DSP functions in DAC8PRO (and DAC8STEREO)

Summary:
The digital boards of DAC8 products are built around an XMOS XU216 device which has extra cores and powerful instructions to perform DSP functions like filtering, cross over or delay. By using the AVDSP framework and its utilities, it is possible to create a DSP program to be uploaded in the DAC8 products. This document explains how this works and provides step by step examples.

DAC8PRO and DAC8STEREO DSP firmwares :
By default, DAC8PRO executes two firmwares. One for Xmos XU216 in charge of USB interface and data transmission with DAC and AES inputs. One for managing the front-panel (volume, display, IR remote…) and adjusting DAC registers according to the USB host sample rate.
In order to enable DSP functionalities, a specific version of the XU216 firmware must be installed with the DFU upgrade utility. Then it will be possible to manually upload DSP programs that will interact with internal data flows existing between DAC, AES and USB host in and out.
The DSP enabled DAC8PRODSP firmware is fully compatible with the latest standard DAC8PRO V1.6 firmware and can also run without any DSP program activated.

For DAC8PRO, two versions are provided. When 8 AES channels are required, the power available for the DSP program is limited to 250 MIPS spread over 4 cores max. When stereo AES inputs are acceptable, the power available for the DSP program is 462 MIPS spread over 7 cores max. The maximum sample rate is configured at 96khz instead of 192k.
For DAC8STEREO a single version is provided with 400 MIPS spread over 4 cores. The maximum sample rate is configured at 192khz instead of 384k.

Summary of firmwares capabilities:

File Name and USB device name	USB OUT	USB IN	DAC	AES spdif	CORES x mips	MIPS (total)	total instructions 48k	total instructions 96k	total instructions 192k
DAC8PRODSP4	8	8	8	8	1 x 106 2 x 88 3 x 75 4 x 66	106 176 226 264	5500	2750	1375
DAC8PRODSP7	8	8	8	2	1..4 x 106 5 x 88 6 x 75 7 x 66	422 440 453 462	9625	4812	2406
DAC8STEREODSP	2	2	2	2	1..4 x 100	400	8333	4166	2083

Remark : the DSP enabled firmwares also accepts 44100 or 88200 hz. for DAC8PRO a specific 192000 version could be proposed on demand for integrators, depending on requirement and compatibility with other timings required internally.
DSD playback is not possible (silence) when a DSP program is selected.
If the target DSP program takes too long to accommodate the time between two audio samples (10us at 96k), then the DAC is dynamically reconfigured at 44/48khz and a decimation by 2 is done. To avoid THD and aliasing due to this downsampling, it is better to send the audio at a sample rate compatible with the maximum rate supported by the DSP program activated.
Upgrading the unit with the DAC8PRODSP firmware.

As of version 1.5, the DAC8PRO can be upgraded with the windows DFU utility with following steps:
Connect the USB cable, launch the DFU utility, select the binary file dac8prodsp4.bin or dac8prodsp7.bin or dac8stereodsp4.bin clicking on Browse button, then click Start button. The DAC8PRO will reboot in a special USB DFU mode and the utility will start flashing the firmware with an indicator showing the progress. The display will show 0 at the beginning and during about 8 seconds which is the time required to erase the upper part of the flash memory (the factory installed firmware is not erased). Then the counter will progress till the end of the process. Once complete, press the volume button to start the display and verify the presence of the **DSP Prog** selection in the **Filter** menu.

If the flash process fails in the middle, or if the USB cable is disconnected when the counter is progressing, then the DAC8PRO will reboot and restore the original embedded factory version. This is also a normal way to revert back your unit to its original non-dsp firmware.

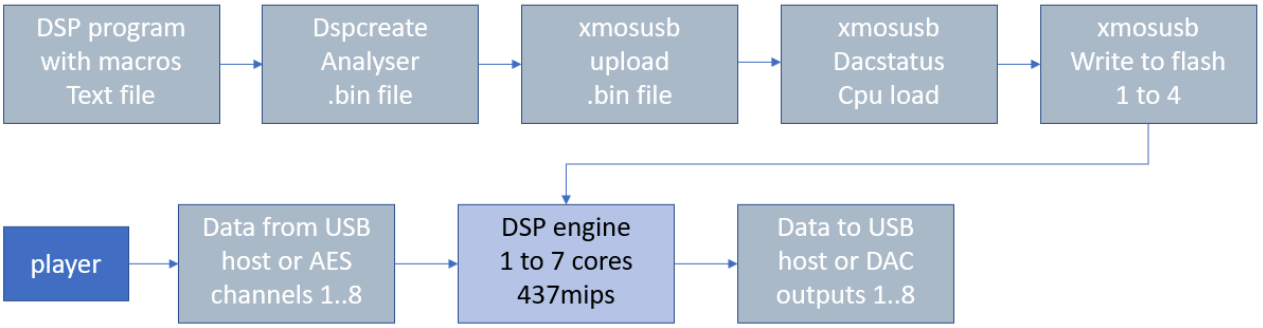
Additional command line utilities are provided, typically for Linux or Mac OSX, to upgrade the firmware.
When a DSP program is selected in the **Filter / DSP Prog** menu, it will always appear on the right display in the top left corner of the screen:

This is important information for the user, as the DSP program may be used to send specific frequencies or volume on some channels. To avoid any situation subject to damaging the drivers, always verify that the expected DSP program is loaded in the DAC8PRO memory with this indicator before playing music, especially at high volume.

Creating a DSP program:
At the time of writing, no graphical user interface is provided to create a DSP flow.
It will require the following steps:

1. defining and organizing the actions expected as a flow of macro instructions that will be executed at each audio sample
2. evaluating cpu load for this flow and grouping / distributing macro-instructions per core
3. using a text editor to write this program in sequences with parameters and code sections
4. converting this file as a binary file with “dspcreate” utility from AVDSP framework
5. uploading the resulting binary file with “xmosusb” command line utility.
6. verifying DAC8PRO status and XMOS cpu load for each core using “xmosusb” utility
7. testing the expected behavior with third party tools like REW using loopback instructions to verify for each channel response, total gains, and verifying clipping behavior.

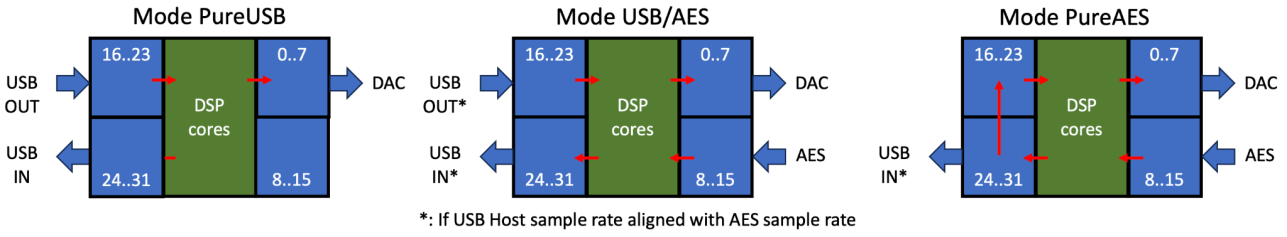
This looks like an intensive effort at first but this gives the possibility to optimize the treatment and to fit a comfortable crossover solution within a limited MIPS quantity. Routing and mixing is also very flexible and efficient as there is no predefined matrix or framework to follow.



The generic behavior of a dsp flow is to : load a sample, treat it, eventually delay it, store it.
A sample can be loaded in the DSP accumulator from any AES input or from any channel provided by the USB host. 16 memory locations are predefined for this.
Transformation can be applied to the DSP accumulator with a set of predefined macro instructions like gain or biquad or saturation.
Then the accumulator can go through a fifo pipeline for applying delays (in microseconds).
The resulting accumulator will be stored as an output sample either for the DACs or for the USB host, represented by 16 other predefined memory locations.
Any core can work with any of these 32 (16 input + 16 outputs) predefined memory locations.

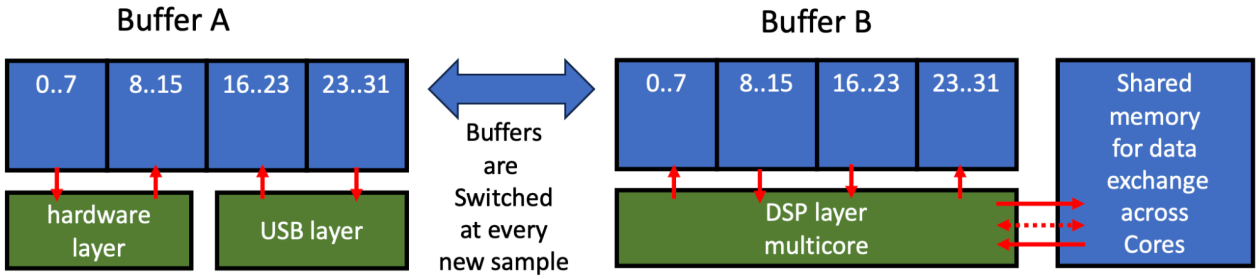
Additional memory locations can be created to share value across cpu cores. For example a core can prepare a stereo input signal with gain and peak correction and save the result in a shared memory location for the other cores to act as a crossover for 8 channels.

The following diagram represents the 32 memory location according to the 3 DAC modes:



*: If USB Host sample rate aligned with AES sample rate

In order to interface the DSP layer with the USB-DAC-AES layer through these 32 memory locations, a mechanism of double buffering is implemented. This guarantees perfect timing across channels and cores (at the expense of 2 samples delay).
The hardware layer interacts only with buffer A, and the DSP layer interacts only with buffer B. and buffer A and B are switched at every sample.
The following diagram represents the buffer management:



MIPS needed for a typical project
A table in this document represents the list of macro instructions available and the time taken to execute them in native cpu core instructions, which is to be compared directly with MIPS available per core or in total.
As an example, a typical flow for filtering and delaying one channel needs 4 macro instructions, and requires 176 core instructions for 6 biquad sections (e.g. Linkwitz-Riley 4th order followed by 4 peak corrections).

The table below summarizes the cpu instructions available per core depending on the sample rate and depending on the MIPS allocated to a core (which depends on the total number of active cores in the program). An indication is given for a maximum number of typical treatments as above.

MIPS per core	48000hz		96000hz		192000hz	
105,6	2199	12	1099	6	549	3
88,0	1833	10	916	5	458	2
75,4	1570	8	785	4	392	2
66,0	1374	7	687	3	343	1

Remark : To maximize the total MIPS capacity, it is recommended to use multiple cores if possible when the program can be split in similar sections (like a 2x3 crossover solution).
Using 4 cores is optimal, as an example for a loudspeaker crossover solution, one core can prepare the stereo left and right, applying gains and equalization, each of the 3 other cores can filter, equalize and adjust delay for 2x3 drivers. Other combinations could be done with a subwoofer or to correct 2 headset outputs in addition.

DSP program:
A DSP program built with the AVDSP framework is a list of sequential binary values (called “opcode”) that will be interpreted one-by-one by a specific runtime library optimized for XMOS XU216 and included within the DAC8PRO DSP firmware.
The opcodes are low-level DSP-like instructions (about 60 different), mainly working on an accumulator or transferring data across memory.

The binary file also contains constant definitions like filters parameters or mixer/gain/delay, which are all computed upfront to cope with all possible audio sample rate (no any SRC involved).
The DAC8PRO front-panel does not provide any mechanisms to modify the filters or gain dynamically during audio listening. All values are embedded in the DSP program binary file and this can be changed only by modifying and uploading a new version of the DSP program.

In order to generate a DSP binary file, a specific command-line utility called `dspcreate` is provided. It accepts either inputs:

A. A text file representing the DSP flow with a predefined set of simple macro-instruction, or
B. An object file generated by a C compiler like GCC using the core AVDSP library, enabling full capabilities and extended syntax.

Both methods are compatible with the DAC8PRO DSP firmware but for simplifying the process, this document describes and focuses on using a text file with a limited set of predefined macro-instructions as an input.

Analyzing a text file and generating a DSP binary program is done with the following command line:

```
OS prompt > dspcreate -dspformat 2 -fsmax 96000 -dsptext
myprogram.txt -binfile mybinfile.bin <param>
```

where `<param>` is optional and describes a list of couples Label=value which can be used as preprocessing information for the DSP program. For example a cutoff frequency can be passed as `Fc=800` on the command line and will be treated before analyzing `myprogram.txt`.

As explained with more details in next chapters, the binary file generated will be uploaded in the DAC8PRO RAM memory with the command-line utility “xmosusb” with the following command:

```
OS prompt > xmosusb --dspload mybinfile.bin
```

The status of the DAC after this upload will be given by the command:

```
OS prompt > xmosusb --dacstatus
```

providing the following informations as an example:

maximum dsp tasks = 4
dsp 1: instructions = 131
dsp 2: instructions = 66
dsp 3: instructions = 66
dsp 4: instructions = 0
maxi instructions = 131 / 520 = 25%fs

DSP program structure:

The text file passed as a parameter with the selector `-dsptext` has to be structured with following sections:

- label definitions for documenting constant parameters (not stored in the final binary file)
- label definitions for filters, or memory parameters which are stored in the binary file
- code sections with macro-instruction for each core.

This structure can be repeated as much as needed according to the maximum number of cores available. The 2 first optional sections can be grouped at the beginning of the file or spread along if they are specific to a core.

Label value definition:
A DSP program is easier to write and verify when defining labels which represent physical locations or constants to be used in different points of the code section. For example, DAC8PRO supports 8 inputs from the USB host (coming from the player), 8 inputs from the AES channels, 8 outputs to the DAC channels, 8 output to the USB host (for recording). It is a good practice (but not mandatory) to start describing these input-outputs with some nicknames defined with their corresponding memory locations. As an example:

```
LeftIn16 ; RightIn 17 ; LeftOut 10 ; RightOut 13 ### USB and DAC IOs
LeftLowOut  =LeftOut ; LeftMidOut =LeftOut+1 ; LeftHighOut =LeftOut+2
RightLowOut =RightOut[0] ; RightMidOut =RightOut[1] ; RightHighOut =RightOut[2]
LeftHeadphone 8 ; RightHeadphone  9
firstGain   -3db
Low_fc400 ; Mid_fc 2000
```

As seen in this example, a label can be defined directly with a numerical value (eg `LeftIn 16` or `firstGain -3db`), or with an expression (+,-,*,/) starting with "=". Multiple definitions can be done on a line with ";" separator. Comments are possible with "#" preceding character.
Remark: when an expression starts with a decibel value, then operators are limited to + and -, and next values in the expression are expected to be also decibel (postfix db).
Numbers can be written in decimal, binary starting with a letter "b" or hexadecimal with "x".

Label parameter definition:
There are 2 types of static parameters :filters, or memories.

Filters describe a list of filters to be computed in a row with a biquad routine. The biquad coefficients are computed upfront according to a list of predefined filter names covering most of the usual requirements (Bessel, Butterworth, Linkwitz-Riley, All-pass, peak, notch…). Generic filters can also be combined to form other filters. As an example:

```
LowPass      LPLR4(Low_fc)
MidPass      HP2(Low_fc, 0.7) HP2(Low_fc, 0.7) LPBU3(Mid_fc)
HighPass     HPBU3(Mid_fc, -2db) HS2(5000, 0.5, +2db)
RoomMode     NOTCH(50, 10) PEAK(80, 0.5, -2db)
```

This represents the settings for a typical 3 way crossover:

- `LowPass` is simply a Linkwitz-Rilley 4th order low pass filter with a cutoff frequency of "Low_fc" (at -6db by design)
- `MidPass` is also a Linkwitz-Rilley 4th order high pass filter but made of 2 successive 2nd order high-pass filters (by definition of an HPLR4 using two HP2 with Q=0.7), followed by a low pass Butterworth 3rd order filter with cutoff frequency Mid_fc (at -3db by design).
- `HighPass` is an high-pass Butterworth 3rd order with an embedded attenuation of -2db after it, followed by a second order high-shelf filter (with Q=0.5 here) with a gain of+2B after 5000hz.
- `RoomMode` defines a notch filter at 50hz with a strong Q=10, and a -2db reduction with a peak correction at 80hz with a relatively large Q=0.5.

These 4 filter labels can be used later in the core code section as parameters of the "biquad" macro instruction. Grouping them at the beginning of the program is easier to maintain.

Remark: any filter can be set with an optional gain (or reduction) parameter. Then its biquad coefficients (b0,b1,b2) will be scaled accordingly by the encoder.

Memory describes a target memory location which can be used to exchange data between cores. As an example, a core can be used for pre-conditioning a stereo signal and then the other core will use these values instead of directly using the 32 Input/output memory locations.

```
stereoMem    MEMORY      2
monoChan     MEMORY
```

This example defines 2 memory locations to store 2 channels related to a stereo signal and a single memory location to store a mono channel.

Code and core section:
a DSP code section starts with the keyword `core`. All the macro instructions are written one by one on the following lines. multiple instructions can be written on a single line with a ";" separator if this brings better visibility. This is an example for treating the Low channels as per the example parameters above:

```
core
      inputgain    (LeftIn, firstGain)
      biquadLowPass
      outputLeftLowOut
      inputgain    (RightIn, firstGain)
      biquadLowPass
      outputRightLowOut
end
```

The same code could be duplicated for treating the Mid channels, inside this core or in another core code section. The keyword `end` must be unique in the program. End of a core section is implicit when a new core instruction is encountered.

This program below provides an example with one preconditioning core in charge of stereo and headphones, and one other core in charge of the Low filtering. This can be merged or duplicated for mid and high filtering.

```
core
      mixer (LeftIn, -6db) (RightIn, -6db)
      savemem      monoChan
      inputgain    (LeftIn, firstGain)
      biquadRoomMode
      savexmem     stereoMem[ 0 ]
      inputgain    (RightIn, firstGain)
      biquadRoomMode
      savexmem     stereoMem[ 1 ]
      transfer     (LeftIn, LeftHeadphone) (RightIn,RightHeadphone)
core
      loadxmem     stereoMem[ 0 ]
      biquadLowPass
      outputLeftLowOut
      loadxmem     stereoMem[ 1 ]
      biquadLowPass
      outputRightLowOut
```

```
end
```

By default, any `output` or `outputgain` instruction will clip the signal between (+1..-1) if the value of the accumulator is outside these bounds. Still this behavior is not satisfying as it generates a high level of harmonics. Therefore it is highly recommended to tune the gain chain to avoid any clipping situation. See also possibility to use saturate instruction as explained later in the document.
Remark : the DAC volume is independent of the DSP and acts directly on the DAC outputs.

Instruction details:

keyword	description	parameters
core	Define the beginning of a new code section for a new DSP core. A core can be active or not depending on a flexible 32 bits condition checked with 1 or 2 parameters. See detailed information later in this document.	optional 1 or 2 parameters representing core condition. First parameter represents the bits set to 1 compatible with this core. The second represents the bit set to 0 compatible with this core.
end	mark the end of the whole DSP program.	no parameter
transfer	load a sample from a memory location and save it to another.	(in, out) …
input	Load a sample from a memory location directly in the main DSP accumulator, without applying conversion or gain.	immediate numerical value or expression representing one of the 32 locations.
output	Save the main DSP accumulator in a memory location directly. Value is always clamped between -1 .. +1	immediate numerical value or expression representing one of the 32 locations.
inputgain	Load a sample from a memory location in the main DSP accumulator and apply a gain or reduction. When multiple couples (in,gain) are provided, they are added all together.	(in, gain) … 0<= in <32 -8 < gain < +8 -18db < gain < +18db
mixergain	Execute an equivalent but faster version of multiple inputgain instructions. same parameters as inputgain.	(in, gain) …
mixer	add multiple inputs	list of input separated with coma
loadxmem **loadymem**	Load the DSP accumulator X or Y with a value from a given memory location. The unique parameter is a label defined as MEMORY, eventually followed by an index number	memory label name with an optional [value] which will be added (as an index of an array)
savexmem **saveymem**	Store the value of the DSP accumulator X or Y in a given memory location. The unique parameter is a label defined as MEMORY	same as loadxmem
delayus **delaydpus**	Propagate the DSP accumulator msb through a FIFO data line in order to create a delay line.	delay in microseconds as a numerical value or

	The parameter represents the number of microseconds. delaydpus perform the same operation on 64bits original data and thus requires twice memory.	expression
delayone	Switch the current DSP accumulator with its value at the previous sample, thus creating a delay of one sample. Used to synchronize outputs across cores when savexmem or loadxmem are used.	no parameter
dcblock	Provides a high pass first order to eliminate any continuous DC signal. The unique parameter is a minimum frequency cutoff. To optimize performance, it is better to include a HP1 filter at the beginning of a biquad filter.	frequency as a value or expression 10 < f < 100
saturate	Eventually modify DSP accumulator so that the value is constrained between -1..+1 . If a saturation is detected, a gain reduction is applied by steps of -6db.	no parameter
saturategain	Combined instruction, applying a gain or reduction to the DSP accumulator and saturating the result immediately (as done with saturate instruction described above)	gain as an immediate numerical value or as an expression.
biquad	Compute multiple biquad sections based on the filter defined by a label and given as a parameter	filter label name
param	Restart a label parameter section, enabling definition of labels memory or filters below.	no parameter
shift	shift register X by n bits left if parameter n is positive, or n bit right if n parameter is negative	fixed value in number of bits -32 < n < +32
valuex, valuey	load register X or Y with a fixed value	-8.000 <= value < +8.0
clrxy	clear the 2 registers X and Y	
swapxy	exchange the 2 registers X and Y	
copyxy, copyyx	copy register X to Y , or Y to X	
addxy, addyx	Add two register : X = X+Y, or Y = X+Y	
subxy, subyx	Substract two registers : X = X - Y, or Y = Y - X	
mulxy, mulyx	Multiply two 64 bits registers X and Y and store result in X or Y	
divxy, divyx	divide two 64 bits registers X and Y : X = X / Y or Y = Y /X	
avgxy, avgyx	Average the two registers : X=X/2+Y/2 or Y=X/2+Y/2	
negx, negy	compute opposite value : X = -X, or Y = -Y	

typical cpu requirements for DSP macro instructions:

avdsp macro instructions	**cpu instruction**
initialization and core start	21 + 5 when saturate flag is raised
basic input or output	10
basic transfer between one input-output	6 + 6 per transferred in-out
load sample and apply a gain (inputgain)	14
load multiple sample and apply specific gains (mixer)	14 + 6 for each (input,gain)
biquad filtering	40 + 19 for each filter (first or second order)
verify accumulator saturation and eventually apply -6db reduction steps	12 + 2 in case of reduction
same with a gain applied before (saturategain)	21 + 2 in case of reduction
fifo pipeline to delay a signal (delayus or delaydpus)	20 for delayus 23 for delaydpus
share data across core through memory (savexmem or loadxmem)	10
fifo pipeline for delaying by one single sample to synchronize core access to memory (delayone)	13
typical basic program for 1 channel: input x biquad 4 sections (eg LR4 + 2 x PEAK) delayus y output z	**138** or **214** for **8** sections

List of supported filter names:

type	**label name** (LP = low pass, HP = highpass)	**parameters**
Bessel	LPBE2, LPBE3, LPBE4, LPBE6, LPBE8 HPBE2, HPBE3, HPBE4, HPBE6, HPBE8	Frequency and optional gain
Bessel computed for fc cutoff at -3db	LPBE3db2, LPBE3db3, LPBE3db4, LPBE3db6, LPBE3db8, HPBE3db2 HPBE3db3, HPBE3db4, HPBE3db6, HPBE3db8	
Butterworth	LPBU2, LPBU3, LPBU4, LPBU6, LPBU8, HPBU2, HPBU3, HPBU4, HPBU6, HPBU8	

Linkwitz-Riley	LPLR2, LPLR3, LPLR4, LPLR6, LPLR8 HPLR2, HPLR3, HPLR4, HPLR6, HPLR8,	
classic first order	LP1, HP1, AP1 (allpass)	Frequency and gain
High and Low shelf first order	LS1, HS1	Frequency and gain
High and Low shelf second order	LS2, HS2	Frequency and Q and gain
classic 2nd order (can be cascaded to create any filter)	LP2, HP2, AP2, (all pass)	Frequency and Q and optional gain
special 2nd order	PEAK, NOTCH, BPQ, BP0DB (bandpass)	Frequency and Q and optional gain
Linkwitz Transform	LT	F0,Q0,Fp,Qp and optional gain

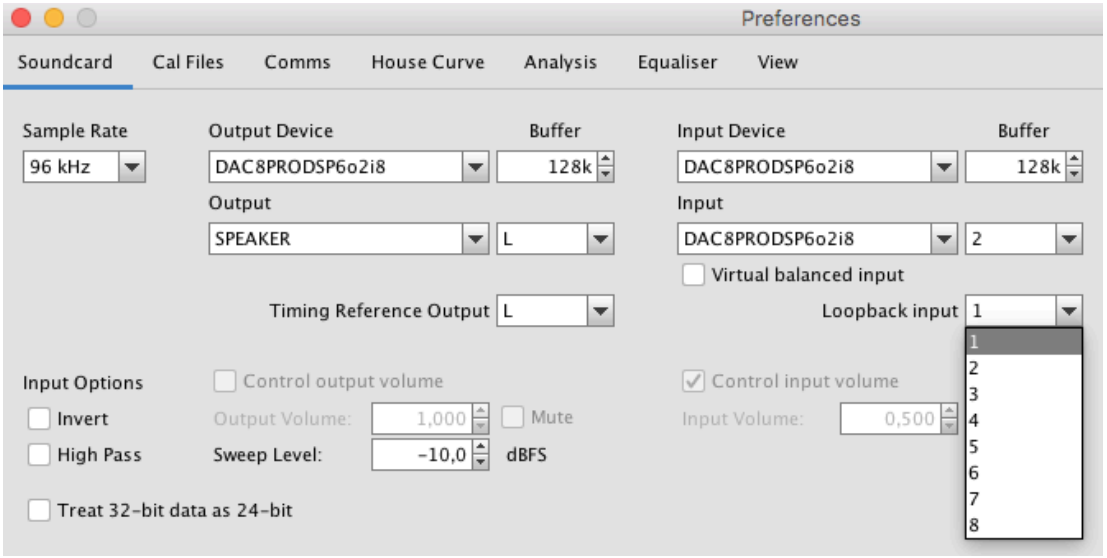
First dsp program creation and implementation.
Clone the github directory or download and unzip the corresponding package.
In the different folder you will find command line utilities for your operating system.

You can create a dedicated DSP working folder in which you copy both utilities and the examples.
Edit the example1.txt file with your preferred text editor.
with a terminal session or command line window, convert the file as a binary file:
dspcreate -dspformat 28 -dsptext example1.txt -binfile ex1.bin
upload the resulting binary file with
xmosusb –dspload ex1.bin
verify status and cpu load with
xmosusb –dacstatus
write binary file loaded into flash memory location 1 (up to 4 locations)
xmosusb -dspwrite 1
go in DAC8 **Filter** menu and select DSP prog 1

A square 1 is now displayed in the top left corner of the volume screen of the DAC8.
Play music and verify that the sound is now Mono instead of stereo.
Or use REW to play a signal either on the Left(1) or Right(2) channel and check the resulting signal either on channel 1 or 2. Channel 1 is a direct copy of Left and shall be used as the reference.
Channel 2 is the mix of Left and Right inputs in this example.

Using REW to test a basic program with loopback:
It is important to verify the result of a DSP program with a tool like REW, displaying the response with a frequency sweep, the time response with an impulse or square wave, and to measure group delay of the solution. REW can provide signal on each of the 8 USB-out channels and display FFT or scope view for any of the 8 USB-in channels. It is good practice to provide a time reference signal, with a loopback of USB-Out and USB-in for channel 1 (also called Left).

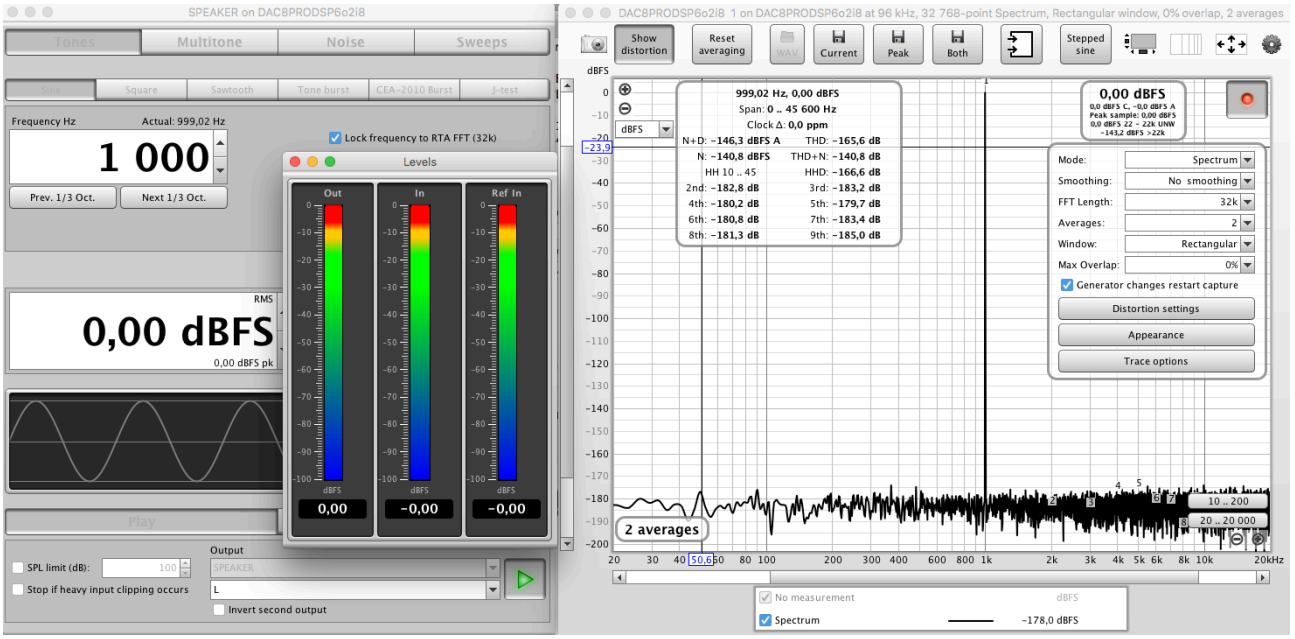
Example of Preference screen :



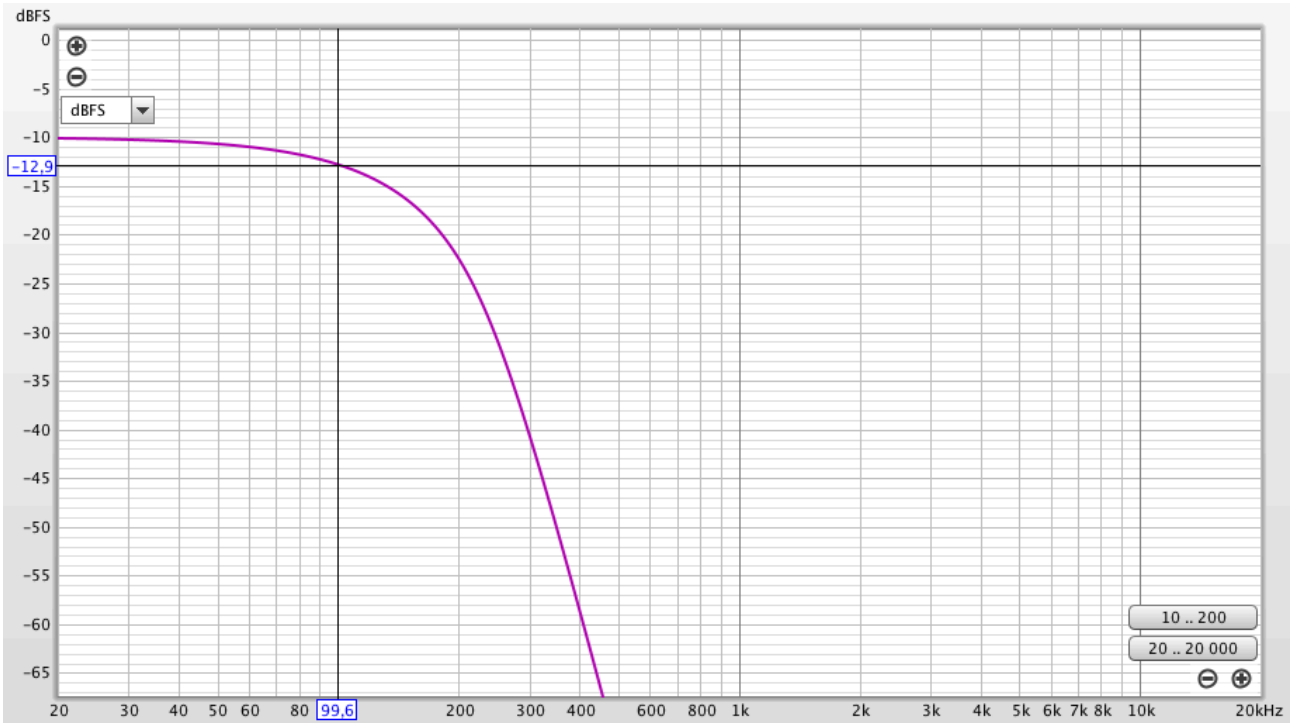
Here is a simple test program with a `transfer` function at the beginning of the program to provide loopback between USB-out 0 and USB-in 0 (16,24). The original signal is also transferred on DAC0 (16,0), a lowpass filter is provided on DAC 1 and USB-in 1, a high pass filter is provided on DAC2 and USB-in 2:

```
lowpass      HPBE8(200)   #bessel filter 8th order (constant group delay)
highpass     HPLR4(400)   #linkwitz-rilley 4th order
core
      transfer (16, 24)(16,0)  #this provides fast loop back on USB channel 0
      inputgain (16, 0db)      #load sample from USB out 0
      biquad lowpass           #apply low pass bessel
      output 1 ; output 25     #output result on DAC1 and to USB for test
      inputgain (16, 0db)      #load sample from USB-out 0 (once again)
      biquad highpass          #apply highpass
      output 2 ; output 26     #output result on DAC2 and to USB for test
end
```

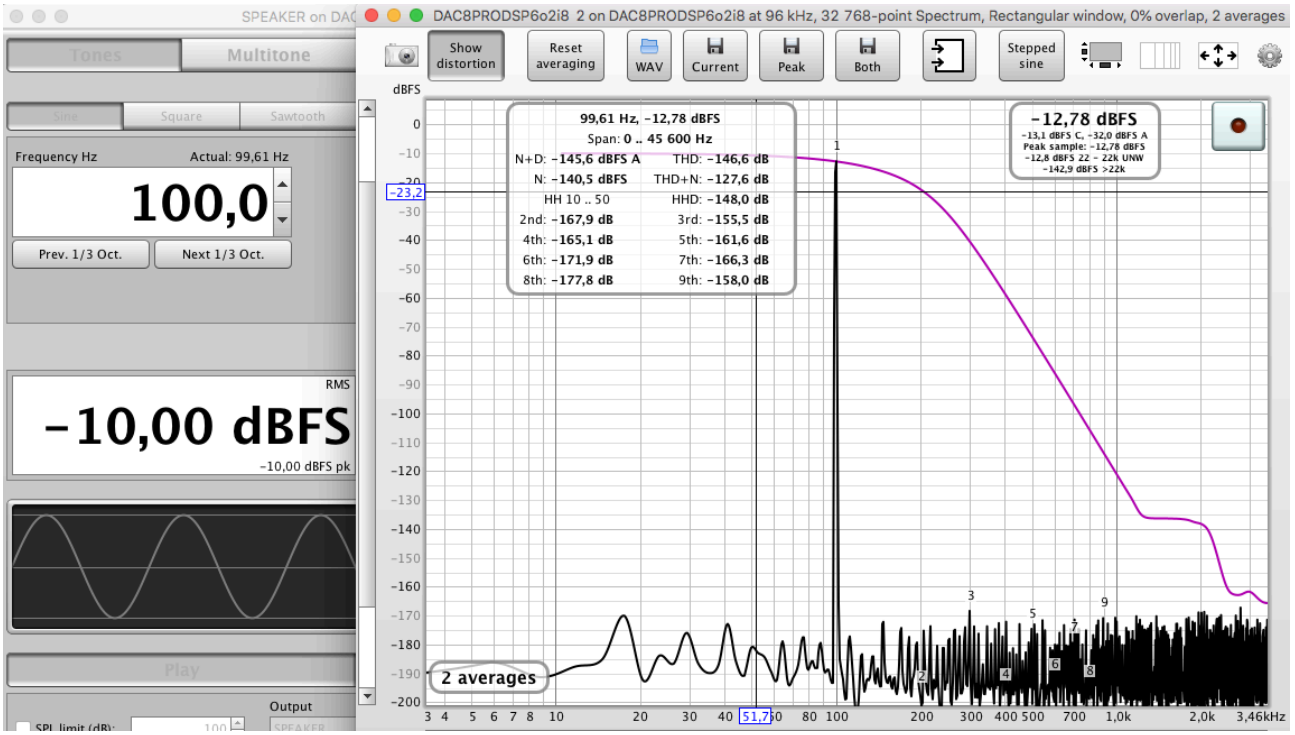
Resulting screen when testing loop back with a 0db sine wave as input: no clipping , no THD



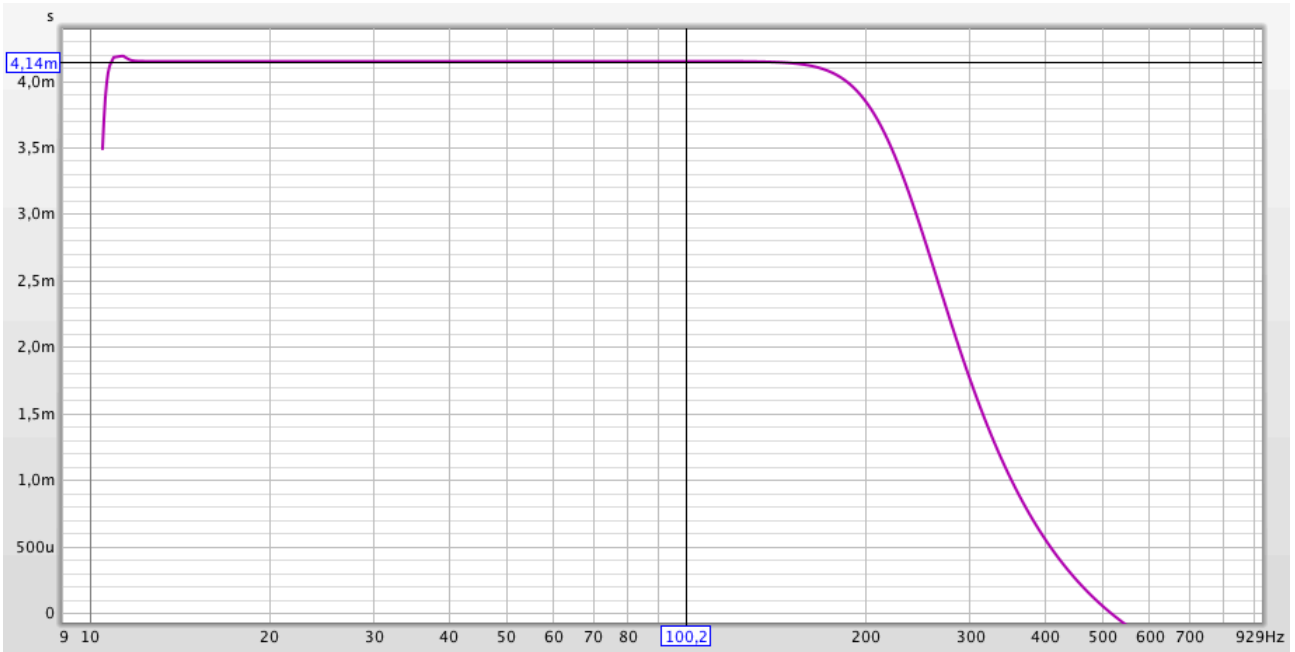
Sweep Measurement on channel 1 with -10db signal in range 10-20000:
LP Bessel 8th response is -2.9@ 100hz, -31@300hz, -77@600hz so 46db/octave (300..600)



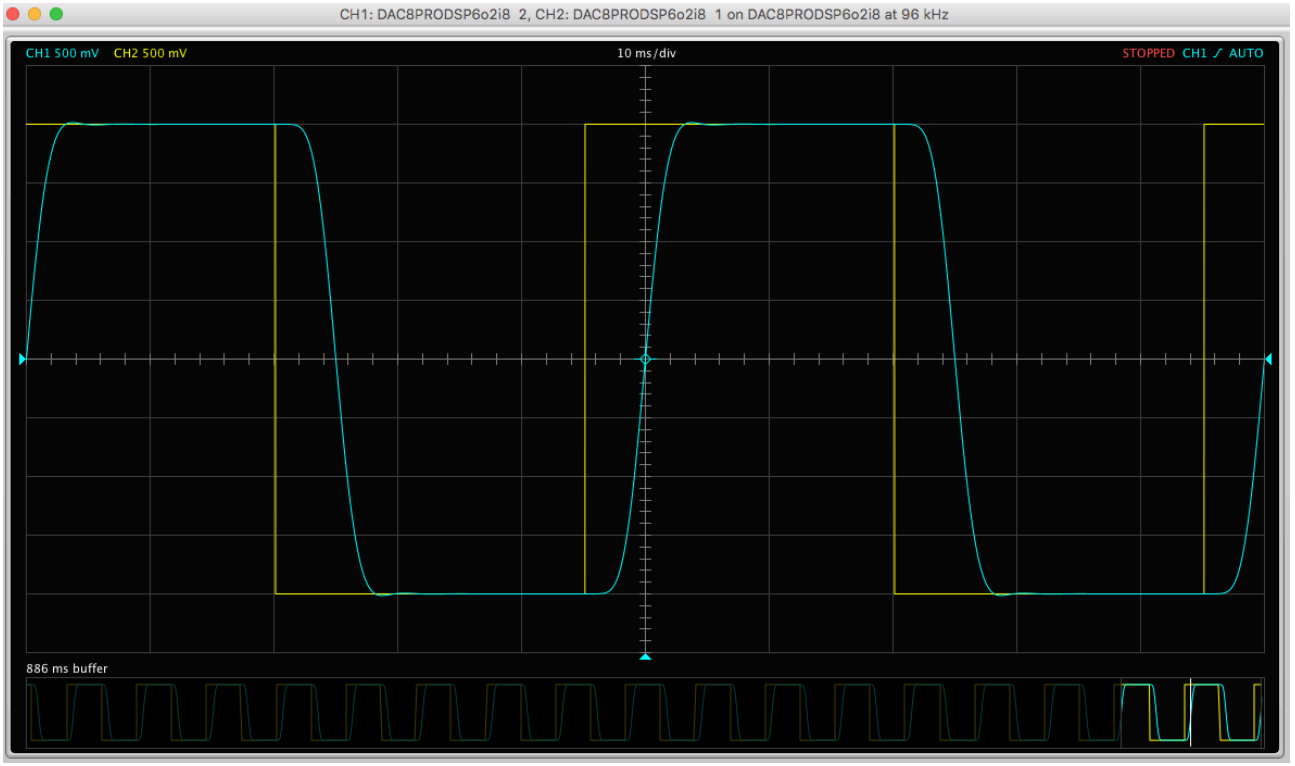
There is no THD for a 100hz test tone (thanks to our specific biquad routine):



As expected, the group delay of the bessel filter is flat below the cutoff frequency, at 4.14ms.



The scope view show a perfect damped step response to a square wave (0db 20hz here) with no clipping:



For the channel 2, response of the high pass LR4:

TBD / WIP

Operating system specificities.

Mac osx.
The two utilities must be launched in a terminal window with the preceding characters ./
They are compiled for x86_64 but are also compatible with Apple silicon M1 or higher via the transparent Rosetta emulator. Eventually the user might need to change the access rights of the compiled files with the command:
chmod +x xmosusb dspcreate

At the time of writing, the utilities have been tested and successfully run on iMac intel core i5 with High Sierra and on Macbook pro M1pro with Sonoma.
Mac OSX usb host driver is compatible with DAC8PRO in 8 channels mode. Configuring the USB with the Midi utility in another mode like 2 or 6 channels will not change the 32 memory mapping explained in this document.
Xmosusb utility can be used to upload DSP programs while a player is using the same USB device playing sound or music to the DAC (like with REW software). From experience this doesn't impact the USB audio streaming and does not impact the REW java audio driver. It is then possible to develop a DSP program seamlessly without constraints and to do repetitive live tests by loading DSP programs in memory “on the fly”.

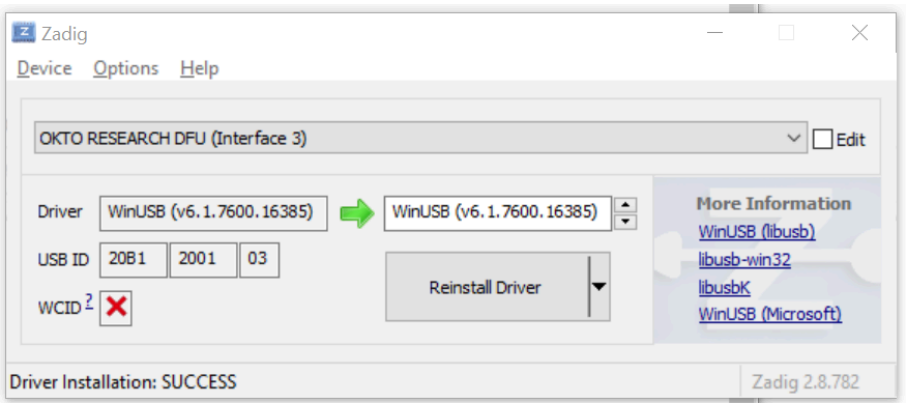
Linux.
The utilities are compiled and tested for Linux Mint cinnamon x86_64.
xmosusb utility requires sudo prefix
Libusb library must be installed separately and visible in the “Path”.
example : sudo apt-get install libusb-1.0-0-dev
A version is provided for raspberry pi os compiled for aarch64.
Finally a version is also provided for raspberry pi ARM 32 bits processor and this was tested with a Volumio 3 image.
As for mac osx, no special limitations are known for using the xmosusb tool with the USB audio driver simultaneously.

remark: for each operating system, the user could create batch files with parameters to automate the execution of the 3 or 4 usual commands, this is not proposed by default.

the dspcreate utility requires libavdspencoder.so/dylib library and might need to update your library path by editing and exporting LD_LIBRARY_PATH, and sometimes just adding a “.” to refer the same folder.

Windows.
The utilities have been compiled with Mingw64 and can be run directly in a windows cmd shell.
Unfortunately, windows driver mechanisms and xmosusb utility using libusb brings some concerns which we have been able to tackle but with some user constraints.
Nevertheless the process explained below can provide a reasonably seamless experience for developing and uploading a DSP program on Windows, but the author recognizes the limitation and recommends using linux or Mac OSX during development and tests phases.

First of all xmosusb.exe utilizes the open-source libusb library and requires Windows winusb driver to be installed on the interface 3 (DFU) of the DAC8PRODSP USB composite device. This has to be installed with Zadig. (using version 2.8 in this example below).



Then the xmosusb.exe will discover the DAC8PRODSP properly on the usb bus:

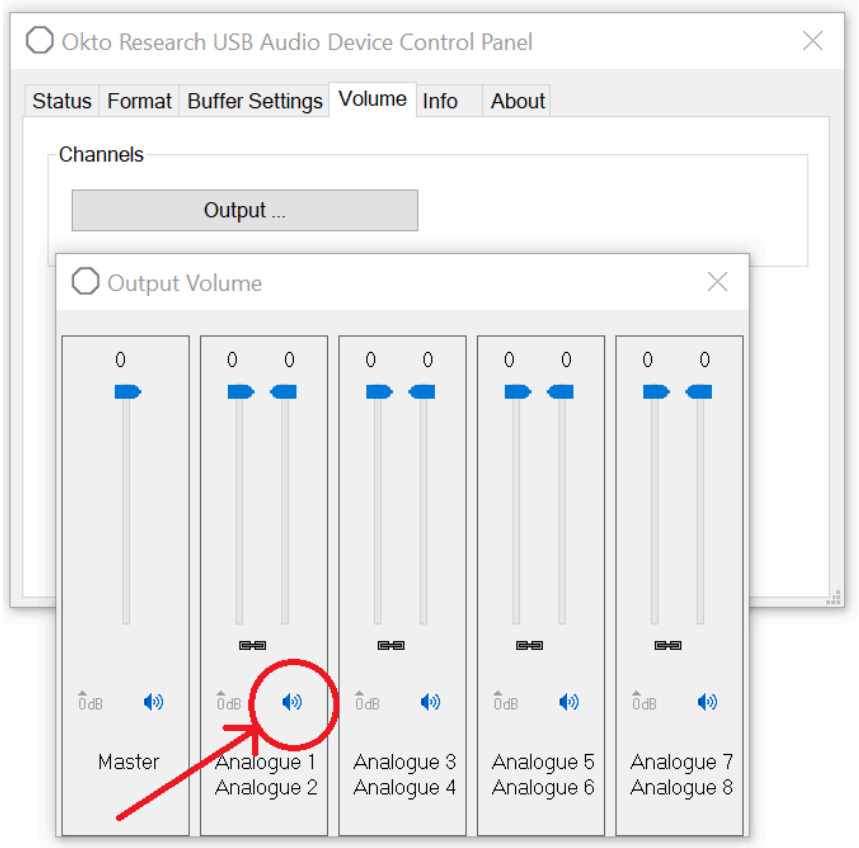
```
C:\XMOS\xmosusb\windows>xmosusb

This utility is using libusb v1.0.26.11724


[0] > VID 20B1, PID 2001, BCD 0160 : OKTO RESEARCH  DAC8PRODSP4  000016
      (0)  usb Audio Control
      (1)  usb Audio Streaming
      (2)  usb Audio Streaming
      (3)  usb DFU
```

But this process is conflicting with the OktoResearch USB driver which captures all the usb interfaces. So the OktoResearch usb driver MUST be uninstalled before installing winusb.
By default the Windows usb audio driver will be installed and will recognize DAC8PRODSP without any issue, but it supports only 2 channels output (playing) and input (recording). This is practically enough to test the DSP program, channel by channel with REW in wasapi mode.

Once the DSP programs are developed, tested, stabilized and written to the DAC8PRODSP flash memory, then it is possible to reinstall the OktoResearch usb audio driver to benefit from the 8 channels handling and to use asio capabilities.

Alternative:
Because the author realizes that it is painful to uninstall and then reinstall the OktoResearch usb driver during the DSP development phase, a specific mode has been implemented so that the DAC8PRODSP can be seen on the USB bus either with its original VID:PID (152A:88C4) or with a temporary and alternate VID:PID (20B1:2001). This gives the possibility to install winusb on this last one and to use xmosusb.exe freely, while the original VID:PID stays attached to the OktoResearch usb audio driver.

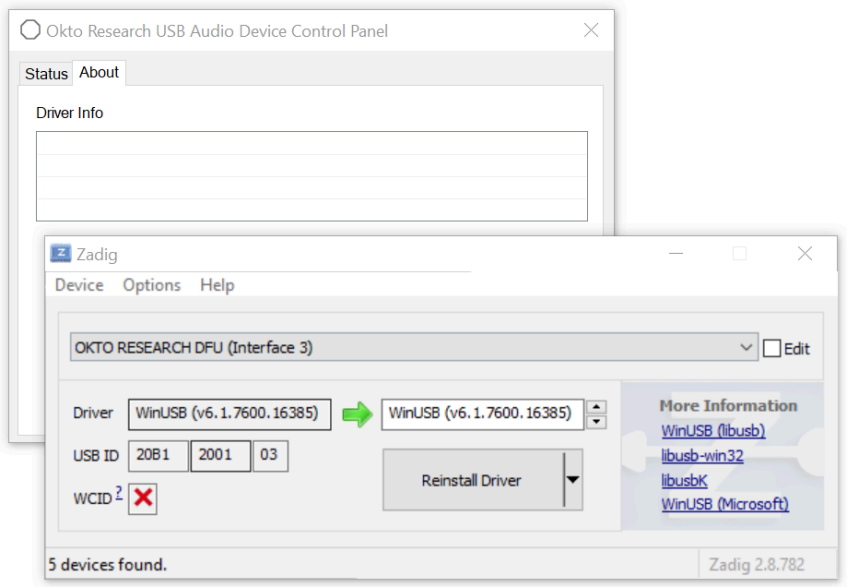
To reboot the DAC8PRO with this temporary VID:PID, go to the OktoResearch driver control panel, select the Volume tab and click the “output” area to display the volume's vertical cursors.



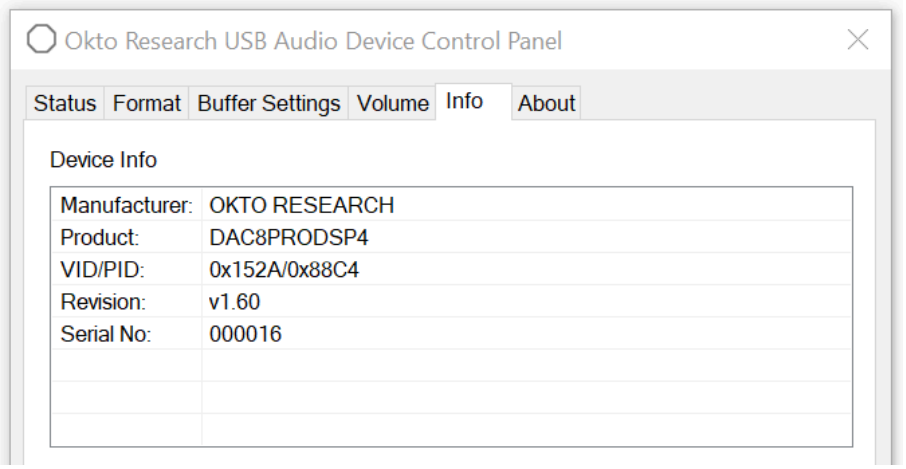
Click on the muting icon for the 2 first channels to mute them, and this will start a 5 seconds period where the USB cable can be disconnected : when reconnecting the usb cable, the dac will re-enumerate with the temporary VID:PID, so it will not be seen by OktoResearch usb audio driver, and windows will install (the first time) its basic audio driver.

remark : If the cable wasn't removed within the 5 seconds period then the DAC8PRODSP keeps running with its original VID:PID and this special sequence is canceled. It is possible to restart this process at any time. The DAC8PRODSP recognizes the sequence going from unmute to mute on channels 1 & 2 only, the states of the other channels doesn't matter.

Then it is possible to install winusb with zadig on the temporarily detected device:



To restore the original VID:PID, just disconnect and reconnect the usb cable. The DAC8PRODSP will immediately reappear in the thesycon driver control panel info box with its original VID:PID.



remark : After re-entering in the normal mode of operation with the original VID:PID, remember to unmute the 2 first channels in the volume/output menu to effectively listen to music !

xmosusb utility

The xmosusb utility is an extension of the original DFU utility provided by XMOS company. It has been extended to support additional commands. The list below is not exhaustive and provides only the commands which may be used at some point in time with the DAC8PRODSP firmware. Each command starts with two successive "-" (minus) characters (for history reasons…):

–dspload file : Sends a binary file to xu216 firmware. The binary codes are loaded in xu216 ram memory and eventually overload the latest DSP program loaded from the front panel Filter menu.

–dspwrite num : Writes the xu216 ram memory to the flash location "num". This is the way to save a DSP program into a permanent flash location within DAC8PRODSP.

–dspread num : Loads a DSP program from the flash location "num" into xu216 ram memory. This program then becomes the one used by xu216, even if the front panel displays another number.

–flashread page : Provides a dump of 64 bytes of internal flash memory at a given page position. A page is 64 bytes. DSP programs are written as of page 64 and onwards.

–flasherase sector : Erases a 4096 bytes of flash memory at a given sector. A sector size is 4096 bytes. DSP programs are stored in flash starting at sector 1. To erase all DSP programs in flash, just type xmosusb –flasherase 1. Remark: original factory firmware is stored in a specific boot partition and is protected from erasure.

–dspheader : Provides a summary of the DSP program currently loaded in xu216 ram memory.

–dacstatus : Provides some internal information and instructions used on each xu216 core.

–xmosload file : This command can be used to flash an xu216 firmware compatible with DAC8PRO, with the same result as using the DFU Windows tool, especially useful for linux or Mac OSX or Raspberry pi platforms. If the provided firmware contains a front panel firmware then it migth be flashed during the next boot process. This command cannot and will not verify if the provided binary file is compatible with the target xu216 digital board and therefore there is a risk to brick the unit. This command is perfectly safe to use as long as the file provided is compatible with DAC8PRO.

–samdload file : Sends a binary file to the xu216 cpu to flash the front panel firmware. No any checks are done on the binary file and this command should be utilized only when instructed by OktoResearch to flash a specific version of the front panel firmware without changing the current version of the xmos firmware. The flash process takes about 50 seconds without any indications.

–samdreflash : Starts a flash process from the internal xu216 firmware to the front panel device, using the version originally embedded inside the xu216 firmware. This can be useful to restore an original front panel version when restoring a DAC8PRO to its factory version, or if the front panel was overwritten with a specific custom or OEM version using a previous –samdload command.

dspcreate utility
This utility is part of the AVDSP project, it is used to generate DSP binary files for devices using AVDSP runtime, like DAC8PRODSP. Source code are available on github fabriceo/avdsp.

The following options can be used (each option starts with a single "-" minus character):

-dspformat x : forces the DSP encoder to generate either integer or float values for any constant (gains, filter coefficient). value 3 will force generating 32 bits floats so precision is limited to 24 bits. Value 2 will force generating 32 bits integer with a mantissa of 28 bits and an integer part limited to 3 useful bits (-7..+7). Any value between 16 and 30 can be used to generate a 32 bits integer with a customized mantissa between 16 and 30 bits.
At the time of writing, the DAC8PRODSP runtime will convert all the constant values (float or integer) into integer q28 or s3.28 format so it is recommended to just use -dspformat 2

-dsptext filename<+filename2…> : Process one or many text files that can be chained with the "+" character (but no space character allowed).
The embedded analyzer recognizes a simplified syntax for label and filter definition and a limited list of AVDSP macros. It generates a list of "opcode" corresponding to a DSP program. The translation into opcodes is printed on the console for information and debugging purposes. In case of syntax error, the line is printed on the console, with an "^" arrow below the token generating the error, and code generation is aborted. At the end of the analysis, a summary is provided with the number of cores used, code size and checksum.
remark: *when chaining multiple source files with the "+" character, it is useful to use the first file as a definition for constants or to label each input output that will be used by the next chained text file(s). For example the file "dac8pro0" or "dac8pro1" in the example folder can be used to define the aes, dac or usb memory location with a convenient name instead of using directly the memory location0..31. The dac8pro0 will consider the channels from 0 to 7, but the dac8pro1 will name them from 1 to 8, which is more aligned with audio software (like REW for example).*

-dspprog filename : open a dynamic library file named filename.so or filename.dylib or filename.dll and execute the code corresponding to a procedure named dspProg(). Such a procedure shall generate the DSP opcode by calling the primitive of the AVDSP library. This more complex and flexible approach is compatible with DAC8PRODSP but not documented here.

-binfile filename : instructs to save the list of opcodes as a binary "filename".

-hexfile filename : instructs to save the list of opcodes as a C formatted text file which could be included in a specific xmos audio project and immediately embedded inside an xmos firmware. This is typically used to onboard a default DSP program inside a custom firmware.

-dumpfile filename : export the DSP program symbols and their address in a text "filename".

-fsmin min -fsmax max : used to force a minimum and/or maximum sampling rate limit. All the filter coefficients and size of the delay pipelines will be pre-computed for each of the sampling rates included within these two boundaries. by default fsmin is set to 44100hz and fsmax is set to 192000hz. For DAC8PRODSP firmware which are voluntarily limited to 96000hz, it is relevant (but not mandatory) to use -fsmax 96000 to limit the size of the generated code to only relevant values.

when using -dsptext selector, then all remaining information at the end of the command line for dspcreate will be used as additional lines of the DSP program, and they will be passed one by one

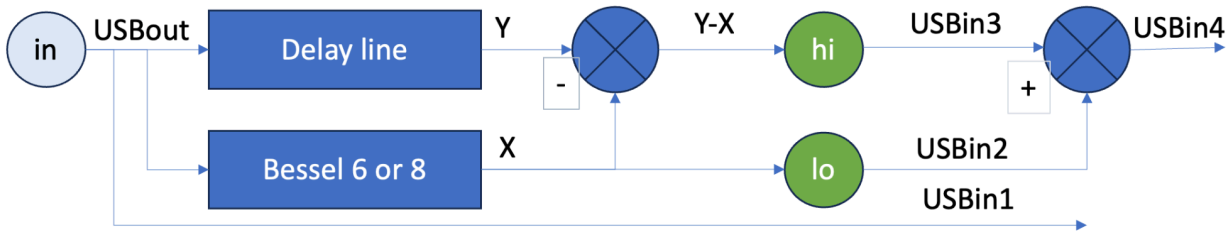
to the analyser and before starting analyzing the main text file. This is useful to pass dynamic parameters like frequency cutoffs, or specific gains on the command line, without modifying a common core text file. Here are some typical command line for DAC8PRODSP:

dspcreate -dspformat 2 -fsmax 96000 -dsptext example.txt -binfile prog1.bin FC=400 GAIN=-3db
dspcreate -dspformat 2 -fsmax 96000 -dsptext example.txt -binfile prog2.bin FC=500 GAIN=-2db
dspcreate -dspformat 2 -dsptext dac8pro1+example.txt -binfile prog3.bin coef=0.6

Extra DSP instruction, example of subtractive filters

The AVDSP runtime includes specific instructions to do some basic math on the 2 DSP accumulators X and Y. This gives the possibility to create special treatments, either experimental or for practical situations.

Subtractive filters are very interesting examples to get crossover with linear phase or constant group delay. Principle:



The Bessel 8 filter provides a flat group delay, and the subtraction after the delay line will give a second order high pass filter. The sum of the 2 resulting signals shows a flat response and a flat group delay. It is possible to adjust the delay line in code below to see the impact on highpass.

```
fc              600
bessel8         LPBE8(fc)
gd      =       986000/fc       #group delay for bessel8 at fc
core
        input 16        #get sample from usb channel 1
        copyxy          #copy X into Y
        output 24       #feedback to usb for reference
        biquad bessel8          #compute lowpass
        output 25       #output to usb channel 2
        swapxy          #getback Y into X
        delaydpus gd    #dual precision delay line
        subxy           #compute high pass
        output 26       #output to usb channel 3
        addxy           #sum signals to check resulting wave
        output 27       #output to usb channel 4
end
```

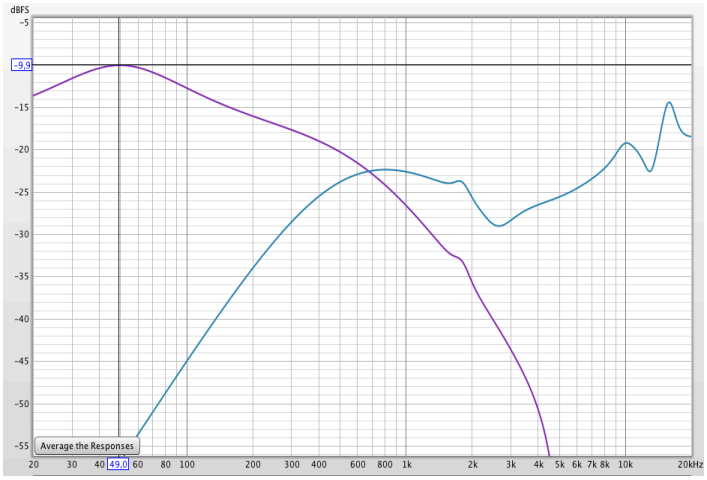
example decay convolution (...0.125, 0.25, 0.5)

```
sum     MEMORY 1
coef    0.6
core
        input 16
        output 24
        gain coef
        copyxy
        loadxmem sum
        gain 1-coef
        addxy
        savexmem sum
        output 25
end
```

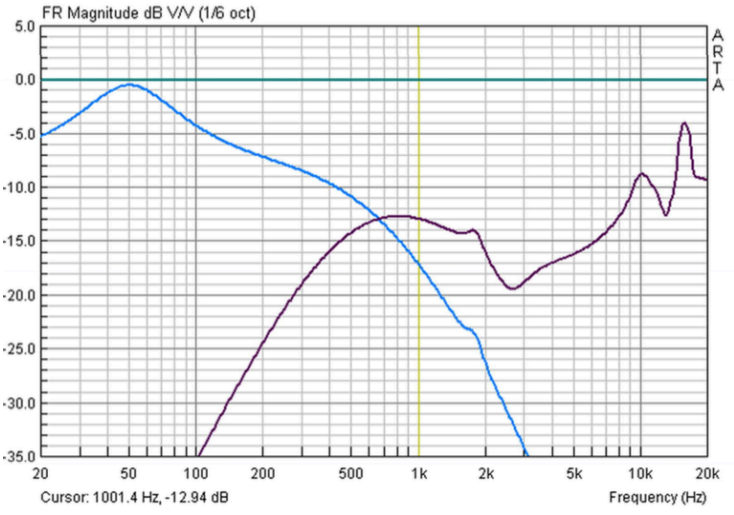
example LXMini crossover 2 ways

```
ineq    PEAK(1800,7,2db)
loeq    LP2(700,0.5) PEAK(50,0.5,7db) PEAK(5100,8,-16db) PEAK(5800,3,-10db)
hieq    HP2(700,0.5) LS2(1000,0.5,16db) PEAK(2600,2,-4db) HS2(8000,0.7,8.5db) PEAK(10000,5,2.3db)
PEAK(13100,7,-5db) PEAK(15800,10,7db)
core
        transfer (16,24)        #feedback to host for reference
        inputgain (16,-7db)     #7db less due to PEAK at 50hz
        biquad ineq
        biquad loeq
        output 25               #output to usb host for tests
core
        inputgain (16,-17db)    #-17db to equilibrate with woofer
        biquad ineq
        biquad hieq
        output 26               #output to usb host for tests
end
```

measured results in REW:



Original on Linkwitz web site: https://www.linkwitzlab.com/The_Magic/The_Magic.htm



Saturate instruction details and example

The following program can be tested with the third party tools REW with “Scope” view, by switching the input channels 1 to 4 and by generating a square wave at 0dbfs. It demonstrates the benefit of using the saturate instruction as a way to avoid clipping in all cases.

```
highpass   HPBU2(400)
core
      input 16
      output 24         #loopback for reference
      biquad highpass #the result in the DSP accumulator exceed +1..-1
      copyxy            #save result in accumulator Y
      output 25         # signal is clamped sharp between +1 and -1
      copyyx            #retreive accumulator Y
      gain -3db         #accumulator is reduced between +0.8..-0.8
      output 26         #clean as biquad routine provides 18db headroom
      copyyx            #retreive accumulator Y
      saturate          # the result is reduced by 6db automatically
      output 27
end
```

When a `saturate` instruction detects an overflow situation (inside any core) it raises a global saturation flag and clamps the signal between +1..-1. Then at the next sample cycle, the first core checks this flag and eventually increments a saturation counter register.
If a `saturate` instruction sees a number N in this counter register, it reduces the value of the accumulator by N x -6db (shifting N bit right in fact). So the hard clipping happened only on one single sample and this is not audible and has no any consequence on the audio chain.
The only way to reset this saturation number register is to switch on/off the DAC, or reload a DSP program, or switch the sampling rate frequency, or cycling a mute/unmute.

It is recommended to use the `saturate` instruction until all demonstrations are done so that no clipping is produced by the user DSP program. Extensive tests can be done with REW square waves at 0dbfs and reviewing each output with the “Scope” view.

Using `saturate` instruction before any `output` requires additional cpu time but provides a safe solution to avoid any harmonics on outputs linked to potential overflow. This approach of reducing dynamically the output gain by steps of 6db is a unique way to avoid traditional signal compressors, which inherently bring a lot of distortion when the signal comes close to the peak threshold. Once all the gain chains have been adjusted to avoid clipping, the `saturate` instructions can be removed safely.

By extension, the `saturategain` and `saturatetpdf` and `saturategaintpdf` instructions have been introduced in the langage to optimize treatment.
`saturategain` executes a gain function and then the explained `saturate` behavior.
`saturatetpdf` applies a tpdf pattern (as explained for `outputtpdf`) and then `saturate`.
`saturategaintpdf` applies a gain, a tpdf pattern and then `saturate`.

Newly introduced tpdf instructions

The `tpdf` instruction provides the possibility to apply N bits dithering with triangular noise.
To use this functionality, the `tpdf` instruction must be defined preferably at the beginning of the first core, with a numerical parameter defining the bit position (8 to 30). This instruction will then compute a signed 31 bits noise value and will apply a mask to keep only (32-N) useful bits. This is done at each sample rate cycle and the noise value can be used in any core.

Then, anywhere in the program and in any core it is possible to use a specific `outputtpdf` instruction which will add this random number to the sample, before storing the value in the target memory location.

It is possible to use the `tpdf` instruction in any core, in addition to the first core : then the number of dithering bits will be specific within this core. for example:

```
core
      tpdf 20
      input 16
      output 24             #original value
      outputtpdf 25         #20 bits dithered value
core
      input 16
      outputtpdf 26         #20 bits dithered (taken from first core)
      tpdf 18
      input 16
      outputtpdf 27         #16 bits dithered value
core
      input 16
      outputtpdf 28         #20 bits dithered (taken from first core)
end
```

In fact, `outputtpdf` instructions always use the number of dithering bits defined by a previous `tpdf` instruction in its own core, otherwise defined in the first core. If no `tpdf` instruction is defined in the program, the encoder will add one automatically when the first `outputtpdf` instruction is encountered in the program flow. Then the default dithering is arbitrarily chosen to 24 bits.

Using conditional `core` and `section`

The AVDSP catalog of macros does not include any conditional testing but there is a possibility to dynamically activate or not a core depending on external conditions provided by the DAC firmware in a global status flag and reflecting the DAC modes and status.

By default any `core` declaration is valid and this will use a physical cpu core in the XU216 processor. But it is possible to set a condition with one or two parameters after the keyword `core`. These optional parameters are one or two binary masks, which can be provided as a decimal value or hexadecimal by using the “x” prefix or binary by using the “b” prefix.
The first parameter defines a mask for a bitwise “and” operation with the DAC status flag, and any matching bit set to 1 will enable the core.
The second optional parameter also defines a mask for a bitwise “and” operation with the DAC status flag, but the result of this “and” must be zero to validate the core, otherwise no physical core is allocated during runtime and then the total number of mips is redistributed.

The status flag is reflecting various status and mode as described below:

bit 15..12	dspcores	bit 11..8	AES inputs	bit 7..4	rate	bit 3..0	mode
0001	4	0001	only 1	0001	44/48k	0001	PureUSB
0011	5	0010	2 inputs	0010	88/96k	0010	PureUSB2
0111	6	0100	3 inputs	0100	176/192k	0100	AES/USB
1111	7	1000	4 inputs	1000	352/384k	1000	PureAES

b31..28	chanadc	b27..24	chandac	b23..20	toUsb	b19..16	fromUsb
0..15	ADC/AES	0..15	DAC/DSD	0..15	chan/2	0..15	chan/2

As an example, to activate core when the DAC is configured with a sampling rate below 176khz and only when the mode PureAES is selected, the core statement must be written as
`core 0x30 7` or
`core 0b0011000 0b0111`
then all the code below this core statement will be executed or not.

By extension, the instruction `section` gives the possibility to enable a code section within any active core if the conditions are met. example:

```
core
      section 0b1000   #check if PureAES mode is selected
      input 8          #if yes, read first AES channel
      section 3        #check if any USB mode is selected
      input 16         #if yes, read first USB channel from host
      section          #do nothing but mark the end of the section area
      output 0         #code continues here in all case and update DAC
      section 1        #check if PureUSB mode is selected
      output 24        #if yes provide the host with a loopback
end                    #end also marks the end of the previous section
```

Optimized fixed point 64 bits math explained for tech savvy customers.
The XMOS XU216 is a powerful micro processor with a 32 bits instruction set and some specific instructions able to handle 64 bits. This is used extensively in the AVDSP runtime (using dual-issue assembler programing) in order to provide a PureDSP treatment with almost no THD.

The audio samples provided from the USB host are either 16, 24 or 32 bits signed integers. The sample provided by the AES inputs are 16 or 24 bits only. They are all represented as 32 bits signed data in memory with format called q31 or s0.31:

original 32 bits audio samples in q31 or s0.31 format	
b31	**b30..0**
sign	31 usefull bits (192db)

The AVDSP runtime is using two 64bits Accumulators called X and Y in this document.
With the standard input instruction the audio samples are loaded in accumulator X with an 8 bits headroom so the 64bits presentation is q56 or s7.56:

audio samples converted to 64 bits accumulator		
b63...b56	**b55...b25**	**b24..b0**
extended sign (00 or FF)	31 usefull bits	25 zeros

A gain value is coded as a 32 bits signed integer, with 1 bit for the sign, 3 bits for the integer part, 28bits for the mantissa, also called q28 or s3.28. This gives the possibility to represent values between +7.999 and -8.0:

32 bits coded values or gain or biquad coefficients as q28 or s3.28		
b31	**b30..28**	**b27..0**
sign	3bits = +18db	mantissa (28 usefull bits 168db)

The inputgain instruction is multiplying a sample s0.31 by a gain s3.28 with a 32x32=64bits multiply instruction which provides a s4.59 base result. The accumulator is then shifted right by 3 to come to s7.56 format, which will be the standard used during all next treatments.

64 bits accumulator as q56 or s7.56 (+/-127.999 = +42db)		
b63	**b62...b56**	**b55..0**
sign (0/1)	integer (7bits)	56 bits accuracy

An output instructions use specifics cpu instructions to quickly saturate and reduce the s7.56 accumulator to a 32bits s0.31 value which is stored in the output location.

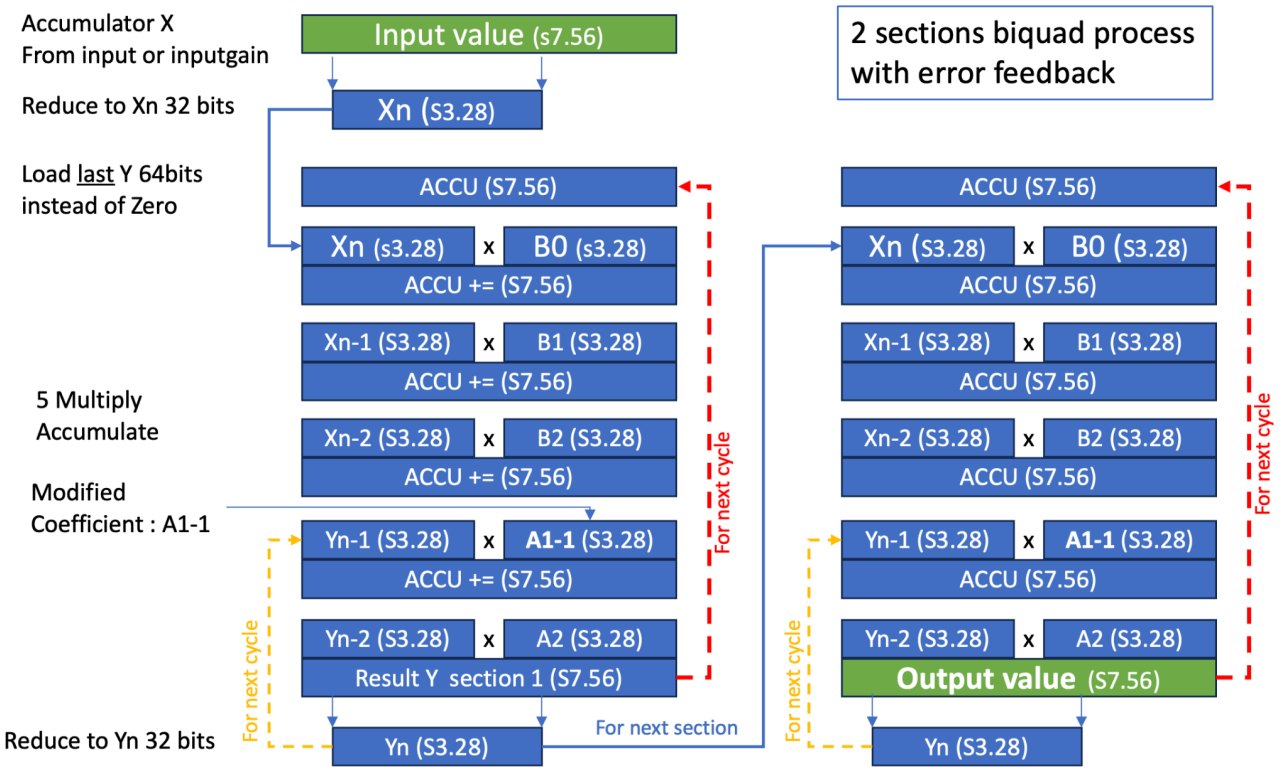
A gain instruction multiplies the 64bits X coded s7.56 by the 32bits gain coded s3.28 with three multiply instructions in a row resulting in a 96 bits value coded s11.84. This is then shifted right by 28bits to reduce the accumulator to the regular s7.56 format.

An outputgain instruction also combines three multiply instructions and then reduce the 96bits result to get 32 useful bits before storing them in the output location.

The mulxy instruction multiplies the two 64bits accumulator X and Y with 5 multiply instructions and keeps only the usefull 64bits so that a s7.56 signal multiplied by a s7.56 signal is reduced as a s7.56 value in the destination accumulator X (or Y with mulyx).

The biquad instruction requires a 32 bits Xn sample and produces a 64bits Yn result by applying the five coefficients multiplication (form II): Yn= Xn.b0+Xn-1.b1+Xn-2.b2+Yn-1.a1+Yn-2.a2

Usually a filter is a cascade of multiple biquad sections, and each Yn output is reduced to a 32bits sample before being fed as the input of the next section.
The approach is to reduce the precision of the accumulator X coded s7.56 to a 32bits value Xn coded s3.28 before applying the five multiplies with the coefficient (b0,b1,b2,a1,a2) coded s3.28. The resulting Yn value is coded 64bits s7.56 and will be reused asis in the next cycle, instead of clearing the accumulator as usually done. To compensate for this, the a1 coefficient is reduced by 1.0 during the encoding phase. This is an optimized approach to “truncation error feedback” as explained by Jon Dattorro in this document : https://ccrma.stanford.edu/~dattorro/HiFi.pdf



The benefit is to get a much larger precision at low frequency or low signal, without requiring 64x32 bits multiply instructions as usually done. The resulting THD for a low pass LR4 filter for high fs (192k) is better than -120db in the passband.
This approach is better than using 32 bits IEEE float (24 bits mantissa+7 bits exponents).

The 28 bit base mantissa used for biquad inputs or coefficient or gain value, generating a 56 bit mantissa as described for instructions above, can be changed between 16 and 30 simply by using the option selector -dspformat x in the dspcreate command line. The user can test the benefit of optimizing this by using third party software like REW with the RTA view, with the requirement of using 32 bits audio drivers (like asio or alsa but not java) to better evaluate the benefit of this tuning. Mantissa of 28bits is a good tradeoff to get maximum performance while providing a 42db headroom during basic gain computation and 18db for biquads (remark, all-pass filters require larger coefficient and headroom is then limited to +12db).

All the optimisation made and the choice of 64 bits integer accumulator gives an excellent precision. This results in almost no measurable distortion on audio signals in the digital domain.

Known limitation or issues or bug and support

Analyzer, opcode generator:
The -dsptext selector of the dspcreate utility is relatively new in the AVDSP project and has been developed to make things easier to design an equalization and crossover solution.
A set of macro has been defined to cover most of the directives from the original AVDSP library and the overall syntax is chosen to be as simple as possible. In addition, a concept of labels, filters and memory is defined and it is possible to type mathematical expressions (without parenthesis).

This evolution is voluntarily not a complex development as the interpreter fits in about 1000 lines. The users are welcome to report issues, bugs or request for enhancement using the “issue” of the original github repo : https://github.com/fabriceo/AVDSP/issues
But this will be treated on a best effort basis by the author and without guarantee of action.

Performances:
The overall performance of the xu216 chip considering the possibility to have 2x8 virtual processors and a capacity of 2000 MIPS at 500mhz is great. But it is not at the level of a specialized DSP chip like ADSP21489 or OMAP138L or equivalent. Therefore the user has to optimize the workflow and to distribute the workload on different cores with the only assistance of the --dacstatus report. The provided example “maxload.txt” demonstrate that it is possible to realize the following:
192k, 8 dac outputs : 6 biquads for each output and a delay line.
96k, 8 dac outputs : 20 biquads for each outputs and a delay line.
48k, 8 dac and 8 aes channels : 20 biquads for each input and output.

This is probably good enough for a majority of users and this is why it was decided to release this software. The effort of combining instructions and spreading them over cores is rewarding.

Limitations:
Unfortunately it is not possible to provide useful FIR filtering with the xu216. If required, the user shall investigate other alternatives with a front end player including this capability (like brutefir).

At the time of writing, no graphical interface is proposed. Considering the performance limitation which requires optimization of the workload and core distribution, the effort to generate a list of opcodes based on a generic graphical environment is quite serious. But it would be possible.

Downsampling:
Because of the explained limitations, it was important to provide a solution for automatically reducing the DAC sampling rate according to the user DSP cpu requirements. And this works well, for example the DAC might decide to operate at 48000hz to cope with a very complex DSP program. Then it is recommended that the music sent to the DAC is resampled to this frequency. Otherwise the DAC will downsample the audio stream, to divide the number of samples by 2 or 4.
At the time of writing the downsampling process is very basic and made with a low cost decay convolution. This is not enough to remove aliasing and this can generate distorsion in the high frequencies. A future version might address this with a dedicated core to execute decimation with a 100+ taps FIR filter on some channels (up to 8 at 48k but less at 96k).

Support for DAC8PRODSP:
All support will be done preferably via user community (ASR forum), or by creating an issue in the github repo: https://github.com/fabriceo/AVDSP_DAC8/issues or email at avdspproject@gmail.com