDER_AM_LEVEL))
                    {
                              g_mi->f_AM_Level = M_FROM_SLIDER (SendDlgItemMessage (hDlg, IDC_SLIDER_AM_LEVEL, TBM_GETPOS, 0, 0)) / 10000.0;

                              char  l_Buf [80];

                              sprintf (l_Buf, "%f", g_mi ->f_AM_Level);  SetDlgItemText (hDlg, IDC_EDIT_AM_LEVEL, l_Buf);
                    }
                    else if ((HWND) lParam == GetDlgItem (hDlg, IDC_SLIDER_PWM_LEVEL))
                    {
                              g_mi->f_PWM_Level = M_FROM_SLIDER (SendDlgItemMessage (hDlg, IDC_SLIDER_PWM_LEVEL, TBM_GETPOS, 0, 0)) / 10000.0;

                              char  l_Buf [80];

                              sprintf (l_Buf, "%f", g_mi ->f_PWM_Level);  SetDlgItemText (hDlg, IDC_EDIT_PWM_LEVEL, l_Buf);
                    }

                    break;
          }

          return 0;
}

LV_COLUMN g_Listview_Cols [] =
          {
                    {LVCF_TEXT | LVCF_WIDTH | LVCF_SUBITEM, LVCFMT_LEFT, 100, "ID",            2  , 0},
                    {LVCF_TEXT | LVCF_WIDTH | LVCF_SUBITEM, LVCFMT_LEFT, 100, "Program Name", 12, 1}
          };

#define NUM_CONTROL_CONFIGS 10

c_Control_Config g_Control_Configs [NUM_CONTROL_CONFIGS] =
{
  {0, "Volume"             },
  {1, "Attack"             },
  {2, "Decay"              },
  {3, "Sustain"            },
  {4, "Release"            },
  {5, "Bend Rate"          },
  {6, "Fractal Effect Low" },
  {7, "Fractal Effect High"},
  {8, "Modulator Fractal Effect Low" },
  {9, "Modulator Fractal Effect High"}
};

BOOL APIENTRY RxBroadcastDialog (HWND hDlg, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
          switch (uMsg)
          {
          case WM_INITDIALOG :

                    CheckDlgButton (hDlg, IDC_CHECK_LISTEN_ENABLE, g_mi ->f_Listen_Enabled );

                    {
                              char  l_Buf [80];

                              sprintf (l_Buf, "%d", g_mi->f_Listen_Channel);  SetDlgItemText (hDlg, IDC_EDIT_LISTEN_CHAN,  l_Buf);

                              EnableWindow (GetDlgItem (hDlg, IDC_BUTTON_PRG_CREATE), FALSE);
                              EnableWindow (GetDlgItem (hDlg, IDC_BUTTON_PRG_DELETE), FALSE);
                              EnableWindow (GetDlgItem (hDlg, IDC_BUTTON _PRG_UPDATE), FALSE);
                              EnableWindow (GetDlgItem (hDlg, IDC_BUTTON_PRG_SELECT), FALSE);
                              EnableWindow (GetDlgItem (hDlg, IDC_BUTTON_ID_CREATE ), FALSE);
                              EnableWindow (GetDlgItem (hDlg, IDC_BUTTON_ID_DELETE ), FALSE);

                              SendDlgItemMessage (hDlg, IDC _SPIN_LISTEN_CHAN, UDM_SETBUDDY, (WPARAM) (HWND) GetDlgItem (hDlg,
IDC_EDIT_LISTEN_CHAN), 0L);
                              SendDlgItemMessage (hDlg, IDC_SPIN_LISTEN_CHAN, UDM_SETRANGE, 0, MAKELONG(7, 0));

                              SendDlgItemMessage (hDlg, IDC_COMBO_PROG_NAME, CB_LIMITTEXT, 31, 0L);
                              SendDlgItemMessage (hDlg, IDC_COMBO_PRG_IDS,   CB_LIMITTEXT, 31, 0L);

                              ListView_InsertColumn (GetDlgItem (hDlg, IDC_LIST_PROG_IDS), 0, &g_Listview_Cols [0]);
                              ListView_InsertColumn (GetDlgItem (hDlg, IDC_LIST_PROG_IDS), 1, &g_Listview_Cols [1]);

                              c_Program::Fill_Program_Combo (g_mi, hDlg);
                              c_Program::Fill_ID_List       (g_mi, hDlg);
                    }

                    return 1;
          case WM_NOTIFY :
          {
                    NMHDR *l_Notify = (NMHDR *) lParam;

                    if (   (l_Notify->hwndFrom == GetDlgItem (hDlg, IDC_LIST_PROG_IDS))
                              && (l_Notify->code     == LVN_GETDISPINFO                    ))
                    {
                              LV_DISPINFO *l_Disp_Info = (LV_DISPINFO *) lParam;

                              if (l_Disp_Info->item.iSubItem == 1)
                              {
        if (g_mi->f_Program_IDs [l_Disp_Info->item.lParam] ==  -1)
        {
          l_Disp_Info->item.pszText = "";
        }
        else
        {
                                          l_Disp_Info->item.pszText = g_mi->f_Programs [g_mi->f_Program_IDs [l_Disp_Info->item.lParam]].f_Name;
        }
                              }
                    }

                    break;
          }
```

```
    case WM_COMMAND :
              switch (LOWORD (wParam))
              {
case IDC_CONFIG_CTRLS :
{
  NR_Dialog_Enable_Channel (g_mi, &g_mi ->f_Listen_Enabled, &g_mi ->f_Listen_Channel,
                           NUM_CONTROL_CONFIGS, g_Control_Configs                 );


  return 1;
}
                        case IDC_CHECK_LISTEN_ENABLE :
                        {
                                  bool l_Enabled = (SendDlgItemMessage (hDlg, IDC_CHECK_LISTEN_ENABLE, BM_GETCHECK, 0, 0) == BST_CHECKED);

                                  if (g_mi->f_Listen_Enabled)
                                  {
                                            if (!l_Enabled)
                                            {
                                                      NR_Stop_Listening  (g_mi ->f_Listen_Channel, g_mi);
                                            }
                                  }
                                  else
                                  {
                                            if (l_Enabled)
                                            {
                                                      NR_Start_Listening (g_mi ->f_Listen_Channel, g_mi);
                                            }
                                  }

                                  g_mi->f_Listen_Enabled = l_Enabled;
                                  return 1;
                        }
                        case IDC_EDIT_LISTEN_CHAN :
                        {
                                  char  l_Buf [80];

                                  GetDlgItemText ( hDlg, IDC_EDIT_LISTEN_CHAN, l_Buf, 32);

                                  int l_Channel = atoi (l_Buf);

                                  if ((l_Channel >= 0) && (l_Channel <= 7) && (l_Channel != g_mi  ->f_Listen_Channel))
                                  {
                                            if (g_mi->f_Listen_Enabled)
                                            {
                                                      NR_Stop_Listening  (g_mi ->f_Listen_Channel, g_mi);
                                                      g_mi->f_Listen_Channel = l_Channel;
                                                      NR_Start_Listening (l_Channel, g_mi);
                                            }
                                            else
                                            {
                                                      g_mi->f_Listen_Channel = l_Channel;
                                            }
                                  }

                                  return 1;
                        }
                        case IDC_COMBO_PROG_NAME :
                        {
                                  char  l_Buf [MAX_PROGRAM_NAME];
                                  int   l_Prog, l_ID;

                                  if (HIWORD(wParam) == CBN_EDITUPDATE)
                                  {
                                            GetDlgItemText (hDlg, IDC_COMBO_PROG_NAME, l_Buf, MAX_PROGRAM_NAME);

                                            l_Prog = c_Program::Find (g_mi, l_Buf);
                                  }
                                  else if (HIWORD(wParam) == CBN_SELENDOK)
                                  {
                                WORD n = (WORD) SendDlgItemMessage (hDlg, IDC_COMBO_PROG_NAME, CB_GETCURSEL, 0, 0L);
                                SendDlgItemMessage (hDlg, IDC_COMBO_PROG_NAME, CB_GETLBTEXT, n, (LONG) (LPSTR) l_Buf);

                                            l_Prog = c_Program::Find (g_mi, l_Buf);
                                  }
                                  else
                                  {
                                            return 1;
                                  }

                                  c_Program::Fill_ID_Combo (g_mi, hDlg, l_Prog);

                                  EnableWindow (GetDlgItem (hDlg, IDC_BUTTON_PRG_CREATE),
                                                      (l_Prog ==  -1) && (l_Buf[0] != 0) && (g_mi ->f_Num_Programs < MAX_PROGRAMS));

                                  EnableWindow (GetDlgItem (hDlg, IDC_BUTTON_PRG_DELETE),
                                                      (l_Prog !=  -1)                           );

                                  EnableWindow (GetDlgItem (hDlg, IDC_BUTTON_PRG_UPDATE),
                                                      (l_Prog !=  -1)                           );

                                  EnableWindow (GetDlgItem (hDlg,  IDC_BUTTON_PRG_SELECT),
                                                      (l_Prog !=  -1)                           );

                                  GetDlgItemText (hDlg, IDC_COMBO_PRG_IDS, l_Buf, MAX_PROGRAM_NAME);

                                  l_ID = atoi (l_Buf);

                                  EnableWindow (GetDlgItem (hDlg, IDC_BUTTON_ID_CREATE ),
                                                      (l_Prog != -1) && (l_Buf [0] != 0) && (g_mi ->f_Program_IDs [l_ID] ==  -1    ) );

                                  EnableWindow (GetDlgItem (hDlg, IDC_BUTTON_ID_DELETE ),
                                                      (l_Prog != -1) && (l_Buf [0] != 0) && (g_mi ->f_Program_IDs [l_ID] == l_Prog) );
```

```
                    return 1;
          }
          case IDC_COMBO_PRG_IDS :
          {
                    char  l_Buf [MAX_PROGRAM_NAME];
                    int   l_Prog, l_ID;

                    GetDlgItemText (hDlg, IDC_COMBO_PROG_NAME, l_Buf, MAX_PROGRAM_NAME);

                    l_Prog = c_Program::Find (g_mi, l_Buf);

                    if (HIWORD(wParam) == CBN_EDITUPDATE)
                    {
                              GetDlgItemText (hDlg, IDC_COMBO_PRG_IDS, l_Buf, MAX_PROGRAM_NAME);
                    }
                    else if (HIWORD(wParam) == CBN_SELENDOK)
                    {
                  WORD n = (WORD) SendDlgItemMessage (hDlg, IDC_COMBO_PRG_IDS, CB_GETCURSEL, 0, 0L);
                  SendDlgItemMessa ge (hDlg, IDC_COMBO_PRG_IDS, CB_GETLBTEXT, n, (LONG) (LPSTR) l_Buf);
                    }
                    else
                    {
                              return 1;
                    }

                    l_ID = atoi (l_Buf);

                    EnableWindow (GetDlgItem (hDlg, IDC_BUTTON_ID_CREATE ),
                                        (l_Prog !=  -1) && (l_Buf [0] != 0) && (g_mi ->f_Program_IDs [l_ID] ==  -1    ) );

                    EnableWindow (GetDlgItem (hDlg, IDC_BUTTON_ID_DELETE ),
                                        (l_Prog !=  -1) && (l_Buf [0] != 0) && (g_mi ->f_Program_IDs [l_ID] == l_Prog) );

                    return 1;
          }
          case IDC_BUTTON_PRG_CREATE :
          {
                    char  l_Buf [MAX_PROGRAM_NAME];

                    GetDlgItemText (hDlg, IDC_COMBO_PROG_NAME, l_Buf, MAX_PROGRAM_NAME);

                    int l_Prog = c_Program::Find (g_mi, l_Buf);

                    if ((l_Prog ==  -1) && (l_Buf[0] != 0) && (g_mi ->f_Num_Programs < MAX_PROGRAMS))
                    {
                              strcpy (g_mi->f_Programs [g_mi ->f_Num_Programs].f_Name, l_Buf);
                              g_mi->f_Programs [g_mi ->f_Num_Programs].Get_From_Current (g_mi);
                              g_mi->f_Num_Programs++;

                              EnableWindow (GetDlgItem (hDlg, IDC_BUTTON_PRG_CREATE), FALSE);
                              EnableWindow (GetDlgItem  (hDlg, IDC_BUTTON_PRG_DELETE), TRUE );
                              EnableWindow (GetDlgItem (hDlg, IDC_BUTTON_PRG_UPDATE), TRUE );
                              EnableWindow (GetDlgItem (hDlg, IDC_BUTTON_PRG_SELECT), TRUE );
                              EnableWindow (GetDlgItem (hDlg, IDC_BUTTON_ID_CREATE ), FALSE);
                              EnableWindow (GetDlgItem (hDlg, IDC_BUTTON_ID_DELETE ), FALSE);

                              c_Program::Fill_Program_Combo (g_mi, hDlg);
                              c_Program::Fill_ID_Combo      (g_mi, hDlg, g_mi ->f_Num_Programs  - 1);
                              c_Program::Fill_ID_List       (g_mi, hDlg);

                              SetDlgItemText (h Dlg, IDC_COMBO_PROG_NAME, l_Buf);
                    }

                    return 1;
          }
          case IDC_BUTTON_PRG_DELETE :
          {
                    char  l_Buf [MAX_PROGRAM_NAME];

                    GetDlgItemText (hDlg, IDC_COMBO_PROG_NAME, l_Buf, MAX_PROGRAM_NAME);

                    int l_Prog = c_Program::Find (g_mi, l_Buf);

                    if (l_Prog != -1)
                    {
                              g_mi->f_Num_Programs --;

                              if (l_Prog != g_mi ->f_Num_Programs)
                              {
                                        g_mi->f_Programs[l_Prog] = g_mi ->f_Programs[l_Prog];  //  Note shallow copy
                              }

                              for (int i = 0; i < MAX_PROGRAM_IDS; i++)
                              {
                                        if (g_mi->f_Program_IDs [i] == l_Prog)
                                        {
                                                  g_mi->f_Program_IDs [i] =  -1;
                                        }
                              }

                              EnableWindow (GetDlgItem (hDlg, IDC_BUTTON_PRG_CREATE), TRUE );
                              EnableWindow (GetDlgItem (hDlg, IDC_BUTTON_PRG_DELETE), FALSE);
                              EnableWindow (GetDlgItem (hDlg, IDC_BUTTON_PRG_UPDATE), FALSE);
                              EnableWindow (GetDlgItem (hDlg, IDC_BUTTON_PRG_SELECT), FALSE);
                              EnableWindow (GetDlgItem (hDlg, IDC_BUTTON_ID_CREATE ), FALSE);
                              EnableWindow (GetDlgItem (hDlg, IDC_BUTTON_ID_DELETE ), FALSE  );

                              c_Program::Fill_Program_Combo (g_mi, hDlg);
                              c_Program::Fill_ID_Combo (g_mi, hDlg, l_Prog);
                              c_Program::Fill_ID_List       (g_mi, hDlg);

                              SetDlgItemText (hDlg, IDC_COMBO_PROG_NAME, l_Buf);
                    }
```

```
                                        return 1;
                              }
                              case IDC_BUTTON_ PRG_UPDATE :
                              {
                                        char  l_Buf [MAX_PROGRAM_NAME];

                                        GetDlgItemText (hDlg, IDC_COMBO_PROG_NAME, l_Buf, MAX_PROGRAM_NAME);

                                        int l_Prog = c_Program::Find (g_mi, l_Buf);

                                        if (l_Prog != -1)
                                        {
                                                  g_mi->f_Programs [l_Prog].Get_From_Current (g _mi);
                                                  c_Program::Fill_ID_Combo (g_mi, hDlg, l_Prog);
                                                  c_Program::Fill_ID_List       (g_mi, hDlg);
                                        }

                                        return 1;
                              }
                              case IDC_BUTTON_PRG_SELECT :
                              {
                                        char  l_Buf [MAX_PROGRAM_NAME];

                                        GetDlgItemText (hDlg, IDC_COMBO_PROG_NAME, l_ Buf, MAX_PROGRAM_NAME);

                                        int l_Prog = c_Program::Find (g_mi, l_Buf);

                                        if (l_Prog != -1)
                                        {
                                                  g_mi->f_Programs [l_Prog].Set_As_Current (g_mi);
                                                  c_Program::Fill_ID_Combo (g_mi, hDlg, l_Prog);
                                                  c_Program::Fill_ID_List       (g_mi, hDlg);
                                                  Update_All_Dialogs ();
                                        }

                                        return 1;
                              }
                              case IDC_BUTTON_ID_CREATE :
                              {
                                        char  l_Buf [MAX_PROGRAM_NAME];

                                        GetDlgItemText (hDlg, IDC_COMBO_PROG_NAME, l_Buf, MAX_PROGRAM_NAME);

                                        int l_Prog = c_Program::Find (g_mi, l_Buf);

                                        GetDlgItemText (hDlg, IDC_COMBO_PRG_IDS, l_Buf, MAX_PROGRAM_NAME);

                                        int l_ID = atoi (l_Buf);

                                        if ((l_Prog != -1) && (l_Buf [0] != 0) && (g_mi ->f_Program_IDs [l_ID] ==  -1))
                                        {
                                                  g_mi->f_Program_IDs [l_ID] = l_Prog;

                                                  EnableWindow (GetDlgIte m (hDlg, IDC_BUTTON_ID_CREATE ),
                                                                      (l_Prog !=  -1) && (l_Buf [0] != 0) && (g_mi ->f_Program_IDs [l_ID] ==  -1
) );

                                                  EnableWindow (GetDlgItem (hDlg, IDC_BUTTON_ID_DELETE ),
                                                                      (l_Prog !=  -1) && (l_Buf [0] != 0) && (g_mi ->f_Program_IDs [l_ID] ==
l_Prog) );

                                                  c_Program::Fill_ID_Combo (g_mi, hDlg, l_Prog);
                                                  c_Program::Fill_ID_List       (g_mi, hDlg);
                                        }

                                        return 1;
                              }
                              case IDC_BUTTON_ID_DELETE :
                              {
                                        char  l_Buf [MAX_PROGRAM_NAME];

                                        GetDlgItemText (hDlg , IDC_COMBO_PROG_NAME, l_Buf, MAX_PROGRAM_NAME);

                                        int l_Prog = c_Program::Find (g_mi, l_Buf);

                                        GetDlgItemText (hDlg, IDC_COMBO_PRG_IDS, l_Buf, MAX_PROGRAM_NAME);

                                        int l_ID = atoi (l_Buf);

                                        if ((l_Prog != -1) && (l_Buf [0] != 0) && (g_mi ->f_Program_IDs [l_ID] == l_Prog))
                                        {
                                                  g_mi->f_Program_IDs [l_ID] =  -1;

                                                  EnableWindow (GetDlgItem (hDlg, IDC_BUTTON_ID_CREATE ),
                                                                      (l_Prog !=  -1) && (l_Buf [0] != 0) && (g_mi ->f_Program_IDs [l_ID] ==  -1
) );

                                                  EnableWindow (Get DlgItem (hDlg, IDC_BUTTON_ID_DELETE ),
                                                                      (l_Prog !=  -1) && (l_Buf [0] != 0) && (g_mi ->f_Program_IDs [l_ID] ==
l_Prog) );

                                                  c_Program::Fill_ID_Combo (g_mi, hDlg, l_Prog);
                                                  c_Program::Fill_ID_List       (g_mi, hDlg);
                                        }

                                        return 1;
                              }
                    }
          }

          return 0;
}
```

```
BOOL APIENTRY EnvelDialog (HWND hDlg, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
          switch (uMsg)
          {
          case WM_INITDIALOG :
                    g_Propsheet_HWnd [2] = hDlg;

                    Update_Envel_Dlg (g_mi, hDlg);

                    return 1;

          case WM_COMMAND :
                    switch (LOWORD (wParam))
                    {
      M_LOG_SLIDER_EDIT_NOTIFY(IDC_SLIDER_ATTACK, IDC_EDIT_ATTACK, g_mi  ->f_Level->Set_Attack  (l_Temp))
      M_LOG_SLIDER_EDIT_NOTIFY(IDC_SLIDER_DECAY,  IDC_EDIT_DECAY,  g_mi  ->f_Level->Set_Decay   (l_Temp))
      M_LIN_SLIDER_EDIT_ NOTIFY(IDC_SLIDER_SUSTAIN,IDC_EDIT_SUSTAIN,g_mi ->f_Level->Set_Sustain (l_Temp))
      M_LOG_SLIDER_EDIT_NOTIFY(IDC_SLIDER_RELEASE,IDC_EDIT_RELEASE,g_mi  ->f_Level->Set_Release (l_Temp))
      M_LOG_SLIDER_EDIT_NOTIFY(IDC_SLIDER_BEND,   IDC_EDIT_BEND,   g_mi  ->f_Pitch->Set_Attack  (l_Temp))
                    }

                    break;

          case WM_HSCROLL :
         M_LOG_SLIDER_SLIDE_NOTIFY(IDC_SLIDER_ATTACK, IDC_EDIT_ATTACK, g_mi  ->f_Level->Set_Attack  (l_Temp))
    else M_LOG_SLIDER_SLIDE_NOTIFY(IDC_SLIDER_DECAY,  IDC_EDIT_DECAY,  g_mi  ->f_Level->Set_Decay   (l_Temp))
    else M_LIN_SLIDER_SLIDE_NOTIFY(IDC_SLIDER_SUSTAIN,IDC_EDIT_SUSTAIN,g_mi  ->f_Level->Set_Sustain (l_Temp))
    else M_LOG_SLIDER_SLIDE_NOTIFY(IDC_SLIDER_RELEASE,IDC_EDIT_RELEASE,g_mi  ->f_Level->Set_Release (l_Temp))
    else M_LOG_SLIDER_SLIDE_NOTIFY(IDC_SLIDER_BEND,   IDC_EDIT_BEND,   g_mi  ->f_Pitch->Set_Attack  (l_Temp))
          }

          return 0;
}

BOOL APIENTRY ModEnvelDialog (HWND hDlg, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
          switch (uMsg)
          {
          case WM_INITDIALOG :
                    g_Propsheet_HWnd [ 4] = hDlg;

                    Update_Mod_Envel_Dlg (g_mi, hDlg);

                    return 1;

          case WM_COMMAND :
                    switch (LOWORD (wParam))
                    {
      M_LOG_SLIDER_EDIT_NOTIFY(IDC_SLIDER_ATTACK, IDC_EDIT_ATTACK, g_mi  ->f_Mod_Level->Set_Attack  (l_Temp))
      M_LOG_SLIDER_EDIT_NOTIFY(IDC_ SLIDER_DECAY,  IDC_EDIT_DECAY,  g_mi ->f_Mod_Level->Set_Decay   (l_Temp))
      M_LIN_SLIDER_EDIT_NOTIFY(IDC_SLIDER_SUSTAIN,IDC_EDIT_SUSTAIN,g_mi  ->f_Mod_Level->Set_Sustain (l_Temp))
      M_LOG_SLIDER_EDIT_NOTIFY(IDC_SLIDER_RELEASE,IDC_EDIT_RELEASE,g_mi  ->f_Mod_Level->Set_Release (l_Temp))
      M_LIN_SLIDER_EDIT_NOTIFY(IDC_SLIDER_START,  IDC_EDIT_START,  g_mi  ->f_Mod_Level->Set_Start   (l_Temp))
      M_LIN_SLIDER_EDIT_NOTIFY(IDC_SLIDER_PEAK,   IDC_EDIT_PEAK,   g_mi  ->f_Mod_Level->Set_Peak    (l_Temp))
      M_LIN_SLIDER_EDIT_NOTIFY(IDC_SLIDER_END,    IDC_EDIT_END,    g_mi  ->f_Mod_Level->Set_End     (l_Temp))
                    }

                    break;

          case WM_HSCROLL :
         M_LOG_SLIDER_SLIDE_NOTIFY(IDC_SLIDER_ATTACK, IDC_EDIT_ATTACK, g_mi  ->f_Mod_Level->Set_Attack  (l_Temp))
    else M_LOG_SLIDER_SLIDE_NOTIFY(IDC_SLIDER_DECAY,  IDC_EDIT_DECAY,  g_mi  ->f_Mod_Level->Set_Decay   (l_Temp))
    else M_LIN_SLIDER_SLIDE_NOTIFY(IDC_SLIDER_SUSTAIN,IDC_EDIT_SUSTAIN,g_mi  ->f_Mod_Level->Set_Sustain (l_Temp))
    else M_LOG_SLIDER_SLIDE_NOTIFY(IDC_SL IDER_RELEASE,IDC_EDIT_RELEASE,g_mi ->f_Mod_Level->Set_Release (l_Temp))
    else M_LIN_SLIDER_SLIDE_NOTIFY(IDC_SLIDER_START,  IDC_EDIT_START,  g_mi  ->f_Mod_Level->Set_Start   (l_Temp))
    else M_LIN_SLIDER_SLIDE_NOTIFY(IDC_SLIDER_PEAK,   IDC_EDIT_PEAK,   g_  mi->f_Mod_Level->Set_Peak    (l_Temp))
    else M_LIN_SLIDER_SLIDE_NOTIFY(IDC_SLIDER_END,    IDC_EDIT_END,    g_mi  ->f_Mod_Level->Set_End     (l_Temp))
          }

          return 0;
}

void mi::Command(int const i)
{
          switch (i)
          {
          case 0 :  //  Edit Ninereeds NRS04
                    {
                              InitCommonControls ();

                              PROPSHEETPAGE l_Pages[7];

                              l_Pages[0].dwSize      = sizeof(PROPSHEETPAGE);
                              l_Pages[0].dwFlags     = PSP_DEFAULT | PSP_USETITLE;
                              l_Pages[0].hInstance   = dllInstance;
                              l_Pages[0].pszTemplate = MAKEINTRESOURCE(IDD_NRS 04);
                              l_Pages[0].pszIcon     = NULL;
                              l_Pages[0].pszTitle    = "Carrier Waveform";
                              l_Pages[0].pfnDlgProc  = (DLGPROC) &MainDialog;
                              l_Pages[0].lParam      = 0;
                              l_Pages[0].pfnCallback = NULL;
                              l_Pages[0].pcRefParent = NULL;

                              l_Pages[1].dwSi ze      = sizeof(PROPSHEETPAGE);
                              l_Pages[1].dwFlags     = PSP_DEFAULT;
                              l_Pages[1].hInstance   = dllInstance;
                              l_Pages[1].pszTemplate = MAKEINTRESOURCE(IDD_ENVEL);
                              l_Pages[1].pszIcon     = NULL;
                              l_Pages[1].pszTitle    = NULL;
                              l_Pages[1].pfn DlgProc  = (DLGPROC) &EnvelDialog;
                              l_Pages[1].lParam      = 0;
                              l_Pages[1].pfnCallback = NULL;
                              l_Pages[1].pcRefParent = NULL;
                              l_Pages[2].dwSize      = sizeof(PROPSHEETPAGE);
                              l_Pages[2].dwFlags     = PSP_DEFAULT;
                              l_Pages[2].hInstance   = d llInstance;
                              l_Pages[2].pszTemplate = MAKEINTRESOURCE(IDD_MOD_WAVE);
                              l_Pages[2].pszIcon     = NULL;
                              l_Pages[2].pszTitle    = NULL;
                              l_Pages[2].pfnDlgProc  = (DLGPROC) &ModWaveDialog;
                              l_Pages[2].lParam      = 0;
                              l_Pages[2].pfnCallback = NULL ;
                              l_Pages[2].pcRefParent = NULL;

                              l_Pages[3].dwSize      = sizeof(PROPSHEETPAGE);
                              l_Pages[3].dwFlags     = PSP_DEFAULT;
                              l_Pages[3].hInstance   = dllInstance;
                              l_Pages[3].pszTemplate = MAKEINTRESOURCE(IDD_MOD_ENVEL);
                              l_Pages[3].pszIcon      = NULL;
                              l_Pages[3].pszTitle    = NULL;
                              l_Pages[3].pfnDlgProc  = (DLGPROC) &ModEnvelDialog;
                              l_Pages[3].lParam      = 0;
                              l_Pages[3].pfnCallback = NULL;
                              l_Pages[3].pcRefParent = NULL;

                              l_Pages[4].dwSize      = sizeof(PROPSHEETPAGE);
                              l_Pages[4].dwFlags     = PSP_DEFAULT;
                              l_Pages[4].hInstance   = dllInstance;
                              l_Pages[4].pszTemplate = MAKEINTRESOURCE(IDD_AUXBUS_FX);
                              l_Pages[4].pszIcon     = NULL;
                              l_Pages[4].pszTitle    = NULL;
                              l_Pages[4].pfnDlgProc  = (DLGPROC) &AuxBusDialog;
                              l_Pages[4].lParam      = 0;
                              l_Pages[4].pfnCallback = NULL;
                              l_Pages[4].pcRefParent = NULL;

                              l_Pages[5].dwSize      = sizeof(PROPSHEETPAGE);
                              l_Pages[5].dwFlags     = PSP_DEFAULT;
                              l_Pages[5].hInstance   = dllInstance;
                              l_Pages[5].pszTemplat e = MAKEINTRESOURCE(IDD_RX_BROADCAST);
                              l_Pages[5].pszIcon     = NULL;
                              l_Pages[5].pszTitle    = NULL;
                              l_Pages[5].pfnDlgProc  = (DLGPROC) &RxBroadcastDialog;
                              l_Pages[5].lParam      = 0;
                              l_Pages[5].pfnCallback = NULL;
                              l_Pages[5].pcRefParent  = NULL;

                              l_Pages[6].dwSize      = sizeof(PROPSHEETPAGE);
                              l_Pages[6].dwFlags     = PSP_DEFAULT;
                              l_Pages[6].hInstance   = dllInstance;
                              l_Pages[6].pszTemplate = MAKEINTRESOURCE(IDD_ABOUT);
                              l_Pages[6].pszIcon     = NULL;
                              l_Pages[6].pszTitle     = NULL;
                              l_Pages[6].pfnDlgProc  = (DLGPROC) NULL;
                              l_Pages[6].lParam      = 0;
                              l_Pages[6].pfnCallback = NULL;
                              l_Pages[6].pcRefParent = NULL;

                              PROPSHEETHEADER l_Prop_Sheet;

                              l_Prop_Sheet.dwSize      = sizeof(PROPSHEETHEADER);
                              l_Prop_She et.dwFlags     = PSH_DEFAULT | PSH_PROPSHEETPAGE;
                              l_Prop_Sheet.hwndParent  = GetForegroundWindow ();
                              l_Prop_Sheet.hInstance   = dllInstance;
                              l_Prop_Sheet.hIcon       = NULL;
                              l_Prop_Sheet.pszCaption  = (LPCTSTR) "Ninereeds NRS04, By Steve Horne"  ;
                              l_Prop_Sheet.nPages      = 7;
                              l_Prop_Sheet.nStartPage  = 0;
                              l_Prop_Sheet.ppsp        = (LPCPROPSHEETPAGE) l_Pages;
                              l_Prop_Sheet.pfnCallback = NULL;

                              g_mi = this;

                              //DialogBox (dllInstance, MAKEINTRESOURCE (IDD_NRS04), NULL, (DLGPROC) &M  ainDialog);

                              g_Propsheet_HWnd [0] = NULL;
                              g_Propsheet_HWnd [1] = NULL;
                              g_Propsheet_HWnd [2] = NULL;
                              g_Propsheet_HWnd [3] = NULL;  //  Modulator Waveform
                              g_Propsheet_HWnd [4] = NULL;  //  Modulator Envelope

                              PropertySheet ((LPCPROPSHEETHEA DER) &l_Prop_Sheet);

                              //  This is in case of asynchronous events that could affect the property sheet if it
                              //  is displayed, but do not care if the property sheet is not displayed.
                              //
                              //  Not going ahead at present because some assurances are   needed that the access will
                              //  not occur between the property sheet closing and these variables being cleared.
                              //
                              //  Is there a notification that warns of the property sheet (or of a single page) closing?

                              g_mi = NULL;

                              g_Propsheet_HWnd [0 ] = NULL;
                              g_Propsheet_HWnd [1] = NULL;
                              g_Propsheet_HWnd [2] = NULL;
                              g_Propsheet_HWnd [3] = NULL;
                              g_Propsheet_HWnd [4] = NULL;
                    }
                    break;

          }
}
```

```
// Chimp's PitchShifter v1.1a
// Simple buffer-based pitch shifting
// buffer size is adjustable (it's one of the machine's attributes)

// TODO:
//        *         implement some kind of filtering or overlap to remove
//                  unwanted discontinuities (clicks) in the output signal
//                  caused by the writehead overtaking the readhead
//                            (when shifting pitch down)
//                  or by the readhead overtaking the writehead
//                            (when shifting pitch up)
//
//                  (i have already created an attribute to deal with
//                  click removal, it just hasn't been used yet)
//
//        *         some bugs need fixing (namely those concerned with  the
//                  buffer sometimes containing 'old' data and emitting
//                  unwanted sounds)
//
//        *         minor bug: when changing the bufferlength attribute
//                  whilst the machine is playing generates a noticeable
//                  click. i have left this since it is fair to assume yo u
//                  aren't going to change the attributes very often when
//                  a buzzsong is playing!
//
//        *         related bug: this machine only senses a change in
//                  the master sample rate when the machine is stopped.
//                  the reasoning behind this is that it puts a non -trivial
//                  load on the CPU to keep checking that the sample rate
//                  hasn't changed, and also that it's fair to assume you
//                  aren't going to change the samplerate very often when
//                  a buzzsong is playing. just stop the buzzsong, restart it,
//                  and the pitchshifter will work properly again.
//
//        The REAL Todo list:
//
//        *         Make the size of the buffer dependent upon the fundamental
//                  frequency of the input signal (ie, when the frequency of
//                  the input signal changes, the buffer size changes accordingly
//                  to generate a more authentic pitch scaling effect)
//
//        *         Ramping for smoother changes of pitch
//
//        *         Consistency (for example, a pitch shift of zero should
//                  NEVER introduce a delay... at the moment, i have gotten
//                  away with this as a special  case but it should really be
//                  built into the algorithm)
//
//        *         Better pitch shifting algorithm? (eg, using ffts / formants)
//
// (c) 1998 dave hooper @ spc

#ifndef _DEBUG
#pragma optimize ("awy",on)
#endif

#include <windows.h>
#include <math.h>
#include "../MachineInterface.h"


CMachineParameter const paraShift =
{
          pt_word,                                          // type
          "Shift",
          "PitchShift in cents (0 = no shift, 2400 = 2 octaves (24 semitones) shift)",
          0,                                                                    // MinValue
          2400,                                                       // MaxValue
          2500,                                                       // NoValue
          MPF_STATE,                                                  // Flags
          1200                                                        // DefValue
};

CMachineParameter const paraDirection =
{
          pt_byte,                                          // type
          "Direction",
          "Shift Direction (0 = down, 1 = up)",                       // description
          0,                                                                    // MinValue
          1,                                                                    // MaxValue
          2,                                                                    // NoValue
          MPF_STATE,                                                  // Flags
          0                                                                     // DefValue
};

CMachineParameter const paraDry =
{
          pt_byte,                                          // type
          "Dry out",
          "Dry out (00 =  -100%, 40 = 0%, 80 = 100%)",                // description
          0x00,                                                       // MinValue
          0x80,                                                       // MaxValue
          0xff,                                                       // NoValue
          MPF_STATE,                                                  // Flags
          0x80                                                        // DefValue
};

CMachineParameter const paraWet =
{
          pt_byte,                                          // type
          "Wet out",
          "Wet out (00 =  -100%, 40 = 0%, 80 = 100%)",                // description
```

```
        0x00,                                                   // MinValue
        0x80,                                                   // MaxValue
        0xff,                                                   // NoValue
        MPF_STATE,                                              // Flags
        0x80                                                    // DefValue
};


CMachineAttribute const attrBufferlength =
{
        "Buffer Size (ms)",                     // description
        2,                                                              // MinValue
        1000,                                                   // MaxValue
        30                                                              // DefValue
};

CMachineAttribute const attrOverlap =
{
        "Overlap Amount on Heads Passed (%)",                   // description
        0,                                                              // MinValue
        100,                                                    // MaxValue
        0                                                               // DefValue
};


CMachineParameter const *pParameters[] =
{
        &paraShift,
        &paraDirection,
        &paraDry,
        &paraWet
};

CMachineAttribute const *pAttributes[] =
{
        &attrBufferlength,
        &attrOverlap,
};



#pragma pack(1)

class gvals
{
public:
        word shift;
        byte direction;
        byte dry;
        byte wet;
};

class avals
{
public:
        int bufferlength;
        int overlap;
};

#pragma pack()



CMachineInfo const MacInfo =
{
        MT_EFFECT,                              // type
        MI_VERSION,                             // always MI_VERSION
        0,                                              // flags
        0,                                              // min tracks
        0,                                              // max tracks
        4,                                              // numGlobalParameters
        0,                                              // numTrackParameters
        pParameters,
        2,                                              // numAttributes
        pAttributes,
#ifdef _DEBUG
        "Chimp's PitchShifter v1.1 (Debug)",    // name
#else
        "Chimp's PitchShifter v1.1",
#endif
        "PitchShifter",                 // short name
        "Dave Hooper",                  // author
        "&About..."                             // commands
};



// derive a new class for this machine
class mi : public CMachineInterface
{
public:
        mi();
        virtual ~mi();

        virtual void AttributesChanged(void);
        virtual void Command(int const command);
        virtual char const *DescribeValue(int const param, int const value);
        virtual void Init(CMachineDataInput * const pi);
        virtual void Stop(void);
        virtual void Tick(void);
        virtual bool Work(float *psamples, int numsamples, int const mode);

private:
        gvals gval;
```

```
	avals aval;

	// variables concerned with the buffer
	float * buffer; // the buffer itself
	bool buffer_zeroed; // flag to say whether the buffer is em pty - not fully implemented
	unsigned short bufferlength, writeposition; // allows for a buffer up to 64k long
	unsigned long readposition, pitchadjustment, bufferlength_times_64k; // 32 bits allows for a buffer up to 64k long, with 16 bits of
fractional accuracy for fractional positions of the read head

	// variables concerned with keeping track of internal data
	int SamplesPerSec;
	unsigned short Shift;
	short Direction;
	float Wet,Dry;

};



DLL_EXPORTS


mi::mi()
{
	GlobalVals = &gval;
	TrackVals = NULL;
	AttrVals = (int *)&aval;
}


mi::~mi()
{
	// free up any allocated memory in the destructor
	// in this case it's just the buffer
	delete[] buffer;
}


void mi::AttributesChanged(void)
{
	// recalculate the bufferlength and associated variables,
	// reallocate the buffer and set the read and write heads
	// to valid positions within the buffer
	// note:  the way i'm resetting the read and write heads
	//				WILL cause clicking when you change the attributes
	bufferlength = (unsigned short)(SamplesPerSec  * (aval.bufferlength/1000.0f));
	delete[] buffer;
	buffer = new float[bufferlength];

	bufferlength_times_64k = bufferlength << 16;
	writeposition%=bufferlength;
	readposition%=bufferlength_times_64k;
}


void mi::Command(int const command)
{
	// called when the machine's context menu generates a
	// command request. in this case, the only command
	// available is 'About...' so popup an About box

	switch(command)
	{
	case 0:
		MessageBox(NULL,
				"Chimp's PitchShifter v1.1 \n\n"
				"Copyright (c) 1998 no-brain@mindless.com \n\n"
				"If you like and use Chimp's plugins, you can \n"
				"support the author by sending any amount of cash \n"
				"(in any currency) to the following address \n\n"
				"\tdave hooper\n\t2a corringway fleet \n\thants gu13 0an \n\tUK",
				"About PitchShifter", MB_OK|MB_SYSTEMMODAL);
		break;
	default:
		break;
	}
}


char const *mi::DescribeValue(int const param, int const value)
{
	// display information in the machine's parameters dialog.
	// returning NULL means buzz interpret s the value data
	// in its default manner  - currently, buzz's default
	// representation is the number itself

	static char txt[16];

	switch(param)
	{
	case 0:			// shift
		sprintf(txt,"%.2f", (double)(value)/100.0);
		break;
	case 1:			// direction
		switch(value)
		{
		case 0: return "down";
		case 1: return "up";
		default: return NULL;
		}
	case 2:			// dry
	case 3:			// wet
		sprintf(txt, "%.1f%%", 100.0*(((double)(value)/(float)0x40)  -1));
		break;
```

```
          default:  // ...else...
                    return NULL;
          }

          return txt;
}


void mi::Init(CMachineDataInput * const pi)
{

          // called when the machine is LOADED into buzz at STARTUP

          // initialise machine defaults that require no thought at all
          readposition=writeposition=0;
          Shift=gval.shift;
          SamplesPerSec = pMasterInfo ->SamplesPerSec;

          // deal with the possibilty that this machine has replaced
          // a previous version that didn't use attributes
          if (aval.bufferlength < 2) aval.bufferlength=2;

          // initialise other machine defaults
          bufferlength = (unsigned short)(SamplesPe rSec * (aval.bufferlength/1000.0f));
          Direction=(gval.direction)*2 -1;
          Dry=((gval.dry)/(float)0x40) -1;
          Wet=((gval.wet)/(float)0x40) -1;
          pitchadjustment = (unsigned long)(65536.0*pow(2.0,Direction*(Shift/1200.0)));

          // create some 'derived' constants
          bufferlength_times_64k = bufferlength << 16;

          //  allocate the buffer and empty it
          buffer = new float[bufferlength];
          memset(buffer,bufferlength*sizeof(float),0);
          buffer_zeroed = true;

          // the memory just allocated is deallocated in ~mi
          // and also in A ttributesChanged (where it is reallocated
          // with a different size)
}


void mi::Stop(void)
{
          // if the samplerate has changed, update the internal record
          // of the samplerate and call AttributesChanged to force the
          // bufferlength to be recalculated,  since it depends on the
          // current sample rate

          if (pMasterInfo ->SamplesPerSec != SamplesPerSec)
          {
                    SamplesPerSec=pMasterInfo ->SamplesPerSec;
                    AttributesChanged();
          }
}


void mi::Tick(void)
{
          // called every 'tick' generated by the buzz engine
          // all i do here is update the internal data since there might
          // be a pattern in the current buzzsong which controls
          // the PitchShifter

          if (gval.shift != paraShift.NoValue)
          {
                    // recalculate pitchadjustment, based on the new shift
                    Shift = gval.shift;

                    // also have to check direction hasn't changed
                    if (gval.direction != paraDirection.NoValue)
                              Direction = (gval.direction)*2 -1;

                    pitchadjustment = (unsigned long)(65536.0*pow(2.0,Direction*(Shift/1200.0)));
          }
          else if (gval.direction != paraDirec tion.NoValue)
          {
                    // recalculate pitchadjustment, based on the new direction
                    Direction = (gval.direction)*2 -1;
                    pitchadjustment = (unsigned long)(65536.0*pow(2.0,Direction*(Shift/1200.0)));
          }

          // recalculate Dry and Wet
          if (gval.dry != paraDry.NoVal ue)
                    Dry = ((gval.dry)/(float)0x40) -1;

          if (gval.wet != paraWet.NoValue)
                    Wet = ((gval.wet)/(float)0x40) -1;

}


bool mi::Work(float *psamples, int numsamples, int const mode)
{

          // called repeatedly when a buzzsong is playing

          // this machine is not  a generator, so it only
          // does work if it has an input AND an output.

          if (!(mode & WM_READ))
          {
                    // if it has no output, then we're ok to ignore the
          // input too
          if (!buffer_zeroed)
          {
                    buffer_zeroed=true;
                    memset(buffer,bufferlength*sizeof(f loat),0);
          }

          // tell buzz we've effectively done nothing
          return false;
}

if (mode == WM_READ)
{
          // if we ONLY have output (no input) then there IS
          // no input so empty the buffer
          if (!buffer_zeroed)
          {
                    buffer_zeroed=true;
                    memset(buffer,bufferlength*sizeof(float),0);
          }

          // and tell buzz we've effectively done nothing
          return false;
}


// else, perform the pitch shift

// create local copies of certain constants, because they
// can be accessed quicker this way

unsigned short const BUFFERLENGTH=bufferlength;
unsigned long const BUFFERLENGTH_TIMES_64K=bufferlength_times_64k;

if (Shift!=0 && Wet!=0)
{
          // in the case where the parameters of the machine are
          // such that there IS a pitch shift (ie, Shift is not
          // set to zero) and there IS some wet output, then
          // pitch shifting takes place here

          // signify that the buffer is not empty and that it's
          // contents are important
          buffer_zeroed = false;

          // generate the output based on the input
          // on input psamples points to the input data, which
          // is ALSO the output data. in other words, to 'effect' the
          // input data, we just change the contens of the buffer
          // pointed to by psamples

          do
          {
                    // write to buffer position currently under write head
                    buffer[writeposition] = *psamples;
                    // advance write head by one position
                    // write head wraps around from the end of the buffer to the beginning
                    writeposition=(writeposition+1)%BUFFERLENGTH;

                    // generate the output as the sum of the unaffected i nput
                    *psamples = (float)((Dry* (*psamples))+
                    // and the data currently under the read head
                                              (Wet * buffer[readposition >> 16]));

                    // note: the top 16 bits of readposition index into the
                    // buffer; the bottom 16 bits of readposition are fo r increased
                    // accuracy when moving the read head

                    // advance the input pointer to the next sample
                    psamples++;

                    // advance the read head by the amount determined by
                    // the current contents of the pitchadjustment register
                    // read head wrap s around from the end of the buffer to the beginning
                    readposition=(readposition+pitchadjustment) % (BUFFERLENGTH_TIMES_64K);

          } while (--numsamples);
}
else
{
          // no effective shift so simply adjust amplitude of output
          // by the factor Wet+Dry
          // also fill buffer, so make a note that the buffer contents
          // are still important to us

          buffer_zeroed = false;

          float const WD = Wet + Dry;

          do
          {
                    // write to buffer position currently under write head
                    buffer[writeposition] = *psamples;
                    // advance write head by one position
                    // write head wraps around from the end of the buffer to the beginning
                    writeposition=(writeposition+1)%BUFFERLENGTH;

                    // adjust the amplitude of the output signal (no pitch
                    // shifting involved, so we d on't even need to worry
                    // about the read head)
                    *psamples = (float)(WD* (*psamples));

                    // advance the input pointer to the next sample
                    psamples++;
                    } while (--numsamples);

          }

          return true;
}


#pragma optimize ("", on)
```

```
#include <string.h>
#include <stdlib.h>
#include <assert.h>
#include <math.h>
#include <float.h>
#include "../MachineInterface.h"
#include "../dsplib/dsplib.h"

#pragma optimize ("a", on)

#define MAX_TRACKS                                8

#define EGS_ATTACK                                0
#define EGS_SUSTAIN                               1
#define EGS_RELEASE                               2
#define EGS_NONE                                  3

#define MIN_AMP                                             (0.0001 * (32768.0 / 0x7fffffff))

double const oolog2 = 1.0 / log(2);

CMachineParameter const paraWForm =
{
          pt_byte,                                          // type
          "Waveform",
          "Shape of waveform",                    // description
          1,                                                                    // MinValue
          7,                                                                    // MaxValue
          0,                                                                    // NoValue
          MPF_STATE,                                                  // Flags
          1
};

CMachineParameter const paraAttack =
{
          pt_word,                                          // type
          "Attack",
          "Attack half life in ms",                                                       // description
          1,                                                                    // MinValue
          0x0800,                                                     // MaxValue
          0,                                                                    // NoValue
          MPF_STATE,                                                  // Flags
          500
};

CMachineParameter const paraDecay =
{
          pt_word,                                          // type
          "Decay",
          "Decay half life in ms",                                                        // description
          1,                                                                    // MinValue
          0x0800,                                                     // MaxValue
          0,                                                                    // NoValue
          MPF_STATE,                                                  // Flags
          500
};

CMachineParameter const paraEffectLow =
{
          pt_word,                                          // type
          "low effect",
          "Fractal Effect (Low)",                 // description
          0,                                                                    // MinValue
          0xfffe,                                                     // MaxValue
          0xffff,                                                     // NoValue
          MPF_STATE,                                                  // Flags
          0x8000,
};

CMachineParameter const paraEffectHigh =
{
          pt_word,                                          // type
          "high effect",
          "Fractal Effect (High)",                // description
          0,                                                                    // MinValue
          0xfffe,                                                     // MaxValue
          0xffff,                                                     // NoValue
          MPF_STATE,                                                  // Flags
          0x8000,
};

CMachineParameter const paraDepth =
{
          pt_byte,                                          // type
          "depth",
          "Fractal Depth",                        // description
          0,                                                                    // MinValue
          10,                                                                   // MaxValue
          0xff,                                                       // NoValue
          MPF_STATE,                                                  // Flags
          0x01
};

CMachineParameter const paraNote =
{
          pt_note,                                          // type
          "Note",
          "Note",                                                     // description
          NOTE_MIN,                                         // MinValue
          NOTE_OFF,                                         // MaxValue
          NOTE_NO,                                          // NoValue
```

```
        0,                                                              // Flags
        0
};

CMachineParameter const paraVolume =
{
        pt_byte,                                        // type
        "Volume",
        "Volume [sustain level] (0=0 %, 80=100%, FE=~200%)",        // description
        0,                                                              // MinValue
        0xfe,                                                   // MaxValue
        0xff,                                           // NoValue
        MPF_STATE,                                              // Flags
        0x80
};


CMachineParameter const *pParameters[] = {
        // global
        &paraWForm,
        &paraAttack,
        &paraDecay,
        &paraEffectLow,
        &paraEffectHigh,
        &paraDepth,
        // track
        &paraNote,
        &paraVolume
};

#pragma pack(1)

class gvals
{
public:
        byte wform;
        word attack;
        word decay;
        word effect_low;
        word effect_high;
        byte depth;

};

class tvals
{
public:
        byte note;
        byte volume;

};

#pragma pack()

CMachineInfo const MacInfo =
{
        MT_GENERATOR,                   // type
        MI_VERSION,
        0,                              // flags
        1,                              // min tracks
        MAX_TRACKS,                     // max tracks
        6,                              // numGlobalParameters
        2,                              // numTrackParameters
        pParameters,
        0,
        NULL,
#ifdef _DEBUG
        "Soft Tone 2 (Debug build)",    // name
#else
        "Soft Tone 2",
#endif
        "Softy2_",                      // short name
        "Steve Horne",                  // author
        NULL
};

class mi;

class mi;

class CTrack
{
public:
        void Tick(tvals const &tv);
        void Stop();
        void Reset();
        void Generate(float *psamples, int numsamples);

    double Fractal (double p, double q);

        void Sine     (float *psamples, int numsamples);
        void Square   (float *psamples, int numsamples);
        void Triangle (float *psamples, int numsamples);
        void Ramp     (float *psamples, int numsamples);
        void Hex      (float *psamples, int numsamples);
        void Jaggy    (float *psamples, int numsamples);
        void AltRamp  (float *psamples, int numsamples);

public:

        double Freq;
        double TargetAmp;
        int    NoteOn;
```

```
          double CurrPhase;
          double CurrAmp;

          mi *pmi;
};

class mi : public CMachineInterface
{
public:
          mi();
          virtual ~mi();

          virtual void Init(CMachineDataInput * const pi);
          virtual void Tick();
          virtual bool Work(float *psamples, int numsamples, int const mode);
          virtual void SetNumTracks(in t const n) { numTracks = n; }
          virtual void Stop();

public:
          int numTracks;
          CTrack Tracks[MAX_TRACKS];

          tvals tval[MAX_TRACKS];
          gvals gval;

          byte   WForm;
          double DecayA;
          double AttackA;
          double AttackB;

          double EffectHigh;
          double EffectLow;

          byte Depth;
};

DLL_EXPORTS

mi::mi()
{
          GlobalVals = &gval;
          TrackVals  = tval;
          AttrVals   = NULL;
}

mi::~mi()
{

}

void CTrack::Reset()
{
          NoteOn    = 0;
          Freq      = 0.0;
          TargetAmp = 1.0;
          CurrPhase = 0.0;
          CurrAmp   = 0.0;
}

double CTrack::Fractal (do uble p, double q)
{
          float s = (p + 1.0) / 2.0;

          double Effect  = pmi ->EffectLow + ((pmi ->EffectHigh - pmi->EffectLow) * q);
          double EffectB = 3.0 - (3.0 * Effect);
          double EffectA = 1.0 - (Effect + EffectB);

          if (s < 0.0)
          {
                    s = 0.0;
          }
          else if (s > 1.0)
          {
                    s = 1.0;
          }

          float ss;
          int n = pmi ->Depth;

          while (n-- > 0)
          {
                    ss = s * s;

                    s =   (EffectA * ss * s)
                              + (EffectB * ss    )
                              + (Effect  * s     );
          }

          return (s * 2.0) - 1.0;
}


void mi::Init(CMachineDataInput * const pi)
{
          WForm = 1;

          AttackA = pow(2.0, -1000.0 / (((double) pMasterInfo ->SamplesPerSec) * 500.0));
          AttackB = 1.0 - AttackA;

          DecayA = pow(2.0, -1000.0 / (((double) pMasterInfo ->SamplesPerSec) * 500.0));

          EffectHigh = (((float) 32768) * 9.0) / ((float) 65534);
          EffectLow  = (((float) 32768) * 9.0) / ((float) 65534);
          Depth  = 1;

          for (int c = 0; c < MAX_TRACKS; c++)
          {
                    Tracks[c].pmi = this;
                    Tracks[c].Reset();
          }

}

void CTrack::Tick(tvals const &tv)
{
          if (tv.volume != paraVolume.NoValue)
                    TargetAmp = (float)(tv. volume * (1.0 / 0x80));

          if (tv.note == NOTE_OFF)
          {
                    NoteOn = 0;
          }
          else if (tv.note != NOTE_NO)
          {
                    NoteOn = 1;

                    int l_Note = ((tv.note >> 4) * 12) + (tv.note & 0x0f)  - 70;

                    Freq = (440.0 * pow (2.0, ((float) l_Note) / 12.0)) / pmi ->pMasterInfo->SamplesPerSec;
          }
}

void CTrack::Generate(float *psamples, int numsamples)
{
          int          c = numsamples;
          float *p = psamples;

          if (NoteOn)
          {
                    do
                    {
                              *p = CurrAmp;
                              CurrAmp = (CurrAmp * pmi ->AttackA) + (TargetAmp * pmi ->AttackB);
                              p++;
                    } while (--c > 0);
          }
          else
          {
                    do
                    {
                              *p = CurrAmp;
                              CurrAmp *= pmi->DecayA;
                              p++;
                    } while (--c > 0);
          }

          switch (pmi->WForm)
          {
                    case 1 :
                              Sine (psamples, numsamples);
                              break;
                    case 2 :
                              Triangle (psamples, numsamples);
                              break;
                    case 3 :
                              Jaggy (psamples, numsamples);
                              break;
                    case 4 :
                              Hex (psamples, numsamples);
                              break;
                    case 5 :
                              Square (psamples, numsamples);
                              break;
                    case 6 :
                              Ramp (psamples, numsamples);
                              break;
                    case 7 :
                              AltRamp (psamples, numsamples);
                              break;
          }

          c = numsamples;
          p = psamples;

          do
          {
                    *p *= 32768.0;  p++;
          } while (--c > 0);
}

void CTrack::Sine(float *psamples, int numsamples)
{
          do
          {
                    *psamples *= Fractal (sin (CurrPhase * 2.0 * PI), *psamples);
                    CurrPhase += Freq;
                    psamples++;
          } while (--numsamples > 0);

          while (CurrPhase >= 1.0)
                    CurrPhase -= 1.0;
}

void CTrack::Square(float *psamples, int numsamples)
{
          do
          {
```

```
                    while ((numsamples > 0) && (CurrPhase < 0.5))
                    {
                              CurrPhase += Freq;
                              psamples++;
                              numsamples--;
                    }

                    while ((numsamples > 0) && (CurrPhase < 1.0))
                    {
                              *psamples = -(*psamples);
                              CurrPhase += Freq;
                              psamples++;
                              numsamples--;
                    }

                    while (CurrPhase >= 1.0)
                    {
                              CurrPhase -= 1.0;
                    }

          } while (numsamples > 0);
}

void CTrack::Triangle(float *psamples, int  numsamples)
{
          do
          {
                    while ((numsamples > 0) && (CurrPhase < 0.25))
                    {
                              *psamples *= Fractal (CurrPhase * 4, *psamples);
                              CurrPhase += Freq;
                              psamples++;
                              numsamples--;
                    }

                    while ((numsamples > 0) && (CurrPhase < 0.75))
                    {
                              *psamples *= Fractal (2.0 - (CurrPhase * 4), *psamples);
                              CurrPhase += Freq;
                              psamples++;
                              numsamples--;
                    }

                    while ((numsamples > 0) && (CurrPhase < 1.25))
                    {
                              *psamples *= Fractal ((CurrPhase * 4)  - 4.0, *psamples);
                              CurrPhase += Freq;
                              psamples++;
                              numsamples--;
                    }

                    while (CurrPhase >= 1.0)
                    {
                              CurrPhase -= 1.0;
                    }

          } while (numsamples > 0);
}

void CTrack::Ramp(float *psamples, int numsamples)
{
          do
          {
                    while ((numsamples > 0) && (CurrPhase < 1.00))
                    {
                              *psamples *= Fractal ((CurrPhase *  2) - 1.0, *psamples);
                              CurrPhase += Freq;
                              psamples++;
                              numsamples--;
                    }

                    while (CurrPhase >= 1.0)
                    {
                              CurrPhase -= 1.0;
                    }

          } while (numsamples > 0);
}

void CTrack::AltRamp(float *psamples, int numsamples)
{
          do
          {
                    while ((numsamples > 0)  && (CurrPhase < 0.5))
                    {
                              *psamples *= Fractal (CurrPhase * 2, *psamples);
                              CurrPhase += Freq;
                              psamples++;
                              numsamples--;
                    }

                    while ((numsamples > 0) && (CurrPhase < 1.0))
                    {
                              *psamples *= Fractal (1.0  - (CurrPhase * 2), *psamples);
                              CurrPhase += Freq;
                              psamples++;
                              numsamples--;
                    }

                    while (CurrPhase >= 1.0)
                    {
                              CurrPhase -= 1.0;
                    }
          } while (numsamples > 0);
}

void CTrack::Hex(float *psamples, int numsamples)
{
          do
          {
                    while ((numsamples > 0) && (CurrPhase < 0.125))
                    {
                              *psamples *= Fractal (CurrPhase * 8, *psamples);
                              CurrPhase += Freq;
                              psamples++;
                              numsamples--;
                    }

                    while ((numsamples > 0) && (CurrPhase < 0.375))
                    {
                              CurrPhase += Freq;
                              psamples++;
                              numsamples--;
                    }

                    while ((numsamples > 0) && (CurrP hase < 0.625))
                    {
                              *psamples *= Fractal (4.0  - (CurrPhase * 8), *psamples);
                              CurrPhase += Freq;
                              psamples++;
                              numsamples--;
                    }

                    while ((numsamples > 0) && (CurrPhase < 0.875))
                    {
                              *psamples =  -(*psamples);
                              CurrPhase += Freq;
                              psamples++;
                              numsamples--;
                    }

                    while ((numsamples > 0) && (CurrPhase < 1.125))
                    {
                              *psamples *= Fractal ((CurrPhase * 8)  - 8.0, *psamples);
                              CurrPhase += Freq;
                              psamples++;
                              numsamples--;
                    }

                    while (CurrPhase >= 1.0)
                    {
                              CurrPhase  -= 1.0;
                    }

          } while (numsamples > 0);
}

void CTrack::Jaggy(float *psamples, int numsamples)
{
          do
          {
                    while ((numsamples > 0) && (CurrPhase < 0.125))
                    {
                              *psamples *= Fractal (CurrPhase * 8, *psamples);
                              CurrPhase += Freq;
                              psamples++;
                              numsamples--;
                    }

                    while ((numsamples > 0) && (CurrPhase < 0.25))
                    {
                              *psamples *= Fractal (2.0  - (CurrPhase * 8), *psamples);
                              CurrPhase += Freq;
                              psamples++;
                              numsamples--;
                    }

                    while ((numsamples > 0) && (CurrPhase < 0.375))
                    {
                              *psamples *= Fractal ((CurrPh ase * 8) - 2.0, *psamples);
                              CurrPhase += Freq;
                              psamples++;
                              numsamples--;
                    }

                    while ((numsamples > 0) && (CurrPhase < 0.625))
                    {
                              *psamples *= Fractal (4.0  - (CurrPhase * 8), *psamples);
                              CurrPhase += Freq;
                              psamples++;
                              numsamples--;
                    }

                    while ((numsamples > 0) && (CurrPhase < 0.75))
                    {
                              *psamples *= Fractal ((CurrPhase * 8)  - 6.0, *psamples);
                              CurrPhase += Freq;
                              psamples++;
                              numsamples--;
                    }

                    while ((numsamples > 0) && (CurrPhase < 0.875))
                    {
                              *psamples *= Fractal (6 .0 - (CurrPhase * 8), *psamples);
                              CurrPhase += Freq;
                              psamples++;
                              numsamples--;
                    }

                    while ((numsamples > 0) && (CurrPhase < 1.125))
                    {
                              *psamples *= Fractal ((CurrPhase * 8)  - 8.0, *psamples);
                              CurrPhase += Freq;
                              psamples++;
                              numsamples--;
                    }

                    while (CurrPhase >= 1.0)
                    {
                              CurrPhase -= 1.0;
                    }

          } while (numsamples > 0);
}

void mi::Tick()
{
          if (gval.wform != paraWForm.NoValue)
          {
                    WForm = gval.wform;
          }

          if (gval.attack != paraAttack.NoValue)
          {
                    AttackA = pow(2.0,  -1000.0 / (((double) pMasterInfo->SamplesPerSec) * ((double) gval.attack)));
                    AttackB = 1.0  - AttackA;
          }

          if (gval.decay != paraDecay.NoValue)
          {
                    DecayA = pow(2.0,  -1000.0 / (((double) pMasterInfo->SamplesPerSec) * ((double) gval.decay)));
          }

          if (gval.effect_high != paraEffectHigh.NoValue)
          {
                    EffectHigh = (((float) gval.effect_high) * 9.0) / ((float) 65534);
          }

          if (gval.effect_low != paraEffectLow.NoValue)
          {
                    EffectLow = (((float) gval.effect_low) * 9.0) / ((float) 65534);
          }

          if (gval.depth != paraDepth.NoValue)
                    Depth = gval.depth;

          for (int c = 0; c < numTracks; c++)
                    Tracks[c].Tick(tval[c]);

}

bool mi::Work(float *psamples, int numsamples, int const)
{
          bool gotsomething = false;

          if (numTracks > 0)
          {
                    Tracks[0].Generate(psamples, numsamples) ;
          }

          for (int c = 1; c < numTracks; c++)
          {
                    float *paux = pCB->GetAuxBuffer();
                    Tracks[c].Generate(paux, numsamples);

                    DSP_Add(psamples, paux, numsamples);
          }

          return true;
}

void CTrack::Stop()
{
          NoteOn  = 0;
          CurrAmp = 0.0;
}

void mi::Stop()
{
          for (int c = 0; c < numTracks; c++)
                    Tracks[c].Stop();
}
```