DSP functions in DAC8PRO

Summary:

The digital board of DAC8PRO is 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 DAC8PRO. This document explains how this works and provides step by step examples.

DAC8PRO DSP firmwares:

By default, DAC8PRO runs with two firmwares. One for XU216 in charge of USB interface and data transmission with DAC and AES inputs. One for managing the front-panel (volume, display…) 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 DAC8PRO firmware is fully compatible with the latest standard DAC8PRO firmware and can also run without any DSP program. It does not include any front-panel firmware, so when upgrading, the front-panel firmware will be kept as it is. Minimum version V1.6 is required.

Version and capabilities:

Two versions are provided. When 4 AES inputs are required, then the power available for the DSP program is limited to 250 MIPS, therefore the maximum sample rate is configured at 48000hz.
When a single AES input is acceptable, then the power available for the DSP program is 437 MIPS. Therefore the maximum sample rate is configured to 96000hz. If the target DSP program is too long to accommodate the duration of a single sample (10us at 96k), then the DAC will divide the output sample rate by 2 and will eventually decimate the input by a factor 2 using a preconditioning FIR filter. In this case it is prefered to send audio at 48000hz directly to avoid decimation which inherently brings some distortion.
DSD mode is not possible with DSP firmware.

Summary of firmwares capabilities:

Name (USB device and file name)	USB OUT	USB IN	DAC	AES	CORES x mips	MIPS (total)	total instructions 48k	total instructions 96k
DAC8PRODSP48K	8	8	8	8	1 x 100 2 x 83 3 x 71 4 x 62	100 166 214 250	5208	N-A
DAC8PRODSP96K	8	8	8	2	1..4 x 100 5 x 83 6 x 71 7 x 62	400 416 428 437	9113	4552

A specific 192000 version could be proposed on demand for integrators, depending on requirement and feasibility.
Remark : the DAC8PRO DSP firmware also accepts 44100 or 88200 hz.

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 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 behaviour.

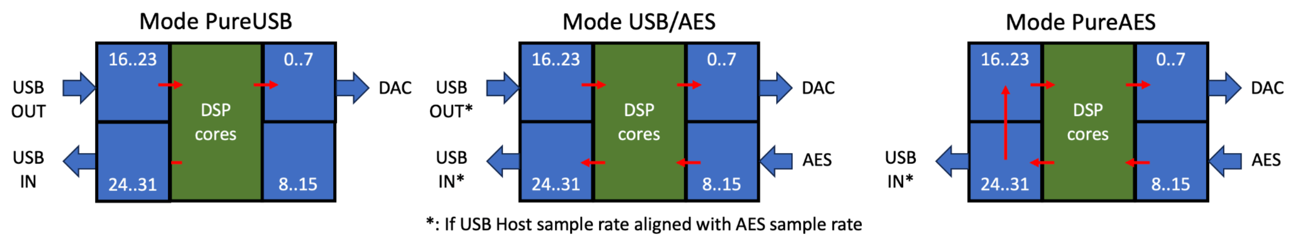
This looks like an intensive effort at first but this gives the possibility to optimize the treatment and to fit a comfortable crossover solution within a limited MIPS quantity. Routing and mixing is also very flexible and efficient as there is no predefined matrix or framework to follow.

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.

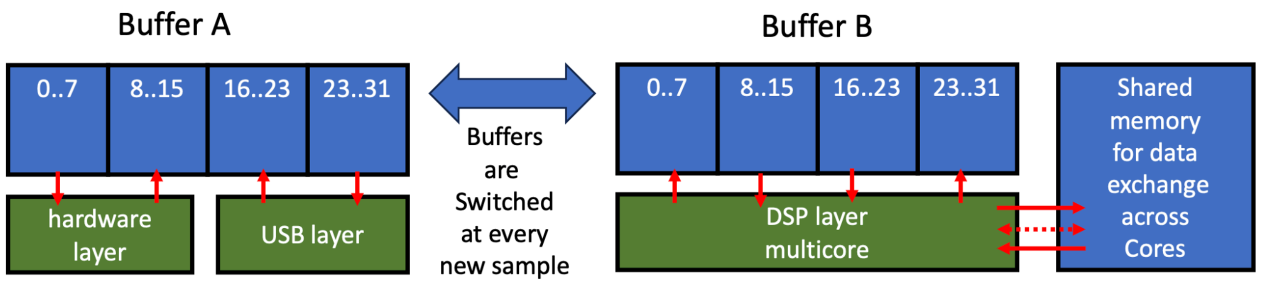
Additional memory locations can be created to share value across cpu cores. For example a core can prepare a stereo input signal with gain and peak correction and save the result in a shared memory location for the other cores to act as a crossover for 8 channels.

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 32 memory location according to the 3 DAC modes:



the following diagram represents the buffer management:



MIPS needed for a typical project
A table in the appendix represents the list of macro instructions available and the time taken to execute them in core instructions which is to be compared directly with MIPS available per core or in total.
A typical flow for filtering and delaying one channel needs 4 macro instructions, and requires 176 core instructions for 6 biquads 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 cores used). An indication is given for the maximum number of typical treatments suggested above.

MIPS	**48000**		**96000**		**192000**	
100,0	2083	11	1042	5	521	2
83,3	1736	9	868	4	434	2
71,4	1488	8	744	4	372	2
62,5	1302	7	651	3	326	1

Remark : To maximize the MIPS capacity, it is recommended to use as much core as possible when the program can be split in similar sections (like a 2x3 crossover solution).

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), 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 -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. As an example:

```
LeftIn          16 ; 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_fc          400 ; 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.
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 first and/or second order 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.

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)
        biquad          LowPass
        saturate ; output LeftLowOut

        inputgain       (RightIn, firstGain)
        biquad          LowPass
        saturate ; output RightLowOut
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 1 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)
        biquad          RoomMode
        savemem         stereoMem[ 0 ]

        inputgain       (RightIn,firstGain)
        biquad          RoomMode
        savemem         stereoMem[ 1 ]

        transfer        (LeftIn, LeftHeadphone) (RightIn,RightHeadphone)

core
        loadmem         stereoMem[ 0 ]
        biquad          LowPass
        saturate ; output LeftLowOut
        loadmem         stereoMem[ 1 ]
```

```
          biquad            LowPass
          saturate ; output RightLowOut
end
```

The saturate instruction protects the signal by reducing it by steps of -6db dynamically if clipping is detected. It is recommended to test each biquad flow with a sweep signal (20..20khz) and with a square signal (e.g. 100hz) to verify that the gain chain will not bring any saturation/clipping situation. Then it is possible to remove the saturate instruction.
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, or to use the saturate instruction before any output to a dac.

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. The input condition is given by the front panel: mode Pure USB : b00000001 mode USB/AES : b00000100 mode Pure AES : b00001000 in addition, the current sampling rate is represented in the bit 4..7: 44/48k : b00010000 88/96k : b00100000 176/192 : b01000000 352/384 : b10000000 the number of USB and DAC channels is coded in the second byte as follow: b0000 stereo version b1111 dac8pro with 8 output and 8 input b1110 dac8pro with only stereo usb output	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. example: core x5 xC0 is only active for pureUSB or PureAES (x5) and only if not 192 and not 384 (xC0)
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
mixer	Execute an equivalent but faster version of multiple inputgain instructions. same parameters as inputgain.	(in, gain) …
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 loadmem

delayus **delaydpus**	Propagate the DSP accumulator msb through a FIFO data line in order to create a delay line. The parameter represents the number of microseconds. delaydpus perform the same operation on 64bits original data and thus requires twice memory.	delay as a numerical value or 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 a savexmem or loadxmem has been used.	no parameter
dcblock	Provides a high pass first order to eliminate and continuous 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 filters based on the list of filters defined by a label and given as parameter	filter label name
param	Restart a label parameter section, enabling definition of labels memory or filters	no parameter
sine	Generate a precise sine wave. Using MCFO algorithm. Remark the thd is almost perfect but amplitude will reduce when frequency increase closer to fs/8	frequency as a numerical value or expression, followed with amplitude
dirac	Generate a one sample pulse. Typically used to test biquad response.	frequency as a numerical value or expression, followed with amplitude
square	Generate square waves.	frequency as a numerical value or expression, followed with amplitude

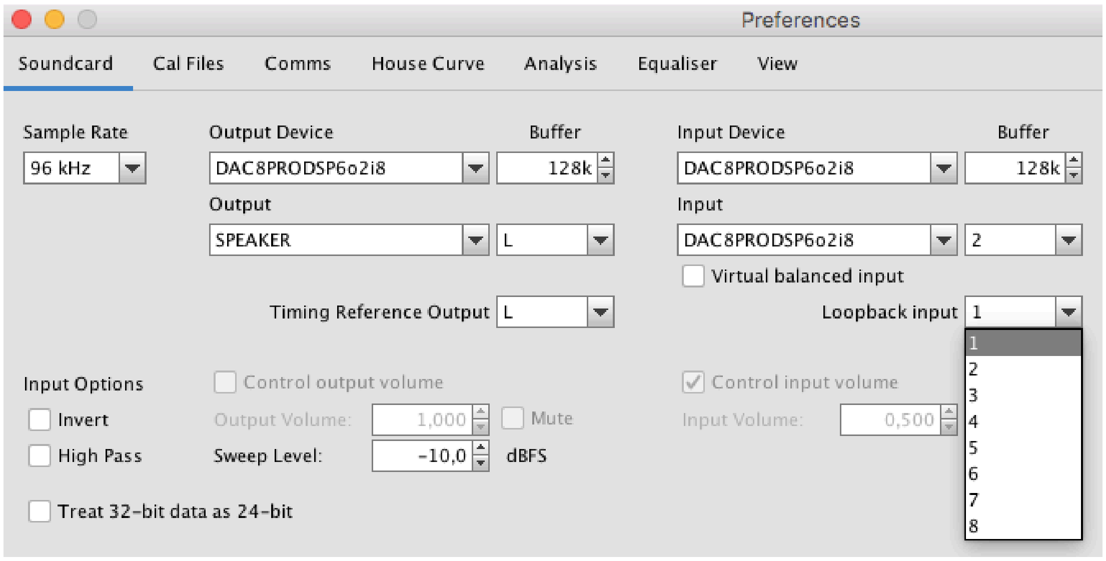
typical cpu requirements for dsp macro instructions

instruction	**cpu instruction**

initialization / core start	tbd
basic input + output	20
basic transfer between one input-output	12 + 6 per transfered in-out
load sample and apply a gain (inputgain)	14
load multiple sample and apply specific gains (mixer)	20 (first inputgain) + 6 x (additional inputgain)
apply multiple sections of 2nd order filters (biquad)	40 + 19 for each 1st or 2ndOrder filter
verify accumulator saturation and eventually apply -6db reduction steps	12 + 2 in case of saturation
same with a final gain applied before (saturategain)	21 + 2 in case of saturation
fifo pipeline to delay a signal (delayus or delaydpus)	20 +3 if delaydpus
store result (output)	10
share data across core (savexmem or loadxmem)	10
fifo pipeline for delaying by one single sample to synchronize core access (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

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 0 (also called Left).

Example of Preference screen :

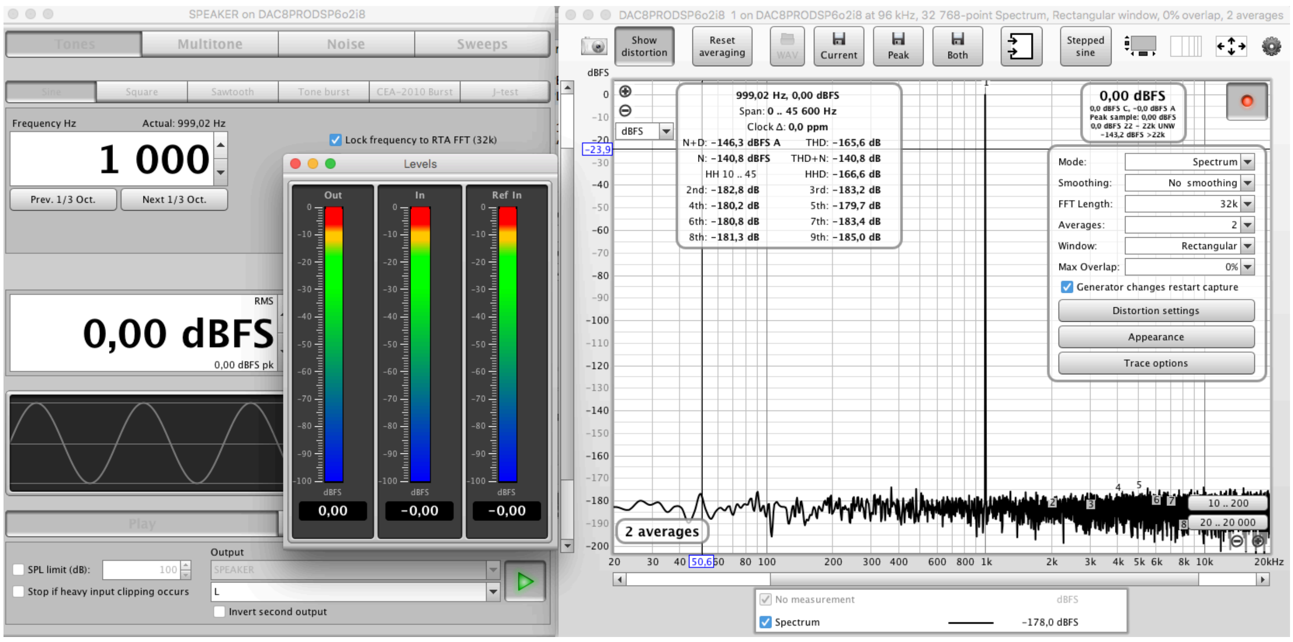


Sample Rate: 96 kHz
Output Device: DAC8PRODSP6o2i8 — Buffer: 128k
Output: SPEAKER — L
Input Device: DAC8PRODSP6o2i8 — Buffer: 128k
Input: DAC8PRODSP6o2i8 — 2
Virtual balanced input: ☐
Timing Reference Output: L
Loopback input: 1 (list: 1, 2, 3, 4, 5, 6, 7, 8)
Input Options
☐ Invert
☐ High Pass
☐ Control output volume
Output Volume: 1,000
☐ Mute
Sweep Level: -10,0 dBFS
✓ Control input volume
Input Volume: 0,500
☐ Treat 32-bit data as 24-bit

Here is a simple test program with a `transfer` function right at the beginning of the program to provide loopback between USB out 0 and USB in 0 (16,24). The original signal is also transfered 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
        saturate ; output 1 ; output 25 #output result on DAC1 and to USB channel 1
        inputgain (16, 0db)         #load sample from USB out 0 (once again)
        biquad  highpass            #apply highpass
        saturate ; output 2 ; output 26#output result on DAC2 and to USB channel 2
end
```

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


SPEAKER on DAC8PRODSP6o2i8

DAC8PRODSP6o2i8 1 on DAC8PRODSP6o2i8 at 96 kHz, 32 768-point Spectrum, Rectangular window, 0% overlap, 2 averages

Frequency Hz: 1 000 — Actual: 999,02 Hz

☑ Lock frequency to RTA FFT (32k)

RMS: 0,00 dBFS — 0,00 dBFS pk

Levels: Out 0,00 — In −0,00 — Ref In −0,00

```
        999,02 Hz, 0,00 dBFS
        Span: 0 .. 45 600 Hz
        Clock Δ: 0,0 ppm
N+D: −146,3 dBFS A      THD: −165,6 dB
  N: −140,8 dBFS      THD+N: −140,8 dB
   HH 10 .. 45          HHD: −166,6 dB
 2nd: −182,8 dB         3rd: −183,2 dB
 4th: −180,2 dB         5th: −179,7 dB
 6th: −180,8 dB         7th: −183,4 dB
 8th: −181,3 dB         9th: −185,0 dB
```

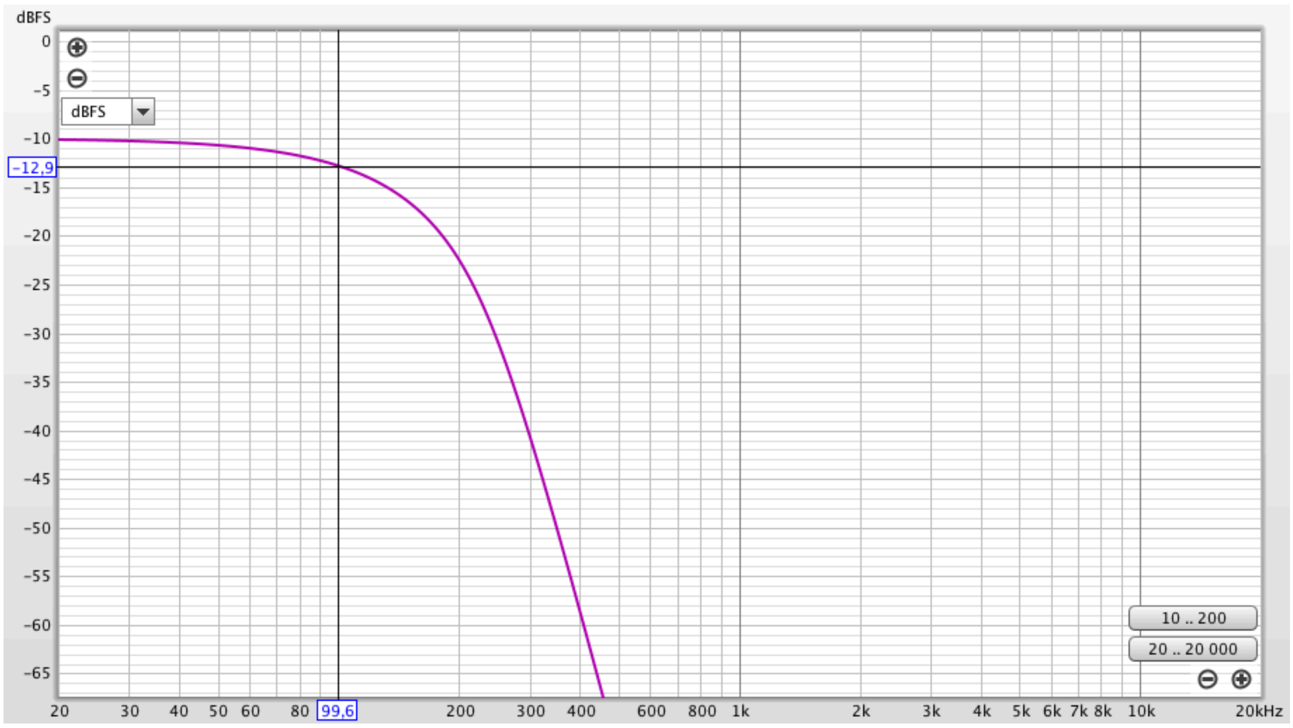
```
0,00 dBFS
0,0 dBFS C, −0,0 dBFS A
Peak sample: 0,00 dBFS
0,0 dBFS 22 – 22k UNW
−143,2 dBFS >22k
```

```
Mode:          Spectrum
Smoothing:     No smoothing
FFT Length:    32k
Averages:      2
Window:        Rectangular
Max Overlap:   0%
☑ Generator changes restart capture
```

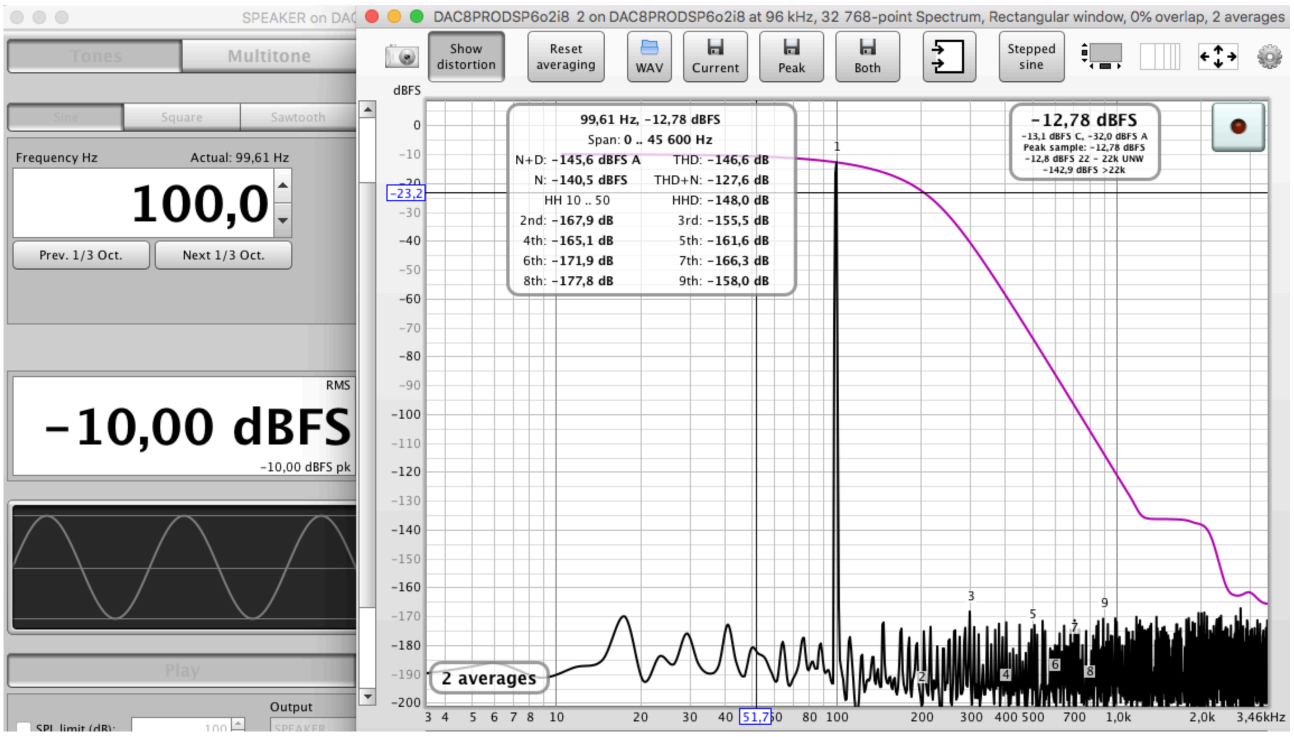
2 averages

☑ Spectrum −178,0 dBFS

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):



SPEAKER on DAC | DAC8PRODSP6o2i8 2 on DAC8PRODSP6o2i8 at 96 kHz, 32 768-point Spectrum, Rectangular window, 0% overlap, 2 averages

Frequency Hz Actual: 99,61 Hz

100,0

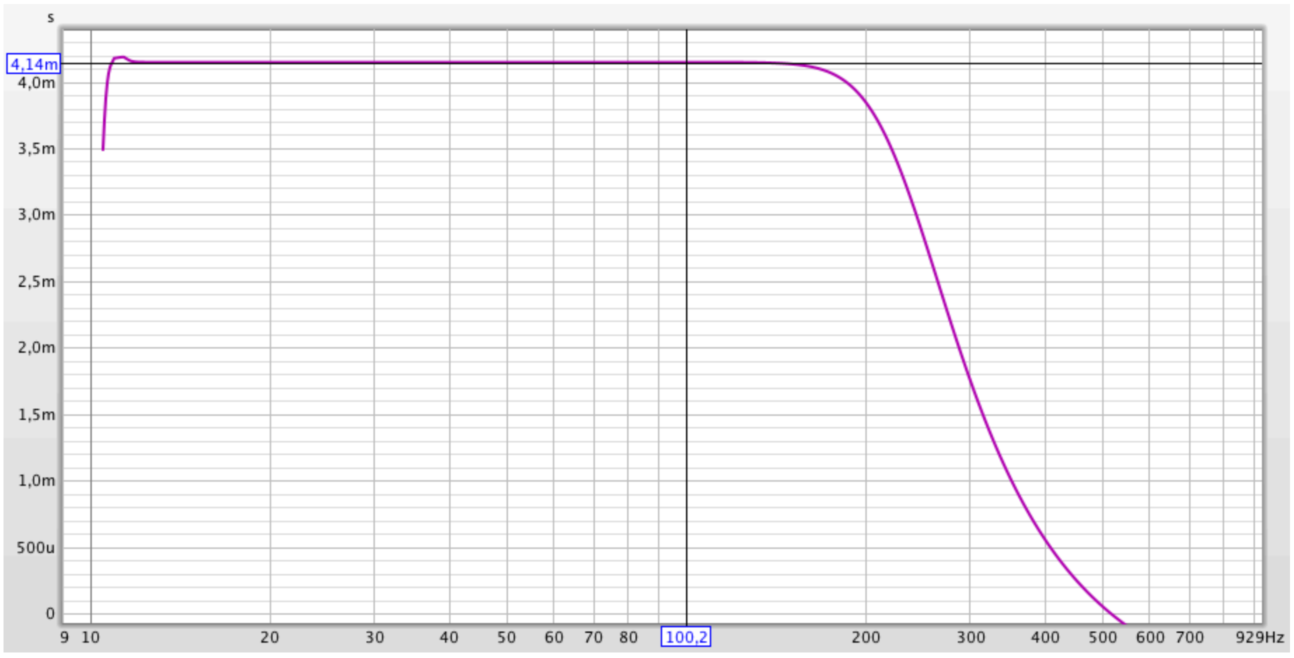
RMS −10,00 dBFS

−10,00 dBFS pk

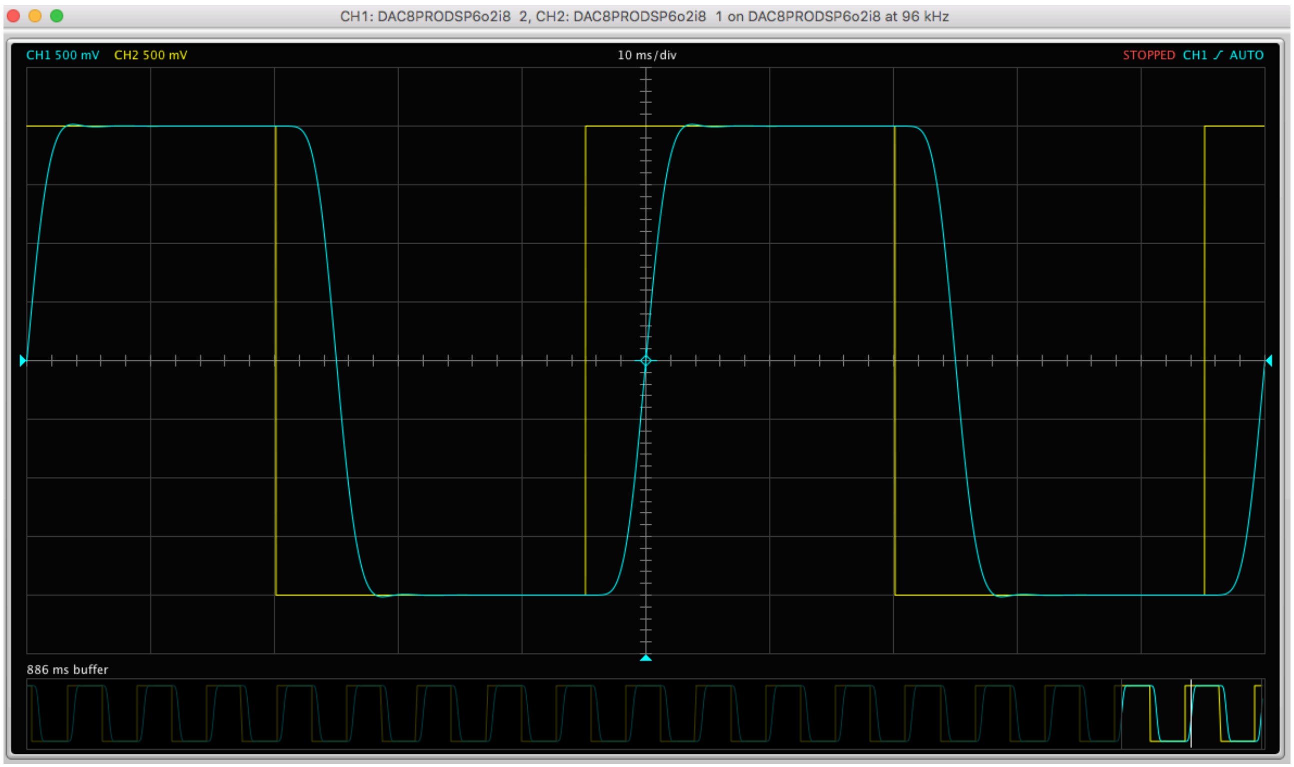
```
99,61 Hz, −12,78 dBFS
Span: 0 .. 45 600 Hz
N+D: −145,6 dBFS A     THD: −146,6 dB
  N: −140,5 dBFS     THD+N: −127,6 dB
    HH 10 .. 50        HHD: −148,0 dB
2nd: −167,9 dB         3rd: −155,5 dB
4th: −165,1 dB         5th: −161,6 dB
6th: −171,9 dB         7th: −166,3 dB
8th: −177,8 dB         9th: −158,0 dB
```

```
−12,78 dBFS
−13,1 dBFS C, −32,0 dBFS A
Peak sample: −12,78 dBFS
−12,8 dBFS 22 – 22k UNW
−142,9 dBFS >22k
```

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:


CH1: DAC8PRODSP6o2i8 2, CH2: DAC8PRODSP6o2i8 1 on DAC8PRODSP6o2i8 at 96 kHz
CH1 500 mV
CH2 500 mV
10 ms/div
STOPPED CH1 AUTO
886 ms buffer

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

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 reduction gain
Bessel with 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	FLP1, FHP1, FAP1 (allpass)	Frequency and optional reduction gain
High and Low shelf first order	FLS1, FHS1	Frequency and gain or reduction

High and Low shelf second order	FLS2, FHS2	Frequency and Q and gain or reduction
classic 2nd order (can be cascaded to create any other filter)	FLP2, FHP2, FAP2, (all pass)	Frequency and Q and optional gain reduction
special 2nd order	FPEAK, FNOTCH, FBPQ, FBP0DB (bandpass)	Frequency and Q and optional gain or reduction

newly introduced tpdf instructions

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 apply a mask to keep only (32-N) useful bits. This is done at each sample rate cycle and the 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 value
        tpdf 18
        input 16
        outputtpdf 27               #16 bits dithered value
core
        input 16
        outputtpdf 28               #20 bits dithered value (taken from first core)
end
```

In fact, outputtpdf instruction 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 any tpdf instruction is defined, the compiler will add one automatically when the first outputtpdf instruction is encountered. Then the default dithering is 24 bits.

saturate instruction details and example

The following program can be tested with the REW scope view, by switching the channels 1 to 4.

```
highpass        HPBU2(400)
core
        square 100, 1.0         #the dsp accumulator is between +1 and -1
        output 24
        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               #the result in the accumulator is reduced between +0.8..-0.8
        output 26               #signal is clean
        copyyx                  #retreive accumulator Y
        saturate                # the result is reduced by 6db automatically to stay between +1..-1
        output 27
end
```

When a saturate instruction detects an overflow situation (in 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 then increments a counter register.
When a saturate instruction sees a number N in this counter register, it reduces the value of the accumulator by N x -6db (shifting one bit right). 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 change the DSP program, or switch the sampling rate frequency, or mute/unmute the DAC.

optimized fixed point 64 bits math explained

The XMOS XU216 is a powerful micro processor with a 32bits instructions sets and some specific instruction able to handle 64 bits. This is used extensively in the AVDSP runtime in order to provide a PureDSP treatment with almost no THD.

The AVDSP runtime is using two 64bits Accumulators called X and Yin this document.
The audio samples provided from the USB host are either 16, 24 or 32 bits signed integers.
With the standard input instruction they are loaded in X with an 8 bits headroom so the 64bits presentation is q56 or s7.56:

b63	b62...b56	b55...b25		b24..b0
sign (0/1)	same as sign	31 usefull bits		25 zeros

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

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 to come to s7.56 format which will be used during all next treatments.
An output instructions use the specific cpu instruction lsat and lextract to quickly saturate and reduce the s7.56 accumulator to a signed 32bits 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 tree multiply instructions in a row resulting in a 96bits value coded s11.84. This is then shifted right by 28bits with two lextract instructions to reduce the accumulator to the regular s7.56 format.
An outputgain instruction also combines tree multiply instructions and then reduce the 96bits result with the lsat and lextract instructions to get 32 useful bits before storing them in memory.
The mulxy instruction multiplies the two 64bits accumulator X by 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 reduces the precision of the accumulator X s7.56 to a 32 bits 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 64 bits s7.56 and will be reused asis in the next biquad 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. The benefit is to get a much larger precision than a 28bits mantissa, without requiring tree 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 is better than using 32bits IEEE float (24bits mantissa+7bits exponents).

The 28 bit base mantissa used for biquad inputs or coefficient and gain value (generating a 56 bit mantissa as described in 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-osx) to better evaluate the benefit of this tuning.

28bits mantissa is a good tradeoff to get maximum performance while providing a reasonable headroom of +42db during basic gain computation and +18db for biquads (but not all-pass filter which requires larger coefficient and limits headroom to +12db).