DSP functionality for DAC8 products

Thank you for choosing the DAC8DSP firmware, part of the AVDSP for DAC8 project.
If you already experienced V1.61 please read information below,
otherwise move on to page 6, Introduction for new users

Welcome to V1.62 for V1.61 users

Version 1.61 was released as a beta version for demonstrating DSP capabilities on the XMOS device while providing a flexible solution to design crossover or bass management programs. Version 1.62 is a more mature version with maximum performances, more functionalities, and is now considered as a release candidate for larger distribution with the DAC8 product family.

Changes

The XMOS runtime has been improved: the opcode interpreter is faster, the biquad routine is optimized by 16%, xy instruction working on both X and Y accumulators reduce execution time, convolution instruction provides FIR filtering capabilities, special warped-convolution provides unprecedented capabilities to equalize speakers, exponential average and moving average filter are introduced, format for 32 bits coefficients and 64 bits accumulators can be changed freely, overclocking is possible, core start/stop is optimized and now identical for up to 7 cores. Spdif/Aes software decoders can be activated one by one upon requirement (typically 5.1 or 7.1 on AES) to maximize mips for PureUSB programs. CPU monitoring and potential overload detection are 99.5% accurate and provide the possibility to tweak and fine tune the DSP pipelines for each core precisely. The cores now use dynamic memory allocation which is important for optimizing memory usage when conditional cores are used extensively; instruction section and elsesection will also optimize the memory usage according to their conditions. DSP programs are now saved in flash memory with larger 32kwords boundaries. Transfer instructions usually used in PureUSB or to provide loopback to the USB host are now embedded in the default processing. Additional 17 instructions to work directly with memory locations. AES output now uses a specific signal not limited to DAC0. A 32bits hash tag is needed to unlock the DAC outputs to their full scale, otherwise a 24db reduction is always applied in addition to the volume displayed. See release note for more details.

The xmosusb utility is slightly improved, with a new `--dspstatus` option which quickly displays the cores cpu load without disrupting the audio streaming, and this is now automatically displayed after uploading a binary file. Available memory is now displayed.

The dspcreate utility is now provided as a single executable (without separate library) and supports much more keywords and DSP instructions. if/else/endif can be used to create conditional compilation, useful when developing DSP programs template with different behaviour depending on constants or parameters chosen; printing in the console can be controlled, either with user messages in the DSP program including label values, or by tweaking the printf verbosity; included files are supported up to 4 levels; parenthesis are allowed in expressions; filter impulse can be declared in line or via included files; runtime estimation of the cpu instructions is displayed for each core with possibility to check each operating mode (PureUSB or AES separately); AES/Spdif cores are declared separately, with conditions. Delayus instructions now support more than 1000us (up to 100ms). Also to simplify the development phase, a new option `-dspload` will call the xmosusb utility automatically after code generation.

An original DSP program developed for v1.61 can be compiled with the new dspcreate utility (called v1.20) without changes, but the `transfer` and `section` instructions usually used to differentiate PureUSB mode could (should) be removed. The overall XMOS CPU consumption will be reduced and the free-ed MIPS can be allocated to provide additional treatments or filtering.
The optimizations provided in V1.62 will not change the characteristic of your DSP program and will not change the sound or the precision of the calculation (when using default values).
remark : *there is no special reason to upgrade from V1.61 to v1.62 if you do not plan to use the new features or to benefit from the CPU optimization : more is not always better !*

Upgrade process

Get the new dspcreate utility and the new xmosusb utility from avdsp_dac8 github (folders v162) according to your operating system.
Put them together in /usr/local/bin (linux or mac osx) or in a location defined in PATH so that they can be launched easily from the terminal or command window.
Install VSCode and avdsp syntax highlight plugin in `.vscode/extensions`. see page

The new firmware V1.62 can be installed in the DAC8with the Windows DFU utility provided by Oktoresearch or their website (as of V1.5), or with new xmosusb utility on linux or mac osx with the command:
OS prompt > *`xmosusb --xmosload dac8prodsp7_162.bin`*

Remark: *if for some reason the transfer is broken before the end (power mains disappear, usb problem or cable disconnected…) then the DAC8 will reboot and will restore the factory version, not the previous 1.61 version ! As of firmware 1.5 the update process is the same anyway so there is no need to re-upgrade with v1.61 before v1.62, so just restart the command above.*

When the transfer is completed, the DAC8 will flash the embedded firmware for the front panel version 1.62, this will take about 60 seconds without indication...
Once done, the DAC8 is rebooted and ready, with the entry **DSP prog** available in the **Filters** menu, and **FW version 1.62** displayed in the **Status** menu. Otherwise unplug USB cable and powermains cable, replug powermains and USB cable, launch xmosusb without option to verify the connection to the DAC8, try again the upgrade process with command above.

Any existing user DSP program in flash will be discarded as the memory map has changed.
Verify in the Filters menu that DSP Prog is not set to 0, otherwise set it to 1 and power off/on the DAC8 with a long press on the volume button in order to permanently save this value in memory.
Launch `xmosusb --dacstatus` to verify the USB connectivity.
Remark: *For Windows, as for v1.61, the xmosusb utility will work only when the winusb driver is installed on "Oktoresearch DFU Interface 3" with Zadig utility, and only after uninstalling Thesycon Asio drivers, or after using the special trick described page* Special USB VID:PID

re-Install your DSP programs

As of now it is recommended to use the VSCode editor and the avdsp file extension.
Rename your source programs with the extension `.avdsp`,
Edit your files with VSCode, the code should now be colored in blue/pink/green/white.
Add the DSPSERIAL xxx sequence at the beginning of the program (xxx is given separately and is specific to your DAC8 serial number).
Compile them and upload them with the simplified command line:

OS prompt > *dspcreate myprog.avdsp -binfile myprog.bin -dspload*

It is always recommended to verify each DAC output response
with a tool like REW before powering the amplifier, the first time.

Save the program uploaded in one of the 4 flash memory locations “x” with command:
OS prompt > *xmosusb --dspwrite x*

These command line actions can be documented in a VSCode file named “tasks.json” so they could be launched easily from the VSCode menu or even with a single keyboard shortcut. An example is provided in the vscode folder on github but requires tuning depending on OS.

Remark : *for Windows, the Asio Thesycon drivers can now be reinstalled, or the DAC8 can be reconfigured to use the original VID:PID as explained page Special USB VID:PID*

to explore new instructions see table Macro Instruction list and details
to try FIR filtering or warped convolution see page Experimenting FIR convolution
to try moving average filters see page Using REW to test CIC filter (moving average):
to overclock see DSPCLOCK a, to change DSP precision see DSPMANT a,b
release note see Version 162

Introduction for new users

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.

DAC8 DSP firmwares

By default, DAC8 products integrate two firmwares. One for Xmos XU216 in charge of USB interface and data transmission with DAC and AES inputs; another for managing the front-panel (volume, display, IR remote…) and adjusting DAC registers according to the USB host status.
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 user's DSP programs that will interact with internal data flows existing between DAC, AES and USB host in and out.
The DSP enabled DAC8 firmware is fully compatible with the latest standard DAC8 V1.6 firmware and can also be used without any DSP program activated.

Versions

For DAC8PRO, a single firmware version is now compatible with the 3 modes (pureUSB, USB/AES, or PureAES). To maximize mips allocated for user DSP cores, the AES connectors can be enabled one by one dynamically with conditions to be defined in the DSP programs.

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
DAC8PRODSP7	8	8	8	1x2	1..4 x 106	422	9625	4812	2406
				2x2	5 x 88	440			
DAC8STEREODSP7	2	2	2	3x2	6 x 75	453			
				4x2	7 x 66	462			

DAC8STEREODSP7 provides the same capability as DAC8PRODSP7. This is probably too much for a pure stereo treatment with PEQ, but this provides the flexibility for users who own a custom version of DAC8STEREO with a DAC board from DAC8PRO exposing the 8 independent connectors available (typical usage as hifi crossover).

Remarks and known limitations

- DSD playback is never possible (silence) when a DSP program is selected in the menu.
- If the user DSP program takes too long to accommodate the time between two audio samples (eg 10us at 96k), then the DAC is reconfigured at a lower rate and the extra samples are skipped (decimation by 2 or 4). To minimize high frequency aliasing due to this basic downsampling, it is better and advised to send the audio stream at a sample rate equal or below the maximum rate supported by the user DSP program activated.

Upgrading the unit

As of version 1.5, the DAC8 products can be upgraded with the Windows DFU utility with following steps:
Connect the USB cable, launch the Windows DFU utility, click on Browse button, select carefully this binary file dac8prodsp7_162.bin for DAC8PRO then click Start button. The DAC8 will reboot in a special USB DFU mode and the utility will start flashing the firmware with an indicator showing the progress. The Windows DFU utility 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 in less than 1 minute. Once complete, wait 1 more minute to complete the second flash process for the front panel, and then 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 DAC8 will reboot and restore the original embedded factory version. This is also an acceptable way to revert back your unit to its original non-dsp firmware if needed.

Remark : choose carefully the filename.bin for the upgrade process as there is no verification done (not possible or too complex).

Enabling a DSP program

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 DAC8 memory with this indicator before playing music, especially at high volume.
When a non compatible stream is sent to the DSP, like DoP or DSD, the firmware will change the 1 squared logo by a 1 “underlined”. When uploading a new DSP program with provided utility, the program number is changed to a squared “zero” whatever was the original DSP Prog chosen.

By default the programs 1 to 4 are “empty” and thus no sound will reach the outputs when selecting an empty program.

When selecting “off” (or 0) in the Filters/DSP Prog menu, no DSP treatment will be provided and the DAC will process the audio stream as for a standard DAC8, then **the full audio stream will be sent to the outputs which the end user must consider carefully before returning to “off”**.

Creating a DSP program

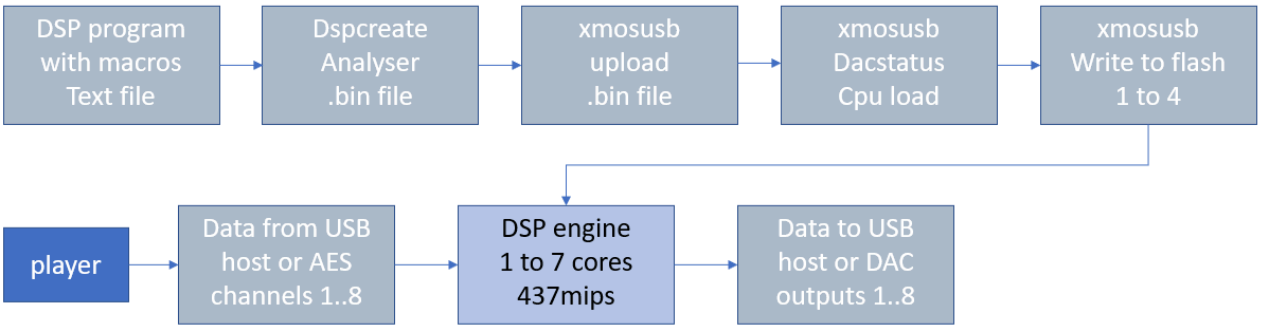
At the time of writing, no graphical user interface is provided to create a DSP flow.

Steps required

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, leveraging the examples provided.
2. evaluating cpu load for this flow and grouping / distributing macro-instructions per core
3. using VSCode 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 and verifying DAC8PRO status and XMOS cpu load for each core with the provided report.
6. testing the expected behavior with third party tools like REW and a sound card, or by adding loopback instructions to verify for each channel : response, total gains, clipping.

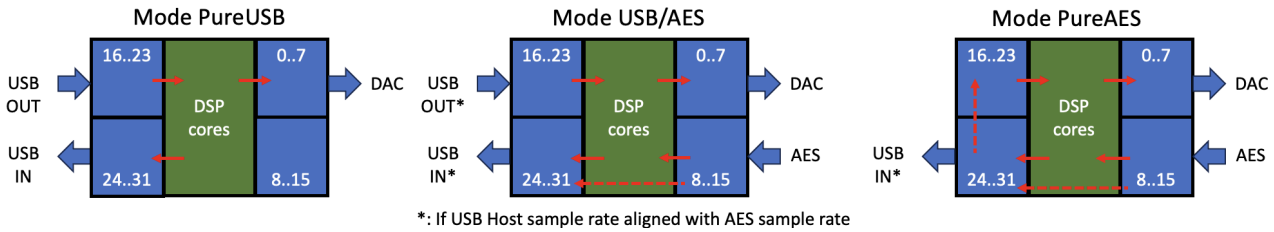
This looks like an intensive effort at first but this gives the possibility to optimize the treatment and to fit a comfortable crossover solution up to 192khz with a limited MIPS. Routing and mixing is also very flexible and efficient as there is no predefined matrix or framework to follow.



IO locations for audio Input and audio Output

The generic model for a DSP flow is : 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 IO locations are predefined for this.
Transformation can be applied to the DSP accumulators with a set of predefined macro instructions like `gain` or `biquad` or `saturate` or `convol` or `delayus...`
The resulting accumulator will be stored as an output sample either for the DACs or for the USB host, in one of 16 other predefined IO locations.
Any core can work with any of these 32 (16 input + 16 outputs) predefined IO locations.

The following diagram represents the 32 IO locations according to the 3 DAC modes:



As of V1.62, 32 additional IO locations exist from value 32 to 63 (not in this picture). They can be used by the DSP program to store 32bits intermediate values or for sharing results across cores. By default, the IO 32 and 33 are assigned to the AES output connector.

As explained later, additional memory locations can be created to share values across 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 8 channels crossover.

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 153 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	44100/48000hz		88200/96000hz		176400/192000hz	
105,6 (4)	2199	14	1099	7	549	3
88,0 (5)	1833	11	916	5	458	2
75,4 (6)	1570	10	785	5	392	2
66,0 (7)	1374	8	687	4	343	2

Remark : To maximize the total MIPS capacity, it is recommended to use multiple cores if possible (5 is optimal) when the program can be split in similar sections (like a 2x3 crossover solution).

background

A DSP program built with the AVDSP framework is a list macro instructions, converted in a list of binary values (called “opcode”) that will be interpreted one-by-one by a specific runtime library optimized for XMOS XU216 and included within the DAC8 DSP firmware.
The opcodes are low-level DSP-like instructions (about 120 different), mainly working on an accumulator or transferring data across temporary memory or from/to input/output memories.
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).

For the time being, the DAC8 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 DAC8 DSP firmware but for simplifying the process, this document describes and focuses on using a text file with a set of predefined macro-instructions.

commands

Analyzing a text file and generating a DSP binary program is done with the following commands:
OS prompt > dspcreate myprogram.avdsp -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.avdsp`.

As explained with more details in next chapters, the binary file generated will be uploaded in the DAC8 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 one of these commands:
OS prompt > xmosusb --dacstatus or *xmosusb --dspstatus*

Since version 1.62, the dspcreate utility can upload the binary file directly by calling the xmosusb:
`OS prompt > dspcreate myprogram.avdsp -binfile mybinfile.bin -dspload`
also, *xmosusb –dspload* now displays automatically the status of each dsp cores

program structure:

The “.avdsp” text file passed as the first parameter has to be structured with following sections:

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

This structure can be repeated as much as needed according to the maximum number of cores available. The 2 first sections for label definition can be grouped at the beginning of the file or spread along for information specific to a core.

A special plugin is provided for `VSCode,` the `visual studio code` open source and free editor, which gives a nice code highlighting feature. It is recommended to use VSCode and to install this module in the sub folder “extensions” located in the user home folder named “.vscode”. to enable it in VSCode, just delete the file named extensions.json, and quit-relaunch vscode.

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 or monitoring). 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:

```
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 ; ratio 33% ; mixer = 1-60%
```

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.
Numbers can be written in decimal, binary or octal, starting with a letter “b” or “x” or “o”.
Numbers can be described as percent by adding “%” at the end, or “milli” by adding “m”, or decibel with the postfix “dB”. 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 in decibel.

A label can be set conditionally, if it was not set earlier in the program or by the command line parameters, by using the “?”. it is possible to force a label to a new value by combining “?” and “=”.

```
fc ? 400    # will set fc to 400 if fc wasn't defined earlier
fc ?= 300   # will set fc to 300 even if it was defined earlier.
```

Comments are possible with “#” preceding character.
To control the compilation process and the exact value of labels, it is possible to print text and label values in the console by using the prefix “#-” and enclosing any label names with “[‘ and “]“:

```
#-*** value of cutoff frequency = [fc], mixer = [mixer] ***
```

Label parameter definition:

There are 3 types of static parameters :*filters*, *memories or impulses*.

FILTER

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 any other filters. As an example:

```
LowPass     FILTER LPLR4(Low_fc)
MidPass     FILTER HP2(Low_fc, 0.7) HP2(Low_fc, 0.7) LPBU3(Mid_fc)
HighPass    FILTER HPBU3(Mid_fc, -2db) HS2(5000, 0.5, +2db)
RoomMode    FILTER 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 for 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 single 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) are scaled accordingly by the encoder.
Filters must be listed on a single line or just below the others by using the backslash “\” symbol.
Since V1.62 the keyword FILTER is used before the list of filters to streamline the syntax, but this is not mandatory, so the previous programs made for V1.61 remain compatible.

MEMORY

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      1
```

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.

TAPS

The keyword `TAPS` defines an impulse to be used for convolutions.

```
fake     TAPS  0,1,0
imp48k   TAPS  "fir48k_2k_3k_60db_59.txt"
imp96k   TAPS  "fir96k_2k_3k_60db_117.txt"
```

The coefficients can be provided in a text file or written on the line, separated with “,” comma. The backslash “\” is not needed on a line ending with “,” comma when the next coefficient is on the next line.
see practical example page Experimenting FIR convolution

Code spread in cores

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
      output       LeftLowOut
      inputgain    (RightIn, firstGain)
      biquad       LowPass
      output       RightLowOut
end
```

The keyword `end` must be unique in the program.
The same code could be duplicated for treating the Mid channels, inside this core or in another core section. Ending 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 duplicated for mid and high filtering.

```
core
      mixer (LeftIn, -6db) (RightIn, -6db)
      savemem      monoChan
      inputgain    (LeftIn, firstGain)
      biquad RoomMode
      savexmem     stereoMem.0
      inputgain    (RightIn, firstGain)
      biquad RoomMode
      savexmem     stereoMem.1
      transfer     (LeftIn, LeftHeadphone) (RightIn,RightHeadphone)
core
      loadxmem     stereoMem.0
      biquad       LowPass
      output       LeftLowOut
      loadxmem     stereoMem.1
      biquad       LowPass
      output       RightLowOut
end
```

output behaviour and DSPSERIAL

By default, any `output` or `outputgain` instruction will clip the signal between (+1..-1) if the value of the accumulator is outside these boundaries. Still this behavior is not satisfying as it could generate a high level of harmonics. Therefore it is highly recommended to tune the gain chain to avoid uncontrolled clipping situations.
As an alternative, it is possible and recommended to use *`saturate`* or *`saturatevol`* or *`outputvolsat`* instructions as explained later in the document, which provide a dynamic reduction only when needed.
Remark : the DAC8 volume control is independent of the DSP and acts directly on the DAC outputs, but it is possible to split the attenuation between DSP and DAC as explained later about *`saturatevol`*. This gives the possibility to accept saturation if the final volume is below 0db.
The keyword DSPSERIAL xxx must be defined at the beginning of the program in order to unlock the DAC output to their full-scale capabilities, otherwise each DAC will provide the expected signal but with 24db attenuation. the value xxx is provided to each customer separately.

Language reference

Macro Instruction list and details

keywords	description	parameters
core	Define the beginning of a new code section for a new DSP core. A core can be activated or not, depending on a flexible 32 bits condition checked with 1 or 2 parameters. See detailed information later in this document. Multiple enabling conditions can now be defined, separated by coma.	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 required to be 0 for this core. This can be repeated.
end	mark the end of the whole DSP program. Any text after is ignored.	no parameter
transfer **transfer2** **transfer8**	transfer one or many samples from a sample location to another. by extension transfer2 and transfer8 will transfer 2 or 8 contiguous samples in a row.	(in, out), … multiple couples possible
input **inputx** **inputy**	Load a sample from a memory location directly in the X (default) or Y DSP accumulator, without applying conversion or gain.	immediate numerical value or expression representing one of the 32 locations.
output **outputx** **outputy**	X (default) or Y DSP accumulator value is eventually clamped between -1 .. +1 and saved in one or many memory locations.	immediate numerical value or expression representing one of the 32 locations. Multiple outputs possibly separated by comma.
outputtpdf	add tpdf noise to accumulator and proceed output instruction	out, … multiple output possible
outputgain	Apply a gain or reduction to DSP accumulator X and proceed with output instruction	(out,gain)
outputvol	Apply a portion of the volume and proceed with output instruction. Accumulator is not changed	out , … multiple output possible
outputvolsat	Combined instruction equivalent to **saturatevol** and **output**, but accumulator not changed	out , … multiple output possible
inputgain	Load a sample from a memory location in the main DSP accumulator and apply a gain or reduction.	(in, gain) 0<= in <32 -8 < gain < +8, or -99db < gain < +18db
mixergain	Adds multiple inputgain in the main X DSP accumulator. The addition may overload the accumulator if the size allowed by DSPMANT is not compatible. see DSPMANT	(in, gain) …
mixer	add multiple inputs without applying any gain.	list of input with comma

	See DSPMANT to size the accumulator.	
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] or .index	memory label name with an optional [value] or .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 **delayusx** **delayusy** **delaydpus**	Propagate the DSP accumulator msb (32bits only) through a FIFO buffer in order to create a delay line. The parameter represents the number of microseconds. delaydpus performs the same operation on 64bits original data and thus requires twice the memory.	delay in microseconds as a numerical value or expression. Value is internally rounded to be a multiple of a single sample duration.
delayone	Swap the DSP accumulator with its value at the previous sample, thus creating a delay of one sample. Typically used to synchronize outputs across cores when savexmem or loadxmem are used.	no parameter
dcblock (will be deprecated)	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 list of filters.	frequency as a value or expression 10 < f < 100
saturate	Eventually modify the DSP accumulator so that the value is constrained between -1..+1 . If a saturation is detected, a future gain reduction is applied by steps of 1db, up to -15db.	no parameter
saturategain	Combined instruction: applying a gain or reduction to the DSP accumulator and proceeding with **saturate** instruction as above	gain as an immediate numerical value or as an expression.
saturatevol	Apply a portion of the volume on the accumulator and proceed **saturate** instruction	no parameter
biquad **biquadx** **biquady**	Compute one or multiple biquad sections based on the filter defined by a label and given as a parameter.	filter label name, representing a list of 1st or 2nd order filters.
param	Restart a label parameter section, enabling definition of labels memory or filters, below.	optional numerical number, see specific documentation
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
tpdf	generate 2 successive random numbers and combine them to generate a triangular distribution. if a value is provided as a parameter, it is used to define the dithering.	8 <= value <= 31 bit mask for dithering when using outputtpdf

white	load the dsp accumulator X with the latest random value generated by a tpdf instruction. This noise is white and its rms value is -1.5dbFS	
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	Subtract two registers : X = X - Y, or Y = Y - X	
mulxy, mulyx	Multiply two 64 bits registers X and Y and store result in either 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 (=180° phase shift)	
New instructions as of v120 for firmware as of v1.62:		
integrator	Integrates the value of the X register over time with formula: yn = 0.999 yn-1 + xn. (this corresponds to 6db/octave attenuation).	
movingavgn **movingavgus**	execute a moving average filter (CIC) for a certain amount of samples, or a fixed time in microseconds (independent of fs). Possibility to cascade them. The result is already scaled by 1/N to compensate for the N accumulations.	1 < N < 10000 or 1< time < 100000us where N will be computed dynamically by N=fs x time
expmovingavg	Compute an exponential moving average filter using the given factor “alpha” with the formula: yn = yn-1 + alpha * (xn - yn-1)	0 < alpha < 1.0 or 0% < alpha < 100%
delayusfbmix	32bits delay line as delayus , with input and output mixer (wet and dry) signal. providing a reverberation effect. coef1 applied on the source (accumulator). coef2 applied on the feedback (delayed). The resulting mixed signal goes in the delay line. coef3 applied on the output of delay line coef4 applied on the mix	delay in micro seconds, followed by 4 coefficient between 0…1 typically described as %
thdcomp	apply thd compensation on h2 and h3, with 2 coefficients c2 and c3 (usually in negative decibel).	c2,c3 between -20db…-140db
convol	execute convolution (FIR filter) according to a list of impulse provided (one per sample rate)	list of impulses, see specific documentation
warpconvol	execute a warped convolution with factor “lambda”	-0.999 < lambda < +0.999, list of impulses
sine	generate a pure sine at frequency f	f < fs/2

clrmem **swapmem** **memadd** **memsub** **memavg** **memneg**	execute actions on target memory location, using accumulator X as second operand. mem = mem op X	memory
addmem **submem** **mulmem** **divmem** **avgmem**	execute action on accumulator, using memory location as second operand X = X op mem	memory
loadmem **savemem**	exactly same as savexmem and loadxmem	
mixermem	accumulate all value from a list of memory locations	memory, …

Cpu requirements for macro instructions:

Since V1.20 (for V1.62), the dspcreate utility is estimating the number of instructions required for each core. To generate the report in the console, add the keyword DSPXS2 at the beginning of the program. After downloading the program with the xmosusb utility, the details of the instructions required for each core is displayed with total accuracy. This includes 21 instructions for the core start/stop. More information can be extracted with the keyword `instructions` explained later.

List of supported filter names

type	**label name** (LP means low pass, HP means highpass)	**parameters**
Bessel (standard)	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	LS1, HS1 (1st order)	Frequency and gain
High and Low shelf second order	LS2, HS2	Frequency and Q and gain
classic 2nd order	LP2, HP2, (can be cascaded as needed) 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 upload.

Clone the github directory avdsp_dac8 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.avdsp` file with your preferred text editor.
with a terminal session or command line window, convert the file as a binary file:
`dspcreate example1.avdsp -binfile ex1.bin`
Select a DSP prog 1 to 4 in the Filter menu (even if empty, this enables DSP runtime).
A squared 0 is now displayed in the top left corner of the volume screen of the DAC8, (because there is no program in the flash memory at the beginning).
upload the resulting binary file with
`xmosusb --dspload ex1.bin`
verify status and cpu load with
`xmosusb --dspstatus`

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.

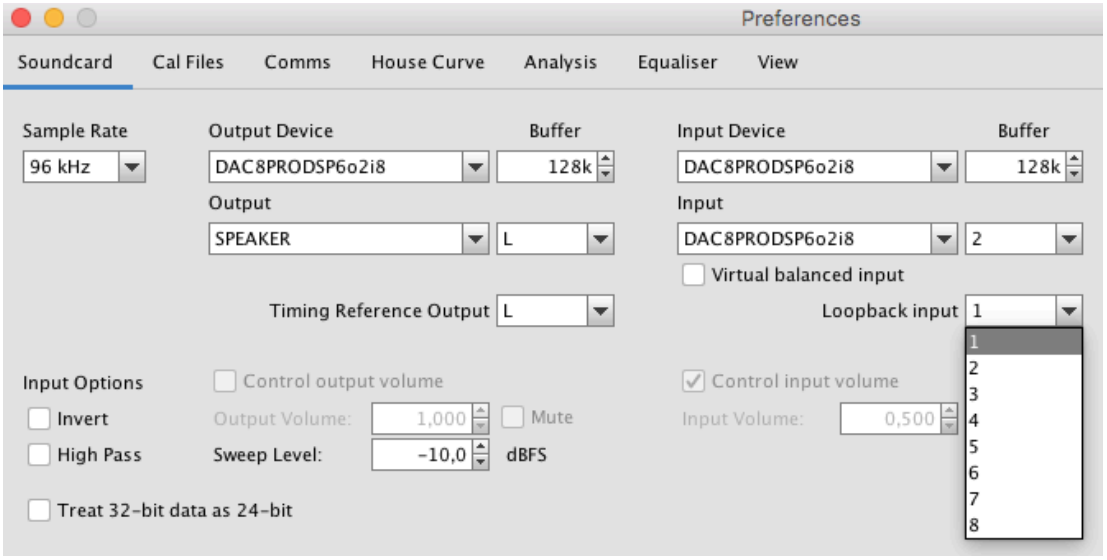
Once finished, write the binary file loaded to a flash memory location between 1 and 4.
`xmosusb --dspwrite 1`

remark: when using instruction *`saturate`* or *`saturatevol`* or *`outputvolsat`*, clipping will be detected and displayed during music playback in the top right corner of the right screen with a value in decibel written on a white background.

Using REW to test a 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 :



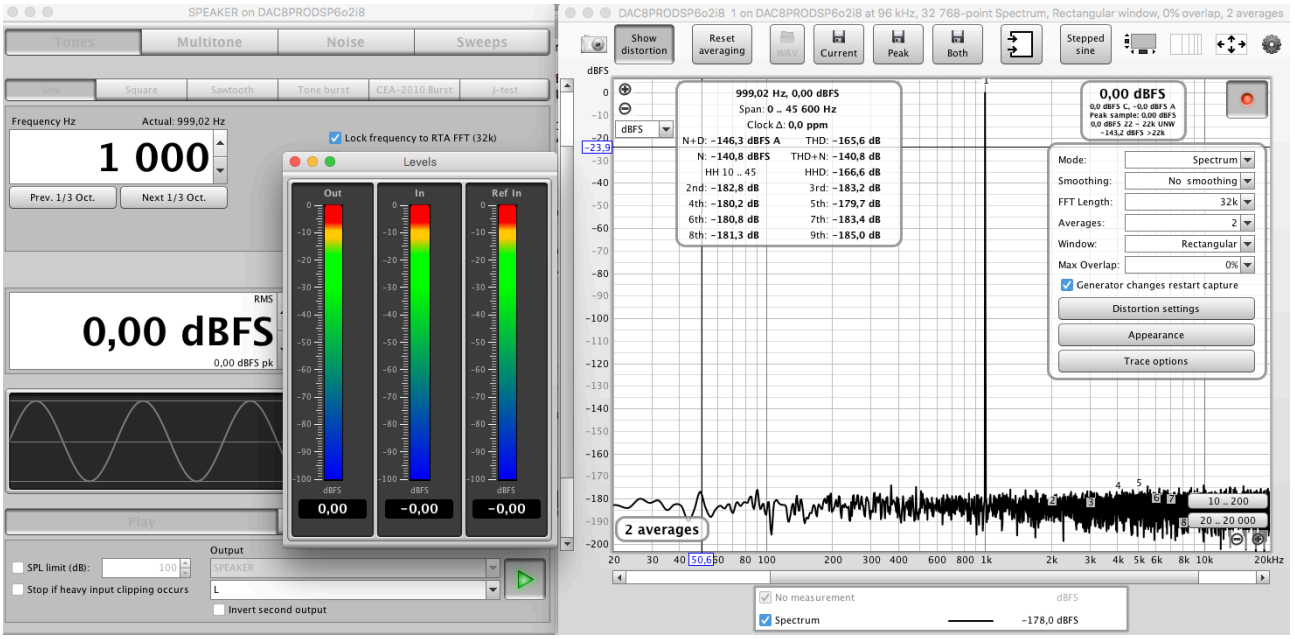
Here is a simple test program with a *transfer* function at the beginning of the program to provide loopback between USB-out 0 and USB-in 0 (16,24). A lowpass filter is provided on DAC 0 and USB-in 1, a high pass filter is provided on DAC1 and USB-in 2:

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

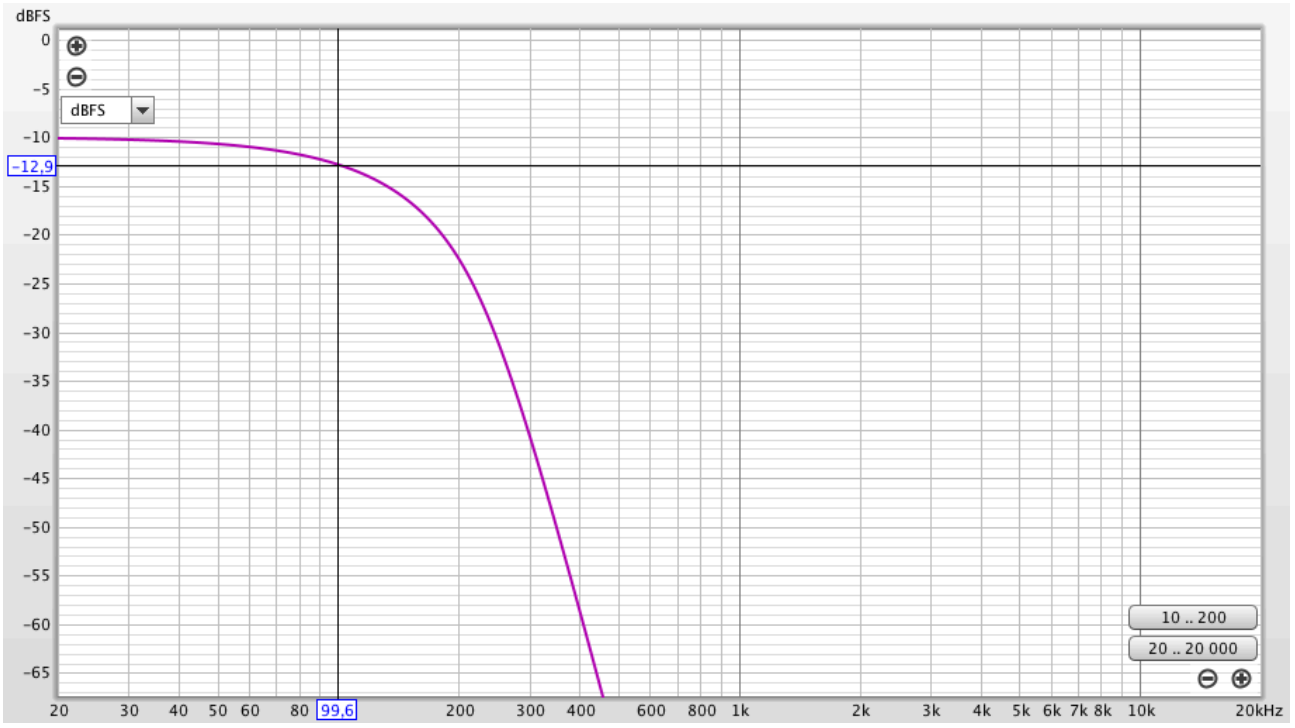
remark: in REW, the input channels corresponding to “USBin” are numbered from 1 to 8, and the corresponding memory location in the DAC8DSP are numbered 24 to 31. instead it is possible to declare labels and to use them like this:

```
rewout 15 ; rewin 23
left = rewout.1 ; right = rewout.2
core
      transfer (left, rewin.1)   #loopback
      input left ; biquad lowpass ; output rewin.2
end
```

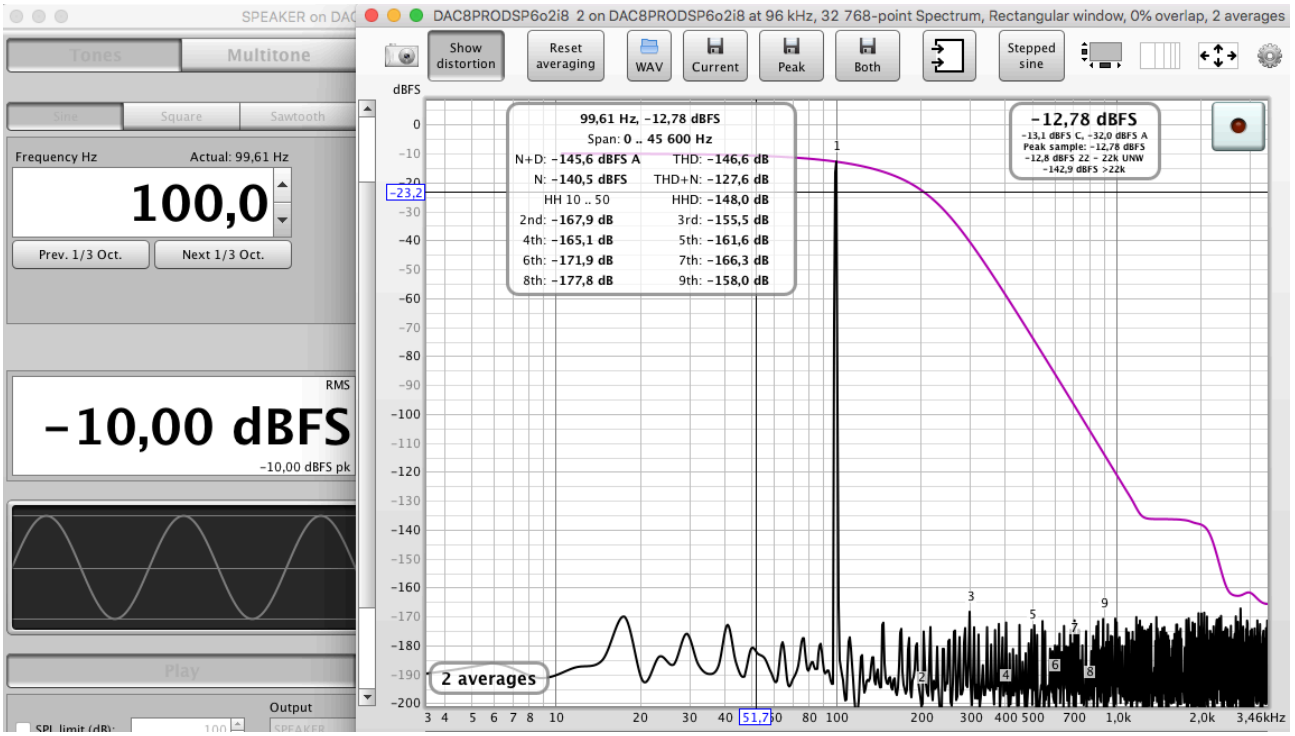
Resulting screen when testing loop back with a 0db sine wave as input: no clipping , no THD



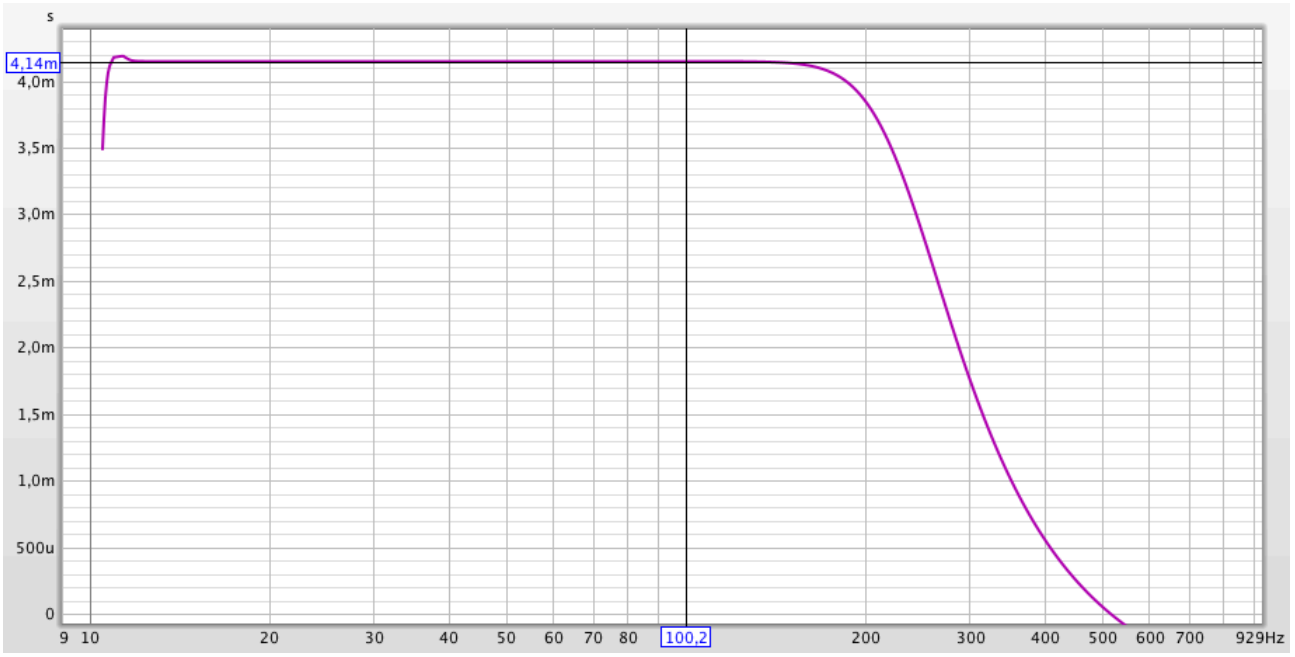
Sweep Measurement on channel 1 with -10db signal in range 10-20000:
LP Bessel 8th response is -2.9@ 100hz, -31@300hz, -77@600hz so 46db/octave (300..600)



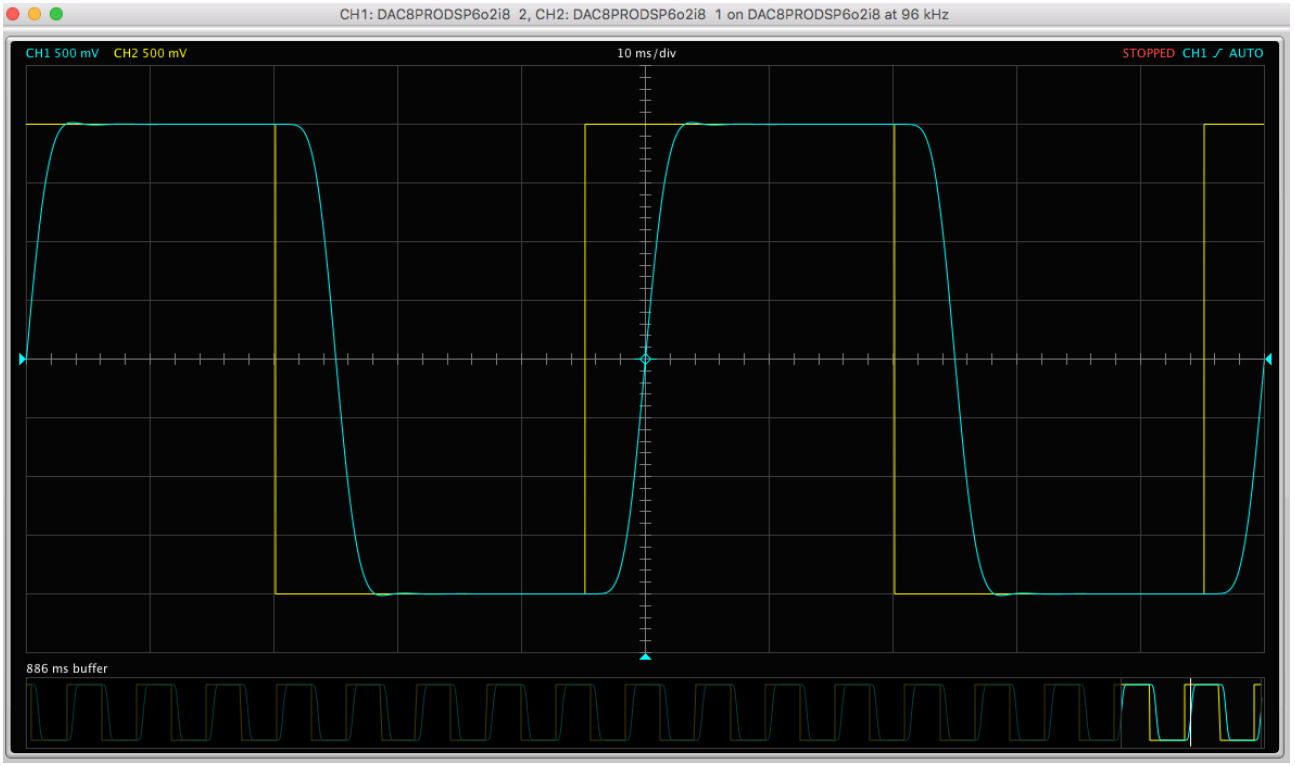
There is no THD for a 100hz test tone (thanks to our specific biquad routine):



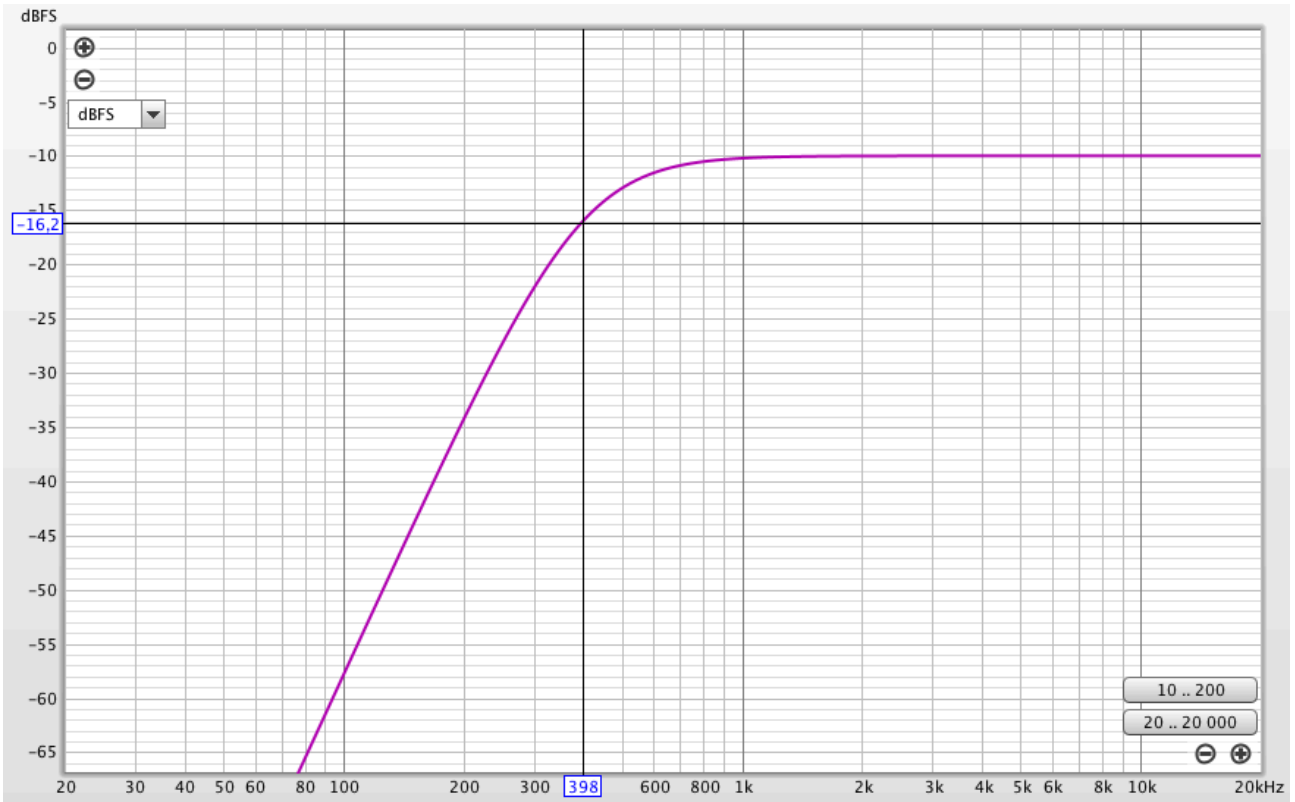
As expected, the group delay of the Bessel filter is flat below the cutoff frequency, at 4.14ms.



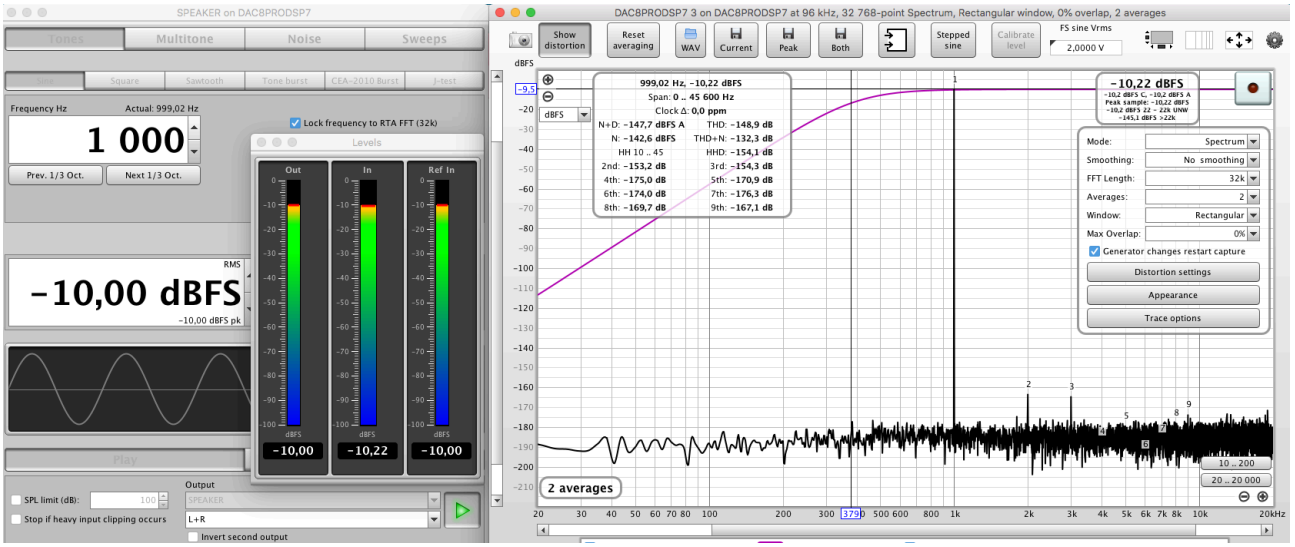
The scope view show a perfect damped step response to a square wave (0db 20hz here) with no clipping:



For the channel 2, response of the high pass LR4:
fc 400hz at -6db



and no thd in the pass band:

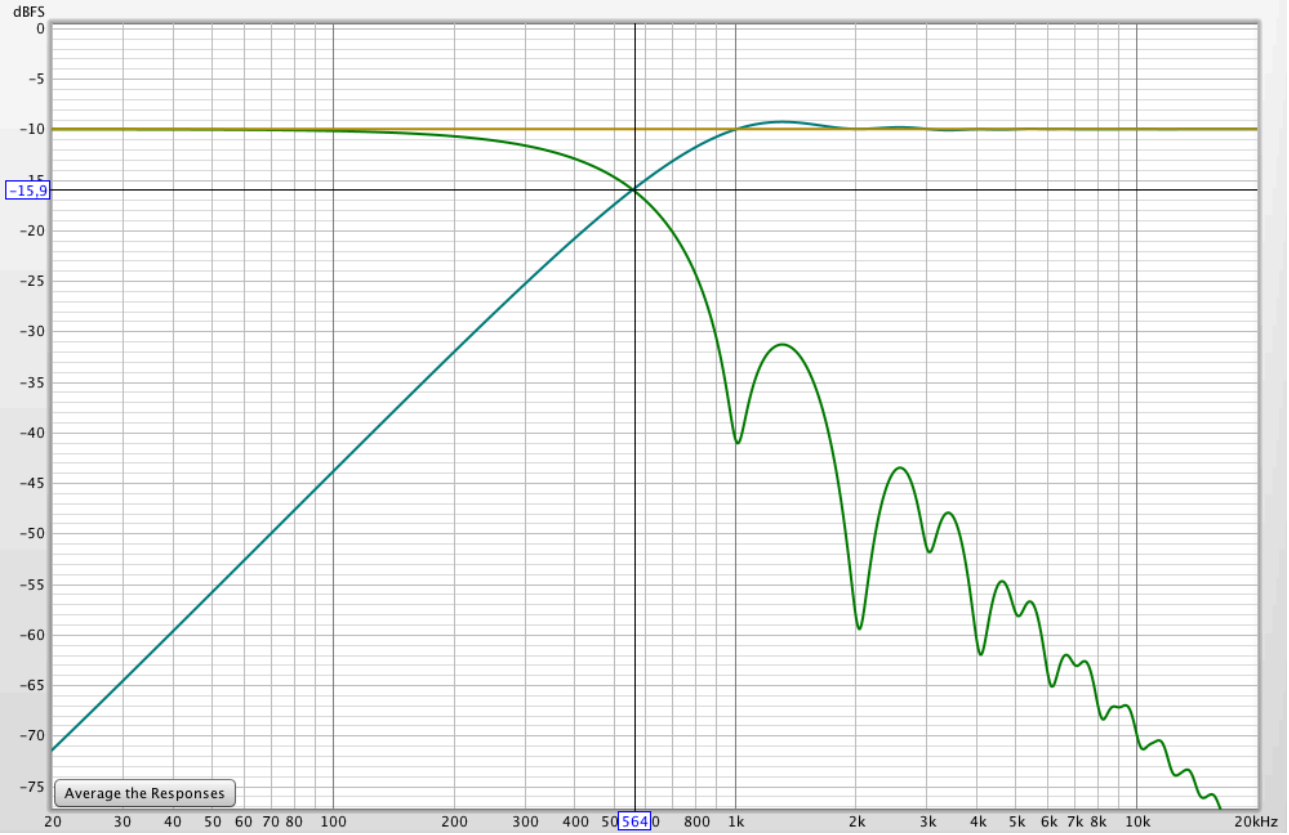


Using REW to test CIC filter (moving average):

the dspcreate utility as of v120 combined with dac8 dsp firmware as of version 1.62 integrates some new instructions, one being movingavgn providing a moving average filter with “N” delay cells, and movingavgus based on the same algorithm, but with N being dynamically computed depending on the given time in microseconds and the actual sampling rate. Cascading multiple filters provides quite good low pass filters with total linear phase behaviour. The high pass can be obtained by subtraction, after a delay equivalent to the CIC group delay which is half of the number of “N” delay cells for a one stage filter, but a more complex formula for cascaded filters. As an example:

```
time  1000  #this will provide a 560hz crossover at -6db
gd    730   #measured group delay for the 2 cascaded CIC filters
core
      transfer (16,24)
      input 16
      movingavgus time
      movingavgus time/2
      copyxy ; output 25          #filtered signal on Rew channel 2
      input 16 ; delayus gd       #delay original signal
      subxy ; output 26           #subtraction for high pass signal
      addxy ; output 27           #sum of the 2 signals on Rew channel 4
end
```

This provides the following low pass and high pass response, with a flat sum and constant group delay of 730us. The low pass can be compared with a Bessel 8th order at 750hz.
The high pass looks like a second order filter at 12db/octave.

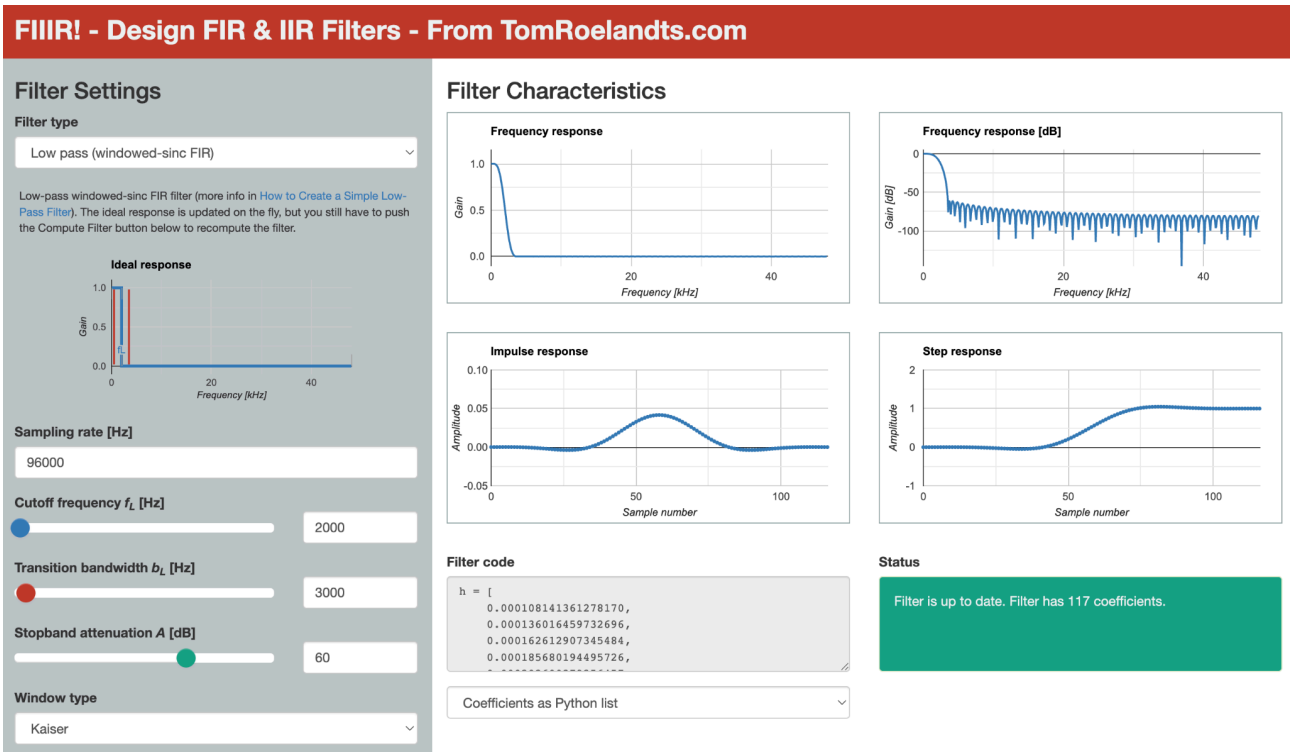


THD+N is extremely low as this filter uses mainly additions without reducing the original 32 bits precision of the samples.
Compared to a Bessel order 8th, the number of cpu instructions needed is 30% lower with the 2 cascaded movingavgus instructions. And the group delay is much lower.

Experimenting FIR convolution

The XMOS XU216 does not integrate any hardware accelerator to facilitate convolution. Instead we have to use the 32x32 bits multiplier (64 bits result) and some optimization to manage a table of coefficients (taps) and the circular buffer for the corresponding samples fifo.
The new runtime provides this capability with a requirement of 2,75 CPU instructions per taps. Considering 1100 instructions available per core at 96k, this gives the possibility to perform a FIR filter with about 370 taps on each core (up to 4 cores for maximizing MIPS). Using overclocking up to 600mhz will raise this to about 420 taps. This is relatively limited and can be used in the context of equalizing or filtering high frequencies above 500 hz or so.
At 48k (or 44k1) the number of taps will double, and at 192k it would be divided by 2.

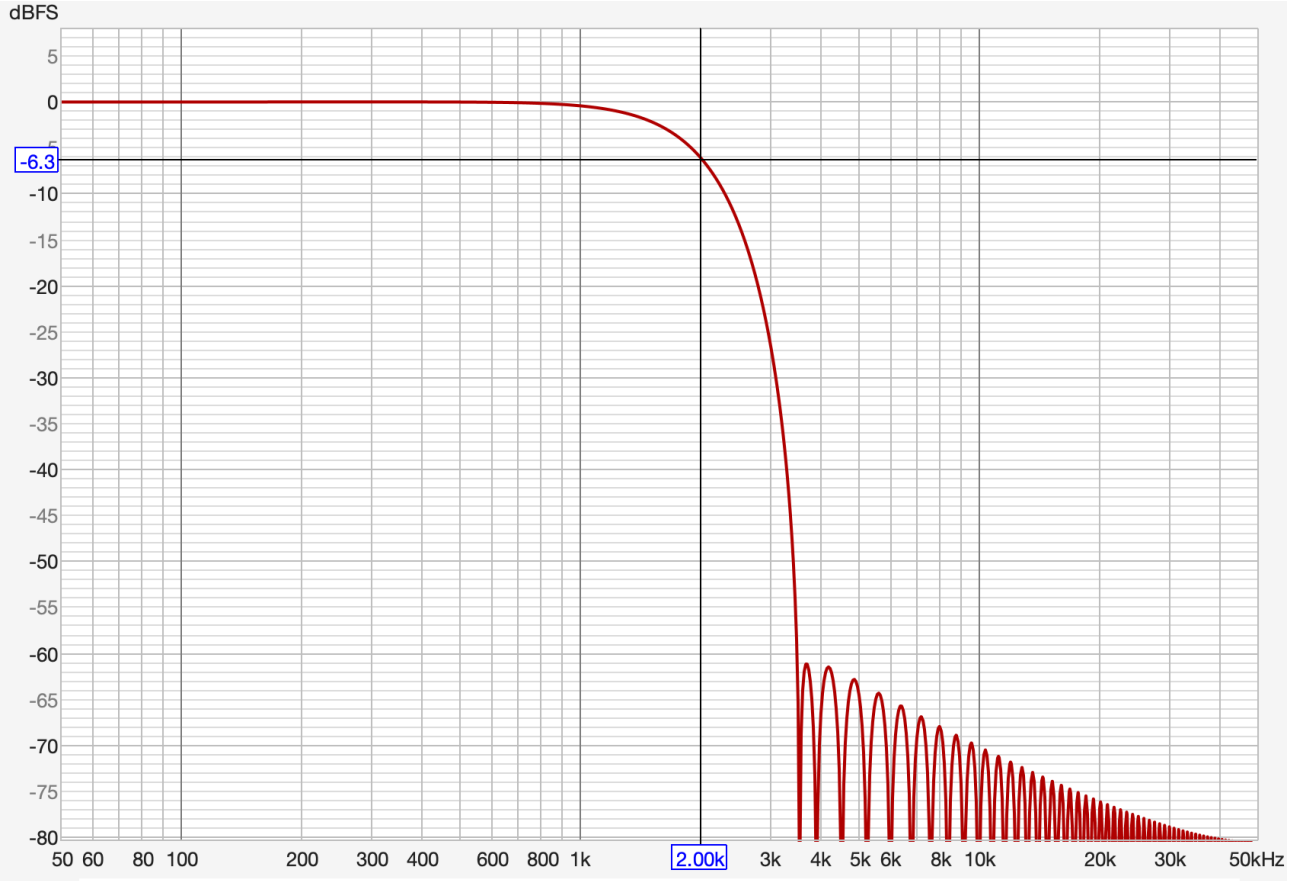
The following example uses a filter impulse made with the online tool available at https://fiiir.com. Thanks to the author Tom Roelandts. The filter is low-pass cutoff 2khz , 3khz transition bw, 60db.



the provided coefficients are saved in the file fir96k_2k_3k_60db_117.txt and fir48k_2k_3k_60db_59.txt, respectively for the impulses at a sampling rate of 96k or 48k.
For the sake of the demonstration, 44k1, 88k1, 176k2, 192k are not generated but replaced by fake impulses. The following program can be tested with REW:

```
fake    TAPS  0,1,0
imp48k  TAPS  "fir48k_2k_3k_60db_59.txt"
imp96k  TAPS  "fir96k_2k_3k_60db_117.txt"
core
      input 16
      convol fake,imp48k,fake,imp96k,fake,fake
      output 25
end
```

Result for the given filter. At 96k, the group delay is 604 microseconds which corresponds to the filter size/2 (58) divided by 96000



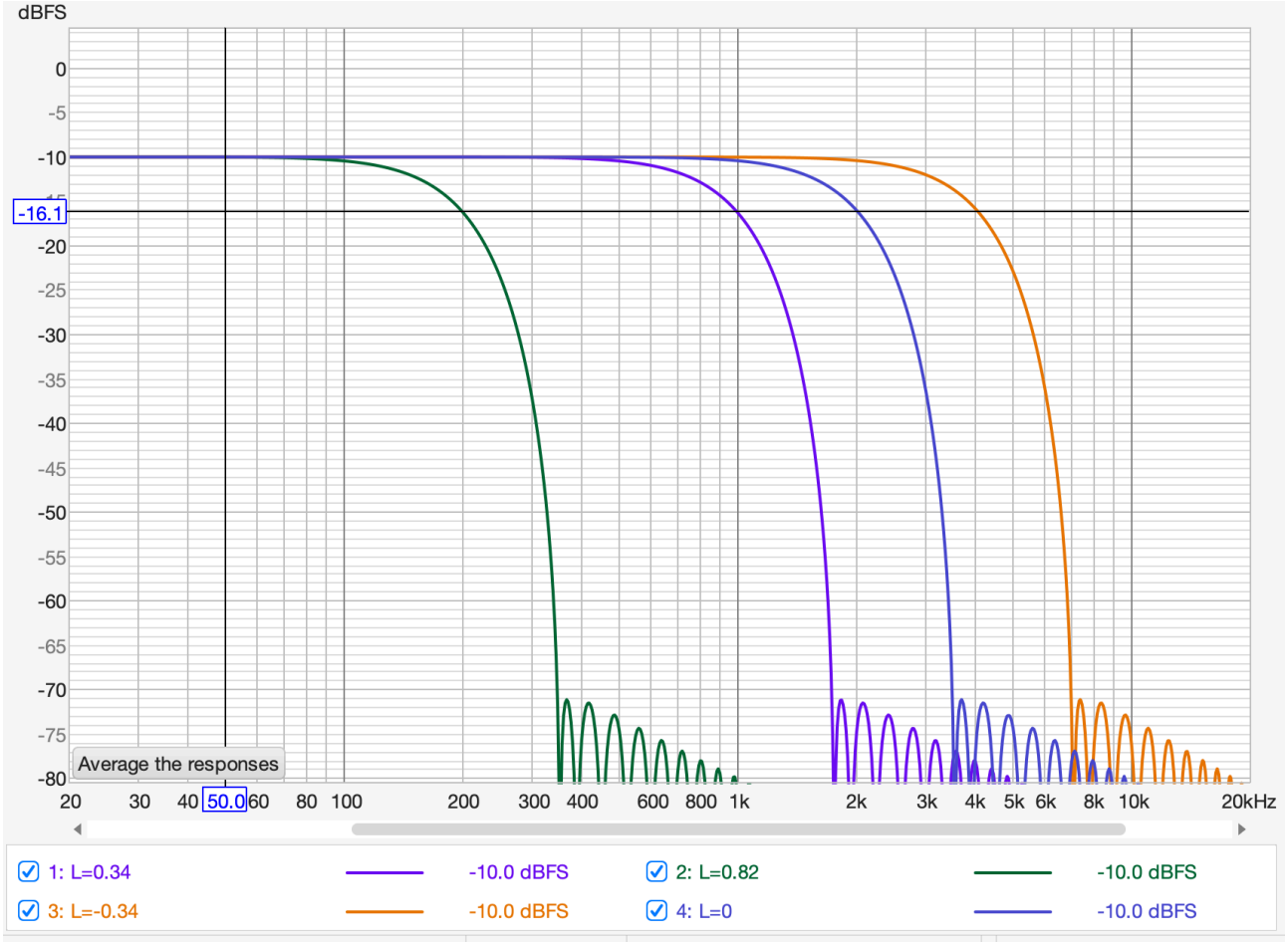
Experimenting Warped convolution

By replacing the delay cell of the FIR filter by a first order all pass filter defined with a coefficient “lambda”, it is possible to execute the convolution in a semi-logarithmic “warped” frequency domain. This gives the possibility to apply a filter on a more focused area of the frequency domain, typically in the bass region where traditional FIR would require thousands of coefficients.

The DAC8 runtime now includes a `warpconvol` instruction which provides this at the expense of 5,25 CPU instructions per tap, but this is highly compensated by the possibility to divide the frequency characteristics of the filter by 5 to 10.
The filters defined above can be used as-is with a “lambda” factor “L” and this will change the cutoff frequency. For example:

```
L= 0.82 # L= 0.34 ; L=0 ;  L= -0.34
core
      input 16
      warpconvol (0,fake),(L,imp48k),(0,fake),(L,imp96k)
      output 25
end
```

The resulting figures for L=+0,82 and L=+0,34 and L=0 and L= - 0,34



The group delay remains almost flat in the passband area but it is larger than with FIR : for L = 0.82, the group delay is 6 milliseconds, 10 times the original FIR filter but its cutoff is now 200hz.

It is possible to generate specific filters for a given lambda factor by using the traditional tools available with matlab or scipy library. The process is to convert the expected response in the warped frequency domain and then to apply the usual process (like remez) for creating impulse.

warping the frequency domain can be done using the equation and example available in this spreadsheet:
https://docs.google.com/spreadsheets/d/16pKxGXOOSBv7V8b1EXAgFfpFKuwrGvPXLKzaNfdrPHk/edit?usp=sharing

Remark: *Both* `convol` *and* `warpconvol` *expect a list of filters separated with "," comma. By default the frequencies supported by the DAC8 and by an avdsp program are 44k1, 48k, 88k2, 96k, 176k4, 192k so it is expected to provide 6 impulses. if less are provided, then the dspcreate utility will use dummy impulse to fill the gaps.*
using DSPPRINTF 3 will display some information about each impulse (taps, sum, max and position of the highest value.
For `warpconvol`*, each impulse is written within brackets (L,imp), representing the lambda factor and then the name of the impulse. So each frequency can have its own lambda factor. This can be used to cover the 6 frequencies with a single impulse and 6 different lambda factors to inherently realign the filter with the sampling rate.*

Operating system specificities

For both Mac OSX and Linux, it is possible and recommended to copy both utilities in the folder /usr/local/bin with the following commands executed in a terminal or shell window, so the utilities can be launched from any other path location and without needing prefix ./

```
cp xmosusb /usr/local/bin
cp dspcreate /usr/local/bin
```

eventually use sudo before these commands

Mac OSX

Utilities are compiled for x86_64 and for arm for Apple silicon Eventually the user might need to change the access rights of the utility with the command:

```
chmod +x xmosusb dspcreate
```

At the time of writing, the utilities have been tested and successfully run on an old iMac intel core i5 with High Sierra and on Macbook pro M1pro with Sequoia.
Mac OSX usb audio host driver is compatible with DAC8 products in 8 channels mode. Configuring the USB with the Midi utility in another mode like 2 or 6 channels will not change the 32 IO locations 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

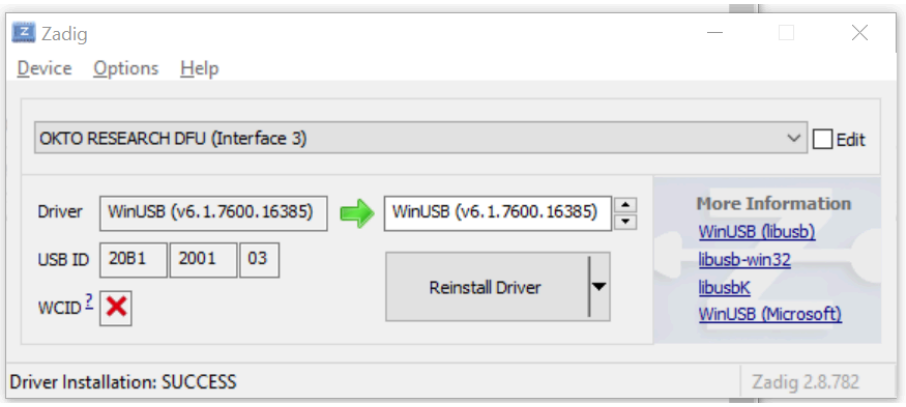
The utilities are compiled and tested for Linux Mint cinnamon x86_64.
xmosusb utility requires sudo prefix unless the user name is added in the proper usb plugdev group.
Libusb library must be installed separately and visible in the “Path”.
example : sudo apt-get install libusb-1.0-0-dev

Copying or moving dspcreate and xmosusb to /usr/local/bin/ gives the possibility to launch them from any working folder.
A version is provided for Raspberry Pi os compiled for aarch64 (V6.1.21-v8+).
Finally a version is also provided for the Raspberry Pi ARM 32 bits os and this was tested with a Volumio 3 image (v3.661).
As for Mac OSX, no special limitations are known for using the xmosusb tool with the USB audio driver simultaneously.

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 DAC8 USB composite device. This has to be installed with Zadig. (using version 2.8 in this example below).
But this process is conflicting with the OktoResearch USB driver (Thesycon manufacturer) which captures all the usb interfaces. So the OktoResearch usb driver MUST be uninstalled before installing winusb with Zadig. Remark : *This will change in a later version where the windows WCID will be enabled and the Thesycon driver will be changed to free the winusb integration.*



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
```

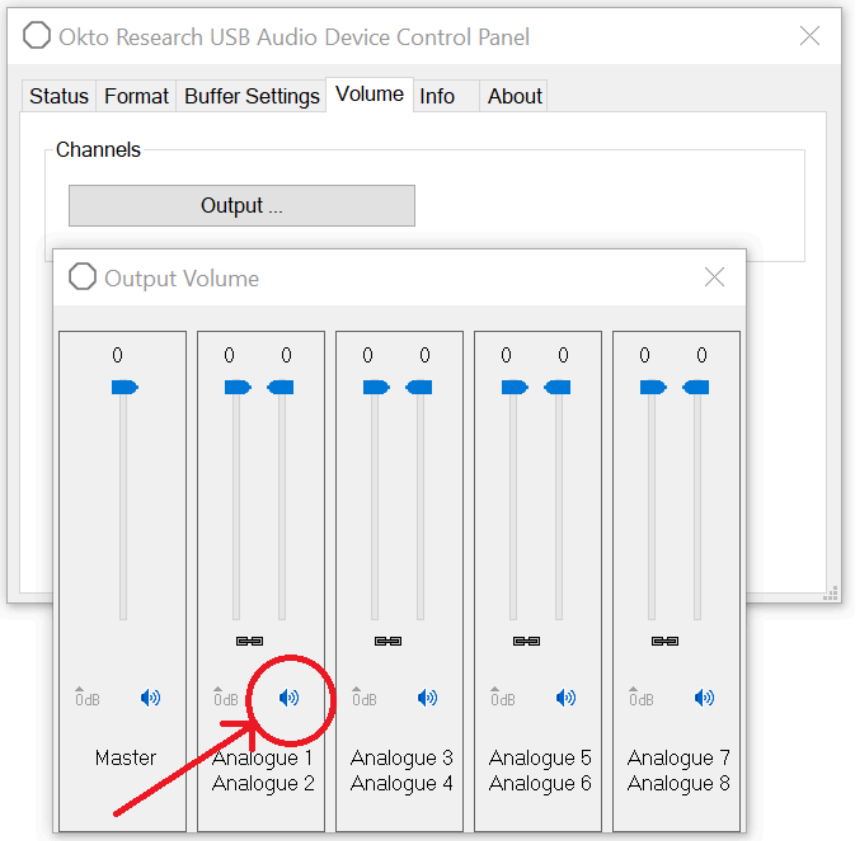
By default the Windows usb audio driver will be installed and will recognize DAC8 products without any issue, but it supports only 2 channels output (playing) and input (recording). This is practically enough to test the DSP program, channel by channel with REW in wasapi mode.

Once the DSP programs are developed, tested, stabilized and written to the DAC8 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.

Windows special USB VID:PID

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 DAC8 DSP 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.

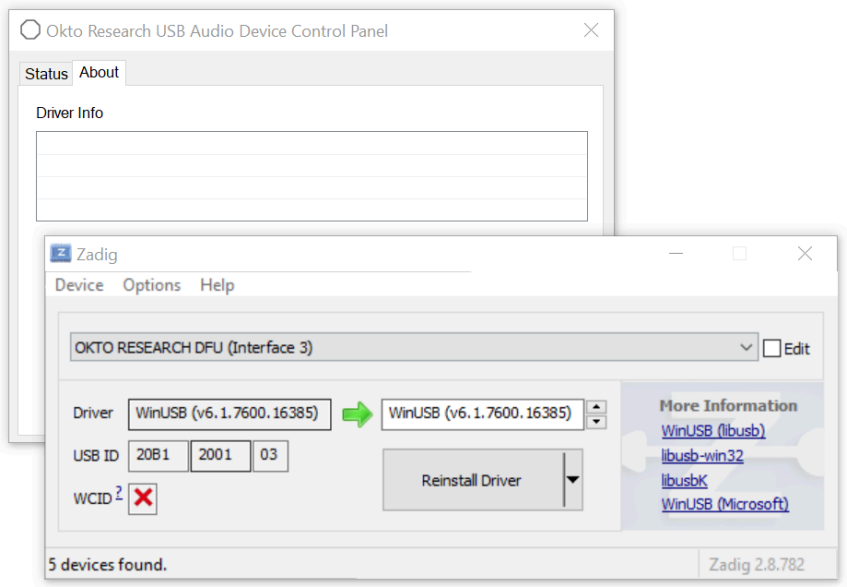
To reboot the DAC8 DSP with this temporary VID:PID, go to the OktoResearch driver control panel, select the Volume tab and click the “output” area to display the volume's vertical cursors.



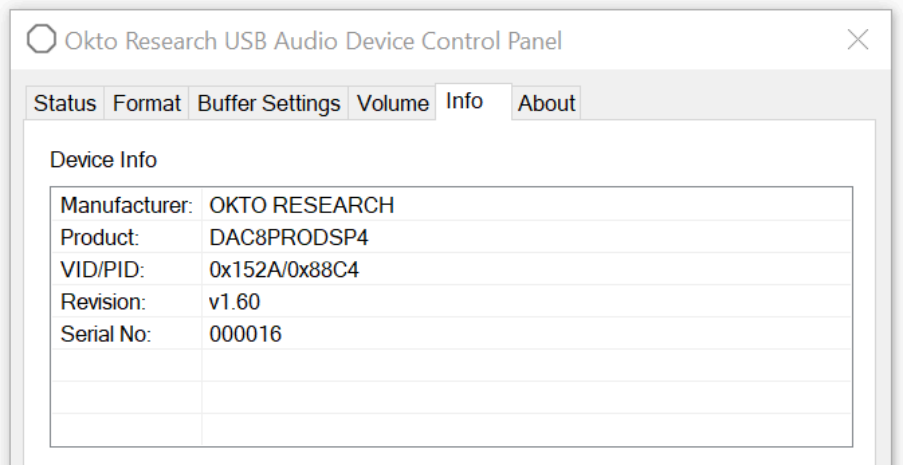
Click on the muting icon for the 2 first channels to mute them, and this will start a 5 seconds period where the USB cable can be disconnected : when reconnecting the usb cable, the dac will re-enumerate with the temporary VID:PID, so it will not be seen by OktoResearch usb audio driver, and windows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

remark : If the cable wasn't removed within the 5 seconds period then the DAC8 DSP keeps running with its original VID:PID and this special sequence is canceled. It is possible to restart this process at any time. The DAC8 DSP 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 DAC8 DSP 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 !

Utilities description

xmosusb utility

The xmosusb utility is an extension of the original DFU utility provided by XMOS. It has been modified 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 DAC8 DSP firmware. Note that each command starts with two successives "-" (minus) characters, for history reasons…:

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

--dspwrite num : Writes the xu216 ram memory to the flash location "num". This is the way to save a DSP program into one of the 4 permanent flash locations within DAC8 DSP.

--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 one.

--flashread num <N>: Provides a dump of one or N (optional) pages of 64 bytes of internal flash memory as of page "num". DSP programs are written as of page 0 with 2048 boundaries.

--flasherase sector <N>: Erase one or N (optional) sectors of 4096 bytes in flash memory at a given sector position. DSP programs are stored in flash starting at sector 0 with 32 sectors boundaries. To erase DSP program 3 in flash, just type `xmosusb --flasherase 64`. Remark: original factory firmware is stored in a specific boot partition and is fully protected from erasure.

--dspheader : Provides a summary of the DSP program currently loaded in xu216 ram memory.
--dspstatus : Provides CPU load of each xu216 core.

--xmosload file : This command can be used to flash an xu216 firmware compatible with DAC8 products, with the same result as using the DFU Windows tool, for linux or Mac OSX or Raspberry pi platforms. If the provided firmware contains a front panel firmware then it will 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 a bad file would brick the unit. This command is perfectly safe to use as long as the file provided is compatible with the DAC8 product.

--samdload file : Sends a binary file to the xu216 cpu to flash the front panel firmware. No checks are done on the binary file. This command should be utilized only when instructed, to flash a specific version of the front panel firmware without changing the current version of the XMOS firmware. The flash process takes 60 seconds without almost no 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 if the front panel was overwritten with a specific custom or OEM version using a previous `--samdload` command.

Remark : if the xmosusb utility seems blocked in an endless waiting loop, just press Ctrl-c to exit…

If the problem remains, the xmos usb stack may be stuck in an unforeseen loop : reboot the DAC8PRO by powercycling the physical mains connector on the rear panel… Version 1.62 now implements a 1 second watchdog on the USB traffic so these situations should not happen.

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 is available on github fabriceo/avdsp.

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

-dsptext filename<+filename2…> : Process one or many text files that can be chained with the "+" character (but no space character allowed).
The embedded compiler recognizes a simplified syntax for label and filter definition and a 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 defined, code size and checksum.
remark, this selector -dsptext can be omitted when the filename contains .avdsp or .avd

-binfile filename : instructs to save the list of opcodes as a binary "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 dac8stereo it is possible to use -fsmax 384000
remark: these default values can now be overloaded inside the .avdsp program with keywords DSPFSMIN and DSPFSMAX. Also the -fsmin -fsmax option selector can be omitted.

-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 integers with a mantissa of 28 bits and an integer part limited to 3 useful bits (-8.0..+7.999). Any value between 16 and 30 can be used to generate a 32 bits integer with a customized mantissa between 16 and 30 bits.
for DAC8 products, this option can be omitted as the default value is now -dspformat 2

-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 xmos firmware.

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

-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 DAC8 DSP but not documented here.

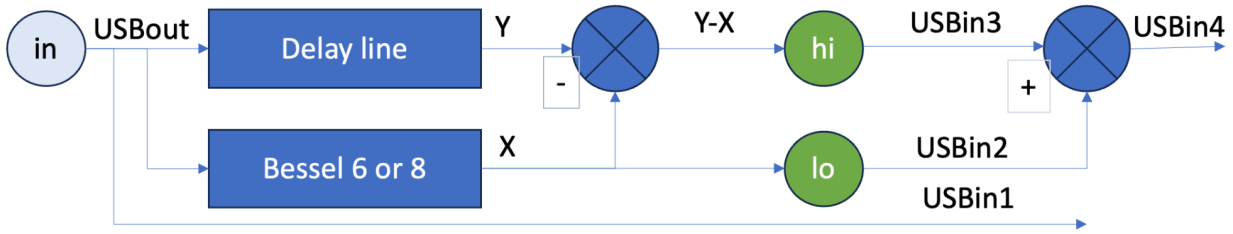
when using -dsptext selector or an .avdsp source file, all remaining information at the end of the dspcreate command line will be used as additional lines of the DSP program given, and they will be passed one by one to the analyser and before starting analyzing the main text file. This is useful to pass 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 examples for DAC8 DSP:

```
dspcreate -dsptext example.txt -binfile prog1.bin FC=400 GAIN=-3db
dspcreate example.avdsp -binfile prog2.bin FC=500 GAIN=-2db
dspcreate -dsptext dac8pro1+example.txt -binfile prog3.bin
coef=0.6
dspcreate test.avdsp -binfile t.bin -dspload G=2db F=1000
```

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     #question mark gives the possibility to overwrite fc in command line
bessel8         LPBE8(fc)       #simple bessel filter order 8 at “fc”
gd      =       986000/fc       #empiric formula for getting group delay for a bessel 8 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          #exchange X and Y, to save X and getback input value
        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
```

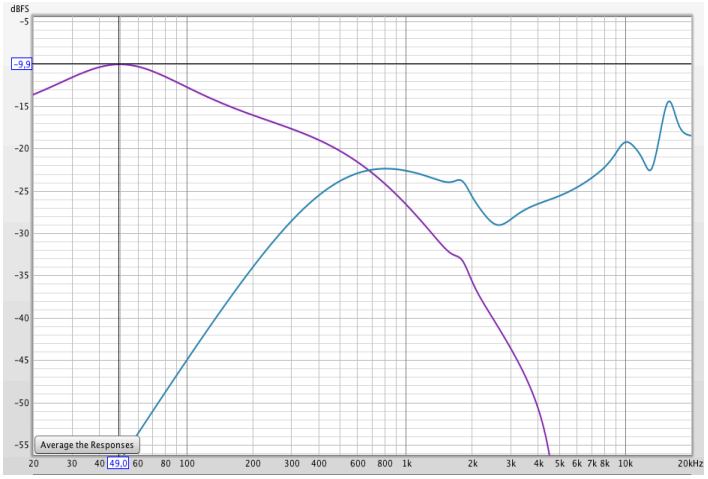
example decay convolution (...0.125, 0.25, 0.5)

```
sum     MEMORY 1
coef    60%
core
        input 16 ; output 24    #feedback to usb host for signal reference
        gain coef ; copyxy
        loadxmem sum ; gain 1-coef ; addxy ; savexmem sum
        output 25               #sent to usb host second channel for sweep measurement
end
```

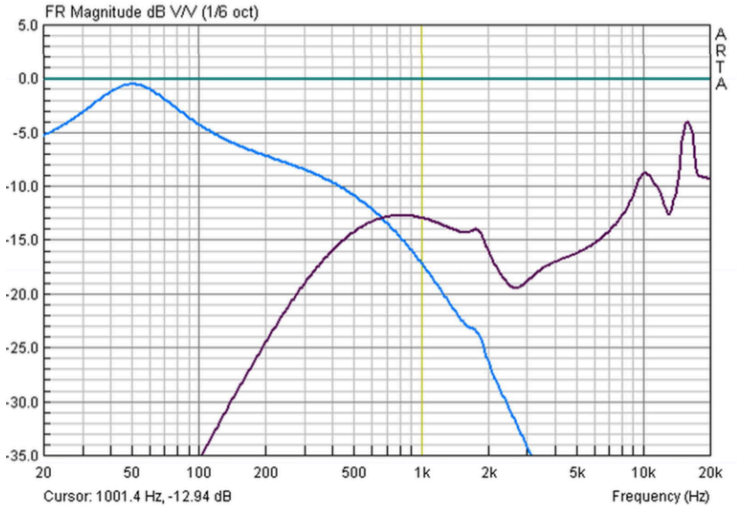
example LXMini crossover 2 ways

```
ineq    FILTER PEAK(1800,7,2db)
loeq    FILTER LP2(700,0.5) PEAK(50,0.5,7db) PEAK(5100,8,-16db) PEAK(5800,3,-10db)
hieq    FILTER HP2(700,0.5,-1) 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



Managing saturation. instruction and example

The following program can be tested with the third party tools REW with “Scope” view, by testing 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   FILTER 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            #retrieve accumulator Y
      gain -3db         #accumulator is reduced between +0.8..-0.8
      output 26         #clean as biquad routine provides 18db headroom
      copyyx            #retrieve accumulator Y
      saturate          # the result is reduced by up to 5db automatically
      output 27
end
```

When a `saturate` instruction detects an overflow situation it raises a global saturation flag and immediately clamps the sample between +1..-1. Then at the next sample cycle, the DSP runtime will check this flag and eventually increments a saturation counter register.
If a `saturate` instruction sees a value N in this counter register, it reduces the value of the accumulator by N decibels. So the previous hard clipping happens 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 on the front panel or simply moving the volume knob.
The saturation is displayed black on white in the top right corner of the volume display area.

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 or white noise and reviewing each output with the “Scope” view.

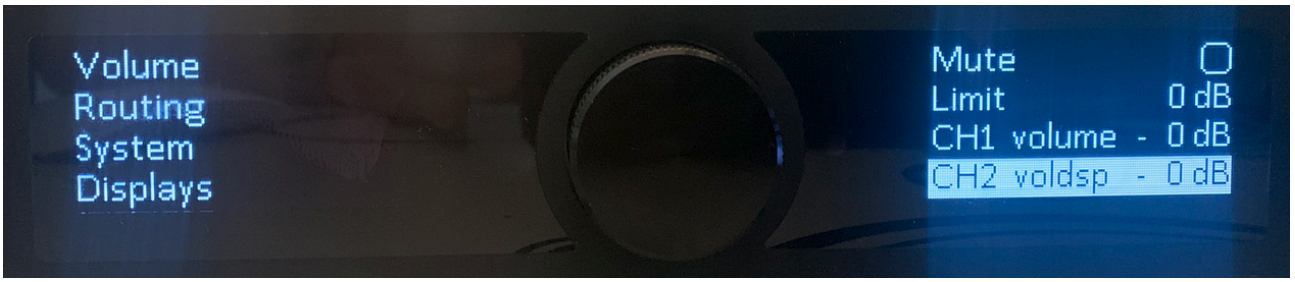
Using `saturate` instruction before any `output` requires additional cpu time but provides a safe solution to avoid any harmonics on outputs linked to potential overflow. This approach of reducing dynamically the output gain by steps of 1db is a unique way to avoid traditional signal compressors, which inherently bring a lot of distortion when the signal comes close to the peak threshold. Once all the gain chains have been adjusted to mimize clipping, the `saturate` instructions can be removed safely from the final DSP program, or replaced by `saturatevol`.

By extension, the `saturategain` instruction executes a gain function and then the explained `saturate` behavior.

Combining volume and saturation

Some musical programs can show an excess of up to 3db (or even more) due to their encoding, or when they pass through audio filters (or through the DAC chip FIR or ASRC itself!). Also any IIR filter (like biquad) with a Q factor above 0.5 will generate overshoot and will produce output above original input (just try playing a square wave through a butterworth filter to get convince).
The 64 bits treatment with the default 8 bits headroom implemented in the DAC8DSP during computation will accommodate this situation perfectly, until the result is stored in memory for the DAC as these 8 upper bits have to be removed to keep the signal between +1..-1.
Using the original `saturate` instruction is bullet proof to detect these situations, but it will reduce the Dynamic Range of the DAC by simply always removing these excess decibels once they are detected. But this reduction is not really needed if we apply the volume as part of the DSP treatment instead of doing it in the DAC. As an example, if a given music program going through a given filter is generating say 9db of excess, this is not a problem as long as you always listen to a volume between -99 and -9dB. and if you listen sometime at a volume of -7db, the `saturate` instruction should just remove 2db instead of 9db, which allows maximum dynamic range.

In order to enable this possibility, the DAC8PRO front panel program has been upgraded to eventually delegate the first 18db of volume treatment to the XMOS DSP. Then the Sabre DAC keeps attenuating the residual part when the volume is between -99 and -18db. This feature can be enabled for any DAC channel, with a long press of the knob when editing a channel volume.



For the channels where this feature is enabled (channel 2 here on this picture), the DSP program is expected to take care of the first 18db of volume reduction and this requires using explicitly one of the 3 new instructions in the treatment chain for these channels.

The new `saturatevol` instruction provides the same behavior as `saturate` but integrates the volume reduction before detecting the signal clipping, and should be used before any `output` instruction when the corresponding channel has been configured as “voldsp”.
By extension and to optimize the CPU treatment it is possible to replace any `output` instruction by `outputvolsat` which combines both `saturatevol` and `output` functionality in a single instruction.

By extension, `outputvol` applies up to 18db of reduction without testing nor adding saturation. This gives the possibility of using a DSP program having excess gain without taking care of the saturation, considering that most of the time the volume will be much lower than the excess.
The `saturate`, `saturatevol` or `outputvolsat` provide 1db granularity and will detect and reduce volume by a maximum of 15db.
remark: *`output`, `outputvol`, `outputvolsat`, `outputtpdf` instructions have been optimized to allow storing the accumulator value in multiple memory locations, listed on the same line, separated with coma.*

Dithering with 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 declared 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 computed at each sample rate cycle and the noise value computed can then 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 single core. for example:

```
core
      tpdf 20
      output 29             #provides the randomized value for tests
      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         #18 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.

remark :
The resulting noise added to the signal is identical to the one generated by the third party software REW, when applying dithering in the frequency generator. This can be seen by looking at RTA/FFT with either REW generating the dither or using outputtpdf instruction.

The tpdf instruction will load the X accumulator with the random value computed and can be provided on any output for test purpose. Its rms value is -4.8dbFS.

The white instruction provides the original random value, before tpdf compute. Its rms value is -1.5dbFS.

Using conditional `core` and `section`

The AVDSP catalog of opcodes 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 (or more) 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	44/48k	0001	PureAES1	0001	PureUSB
0011	5	0010	88/96k	0010	PureAES2	0010	PureUSB2
0111	6	0100	176/192k	0100	PureAES3	0100	AES/USB
1111	7	1000	352/384k	1000	PureAES4	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 0x300,7` or
`core 0b00110000000,0b0111`
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 (or core) also marks the end of section
```

Creating a program for different Mode (PureUSB, USB/AES PureAES)

Understanding In/outs locations in DSP memory

In PureUSB or USB/AES, the audio stream from the USB Host is provided in the IO locations 16..23.
In PureAES and USB/AES, the stream from AES channels is stored in the IO locations 8..15.
In USB/AES, it is possible to use either the usb stream (16..23), or the AES stream (8..15) directly with the `input` or `mixer` or `transfer` instructions.

As of version 1.62, in USB/AES or PureAES mode, the IO locations 8..15 are duplicated in locations 24..31 to give visibility on the sound presence and to provide a default stream to the USB host when it is connected (with the same sampling rate).
In PureUSB, the locations 16-17 will be copied to 24-25 by default to provide a basic loopback reference for the USB host which is typically useful with tools like REW.
In all cases, this mapping can be overlapped by the user program and the user can generate its own value for the usb host in locations 24..31 with usual `output` instruction.

In addition, in PureAES, the Spdif/AES stream is also duplicated in locations 16..23 so that they can be retrieved from the same location as the one provided by the USB host in PureUSB mode.

So as an example, `inputxy 16,17` will load the stereo samples Left and Right respectively in accumulators X and Y, either from the USB Host when the mode is PureUSB, or from the AES connector 1 when the mode is PureAES.

Impact on the display:

In PureUSB and USB/AES, the left display indicates the 1..8 channels which are effectively streamed by the USB Host; they are surrounded by a square when the host player starts streaming. The numbers are inverted (black on white background) when a sound presence is detected (not 0).

In PureAES, the behaviour is different : the 1..8 channels are surrounded when an AES/SPdif signal is recognized on the respective AES inputs (not only an electrical connection but a real biphase mark stream). This process of displaying Lock and Sound presence is done by the processor by reading value in memory location 24..31

Optimized fixed point 64 bits math explained

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 samples provided by the AES inputs are 16 or 24 bits only. They are all represented as 32 bits signed data in XMOS 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 format 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

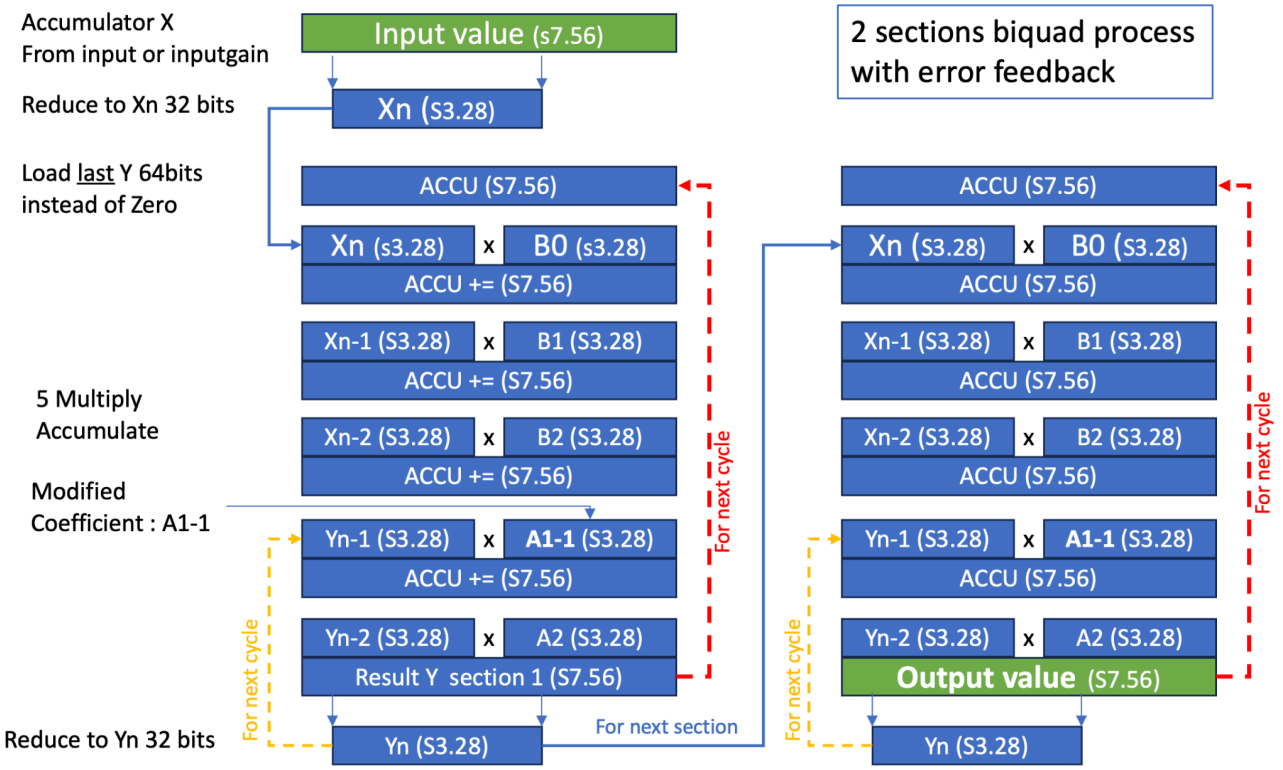
An output instructions use specific cpu instructions to quickly saturate and reduce the s7.56 accumulator to a 32bits s0.31 value which is stored in the output location.

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 64 bits regular s7.56 format.

An outputgain instruction also combines three multiply instructions and then reduces the 96bits result to get 32 useful bits (q31 format) 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 useful 64 bits 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 I): 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 32 bits 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 64 bits 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
and already implemented by Robert Bristow-Johnson in
https://dsp.stackexchange.com/questions/21792/best-implementation-of-a-real-time-fixed-point-iir-filter-with-constant-coeffic



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 floats (24 bits mantissa+7 bits exponents).

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, highshelf and lowshelf filters require larger coefficients and their gain is 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.

DSPMANT a,b

In firmware as of version V1.62, the instruction DSPMANT can be used at the beginning of the DSP program to modify the behaviour of the XMOS fixed point ALU. The first parameter defines the mantissa for the biquads coefficient and for any gain values. The second parameter defines the mantissa in the 64 bits accumulator (X and Y). The default values explained in previous chapter correspond to the default DSPMANT 28,56

As an example, to increase the biquad coefficient span from default +18db to +24db, just use DSPMANT 27 which will increase by 1 bit the size of the integer part. This also modifies the encoding for gain values when used in instructions like outputgain or gain, then the instruction will accept values between -16.0…+16.0, or up to +24db.
By Opposite, if this is not needed and if +6db is acceptable instead of +18db, just use DSPMANT 30 which will bring 2 bits of extra precision on the coefficients and in all computations.

The default headroom in the accumulator is +42db, using 1 sign bit and 7 bits for the integer part, the remaining 56 bits being for the decimal part called mantissa.
If this is not needed, it is possible to reduce this by increasing the mantissa.
As an example, +30db might be more than enough, requiring 5 bits for the integer part. Just use DSPMANT 28,58 (or DSPMANT 30,58 if combining with above discussion).

DSPCLOCK a

or DSPCLOCK a,b,c,d
This instruction gives the possibility to overclock the XMOS processor with value “a” between 480mhz and 648mhz maximum (+30%). The value must be a multiple of 4.
The default value set as of firmware v1.61 and for v1.62 is 528.
Increasing by 20% to 600mhz or even more should be tested extensively in the end user environment (including temperature) before concluding to feasibility and reliability....
Remark : the behaviour of the XMOS processor when the frequency is too much is not defined. It can be expected that the CPU would enter in some fatal exception, blocking it until a complete power cycle is made by removing the mains plug. There is no watch-dog in XMOS XU216.

The Values “b” and “c” can be set to tweak the frequency used when the DAC8PRO is set in USB/AES or PureAES mode. As the AES inputs 2,3,4 are decoded in software, this might be useful to accommodate difficult AES front-ends or long connections. Default value “b” is 496mhz and is used when frequencies are 44/88/176k, while default value “c” is 528mhz, used when the frequencies are 48/96/192k. Parameter “d” will configure the AES decoders with or without CPU priority and in fast mode or not according to following values :
0 (default)= prio OFF, fast OFF. 1= prio ON, fast OFF. 2= prio OFF, fast ON. 3= prio ON, fast ON.
Priority ON means that the XMOS processor gives as much mips as needed for this core (with the limit of 1/5th of the frequency) whatever the number of cores are running.
Fast ON, means that when the core is in a waiting mode, it still consumes mips in order to anticipate a restart when the pending condition is released.
Changing this mode from 0 to other 1…3 will create timing changes on all DSP cores (excluding the first one in charge of I2S-USB-DAC) as soon as more than 5 cores are used in total.

Special function instructions to measure available instructions

The XMOS processor is able to provide 106mips (528mhz / 5) for the 5 first cores.
if 6,7 or 8 cores are running then it will distribute up to 106 mips to the prioritized cores (like the first one in charge of I2S-USB-DAC) and then the remaining mips will be spread evenly to the others cores.

A special function called instructions is introduced in the DSP language in order to estimate how much cpu instructions are available in a core (including overclocking). As an example

```
core
      instructions
end
```

The function `instructions` just enter in an infinite loop incrementing a counter, until the next audio sample is ready. Then it returns the number of instructions that were processed in the meantime.
The result is seen with the usual `xmosusb --dspstatus` command. This number accurately represents the number of CPU instructions available between 2 audio samples, and this should be the limit for the user program. If the user program is longer and uses more instructions, then the DAC8 will reduce the sampling rate by 2 and will restart the DSP program in “decimation” mode.

If a numerical value is given as a parameter for instructions then it doesn't count but instead will accurately consume the number of instructions requested. This is a way to force loading a core in order to verify its capacity. In example below, the number returned is same as above minus 200:

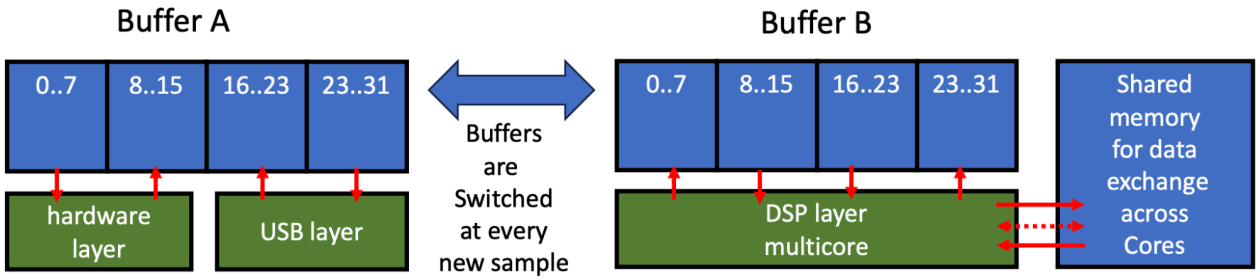
```
core
      instructions 100 ; instructions 100 ; instructions
end
```

With a parameter, `instructions` can be used anywhere in a DSP program.
Without any parameter (or value 0) it will end the core without continuing any potential instructions listed after it.

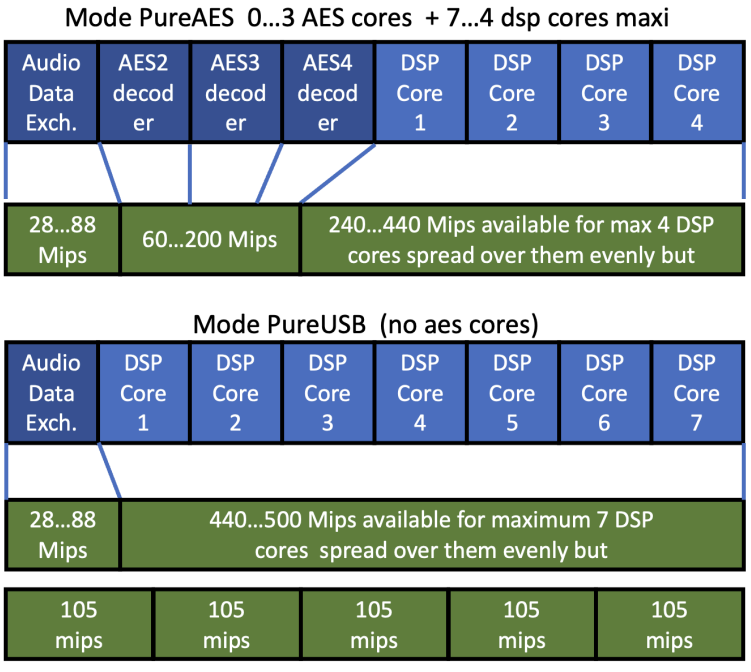
A detailed table of information is given in this document to show the available mips depending on the context. This has been measured accurately with this function `instructions` .

Memory and Cores, technical information

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



The XMOS provides 8 cpu cores on its Tile 0 to execute audio tasks. The first one is always dedicated to handling internal data transfer between USB and DACs or AES and USB and has full priority. It uses between 28 and 88 mips depending on the sampling rate (44k1…192k).
On the DAC8PRO, using the AES2,3,4 for SPDIF content (like for 5.1 or 7.1 configurations) requires activating extra cores manually in the DSP program, with the instruction `coreextern 1,` eventually followed by enabling conditions as for a standard `core` instruction. They are not prioritized and consume between 20 and 70 mips each, depending on the sampling rate.



The remaining cores available (7,6,5 or 4) can be used in the user DSP program.
The XMOS hardware scheduler will always provide the required mips to the first audio task and all the remaining mips will be distributed evenly to all the other cores (aes and DSP) with a maximum limit of 105 mips per core (528 mhz / 5).

The `xmosusb --dspstatus` command can be used to read the DSP core load in realtime and to adjust or spread the program to maximize and harmonize mips across cores.

Remark : at 176k/192k, aes cores requires that no more than 7 cores out of 8 are used, otherwise the SPDIF decoder will not have enough mips to lock on the signal, or may generate errors…

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 (with parenthesis).

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 `--dspstatus` report. The provided example “maxload.txt" demonstrate that it is possible to realize the following with 4 cores:
192k, 8 dac outputs : 13 biquads and a delay line for each output .
96k, 8 dac outputs : 26 biquads and a delay line for each output.
48k, 8 dac and 8 aes channels : 26 biquads and a delay line for each input and each 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.

When DAC8PRO is configured in 96khz (compared to max 192k), 13% additional MIPS are available and it is possible to use them by enabling a fifth core (or 6 or 7) and to spread the load across the total cores. At 48k, 20% additional MIPS are available.

Downsampling/decimation

Because of the CPU performance 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 48000 hz 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. This is not enough to totally remove aliasing and this can generate distortion in the high frequencies.

Support for DAC8 DSP

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

information for developers

The AVDSP repository on github contains 2 branches, master and develop.
The master branch is the original source code providing compatibility for linux and an early version for xmos.
At that time (May 1st 2020, release 1.0) the dspcreate utility did not include the macro interpreter with the -dsptext selector.
The develop branch has modified a lot of code to provide better optimization for xmos and to integrate the -dsptext selector capabilities. Unfortunately there is still a lot of work to do to merge the 2 branches. Users willing to test the new DAC8 style DSP programs on linux with the AVDSP alsa plugin will have to wait for a merge of the 2 versions…

The xmos runtime is now written in assembly and not yet published on the AVDSP develop branch. instead it is embedded in the xmos usb application specifically customized for DAC8 products. Contact the author to get a version of this runtime and some advice for its integration in a custom product. The plan is to remove the specific xmos XS2 instructions from the AVDSP standard runtime and to upgrade the C runtime with the latest opcodes.

DAC8 Firmwares release notes:

Version 162

XMOS firmware

Rewrite task scheduler to optimize XMOS core allocation and only when required.
Rewrite core start/stop, overload detection, timing measurement.
Rewrite opcode dispatcher to optimize response time “everywhere”.
Optimize spdif/aes software decoder for less sensitivity to jitter and to better synchronize them with hardware spdif/aes receiver, including situation with decimation.
Optimize biquad routine (16 instructions per filter instead of 19, + overhead)
Rewrite spdif/aes tasks to allocate cores dynamically (with coreextern instruction).
Introduce linear filtering : moving-average, convolution and warped convolution.
Unique firmware version, not dependent any more on static aes/spdif requirements.
New keyword to modify compute precision or coefficient span (DSPMANT).
New keyword to modify cpu clock for overclocking up to +20% (DSPCLOCK).
New keyword to enable full scale output (DSPSERIAL) otherwise -24dB applied.
Possibility to tune spdif/aes core sampling frequency at 44/88/176k or 48/96/192k.
Multiple OR conditions are now possible for enabling cores and sections.
Dynamic memory allocation for cores, very useful when using conditional cores.
Some new instructions opcodes (sine, thdcomp, integrator, reverb…)
New instructions working on y or xy accumulator (biquadxy, gainxy…)
Set of 17 new instructions to load, save, add… directly from/to memory location.
Augment total memory available for code and delay lines (46kwords total).
Double IO sample array and allocate value 32-33 specifically for AES Output.
Loopback done by default on the IO sample array for 16..17-24..25 (PureUSB), 8..15-16..23 (PureAES), 8..15-24..31 (USB/AES and PureUSB).
Immediately reset saturation indicator when volume is changed on the front panel.
Watchdog 1 second timeout on each USB transaction to avoid USB stack stall.
Reading dspstatus done seamlessly without sound interruption.
Timing optimization in the main I2S core to free extra MIPS for DSP cores.
Changing multi-core priority model and providing new instructions priorityon and priorityoff to be used (carefully) as first instruction of DSP cores.
New keyword `instructions` to simulate full CPU core load to measure remaining instructions before the next sample, or to simulate an exact number of instructions.

Front panel firmware

no change in front panel, same as v161, just showing new version 1.62 instead

dspcreate utility

Syntax highlight template for visual studio code. very convenient to read big programs.
Possibility to define impulse for convolution (TAPS keyword), with included files.
New include instruction to break down source files. 4 levels supported.
Numerical expressions now support parenthesis 4 levels.
Decimal numbers can be followed by “%”, dividing them by 100

Decimal numbers can be followed by “m”, dividing them by 1000
Better management of multiple instructions on a same line with “;”
Possibility to display a comment at compilation time using prefix “#-”
Possibility to display variable value at compilation time in comments using [myvar]
Introduce `if,else,endif` keywords for compiling only some pieces/lines of code.
New keyword sectionelse, with or without conditions to facilitate conditional split and enable reusing memory allocated across section and sectionelse (typically delay lines).
Simplified command line options :

-dsptext can be omitted when source filename ends with .avdsp or .avd
-dspformat 2 is default value and can be omitted or overloaded with DSPMANT.
-minfreq -maxfreq can be omitted by using DSPFSMIN and DSPFSMAX in source.

Keyword DSPPRINTF x can be used to set the level of verbosity x from 0 to 4.
Keyword DSPXS2 can be used to inform that the target binary file is for XMOS XS2, to display an estimation of XMOS instructions used per cores at runtime.
Keyword DSPCOND x set a condition for the XS2 instruction estimator, to simulate DAC8 mode like PureUSB or PureAES. default 0 will consider all cores and sections enabled.
Keyword DSPLINE can be used as a predefined label representing the current line.
New operator “?” can be used after a label name to verify if the label is defined.
New option selector `-dspload` will launch *xmosusb --dspload file.bin* just after code generation to the provided *-binfile file.bin*
Dynamic library libavdspencoder now included statically in a single executable file.

xmosusb utility

--dspstatus prints better core instructions accuracy, and remaining memory and saturation.
--dspload also prints the information from --dspstatus after successful upload.

known issues or limitations

ES9028 DAC filters do not remove all aliasing terms produced by potential decimation.

Version 161

Based on the official V160.
Front panel upgraded to enable “voldsp” (or “bal+dsp” for stereo) when setting channel volume and applying a long press on the volume knob.
Some display glitch corrected (input selected and selection).
xmos dsp code upgraded to optimize multiple outputs in single instruction.
instruction outputgaintpdf removed.
introduction of digital volume attenuation (max18db in this version) with either outputvol or outputvolsat or saturatevol instructions.
introduction of saturatevol and outputvolsat to detect up to 15db of saturation after volume reduction. Saturation granularity now by 1db instead of 6db.
Correction of a critical bug when delayus and delaydpus were used with value zero.
Extension of all available RAM, providing up to 38kwords for dsp program and data.
Dspcreate utility upgraded, new opcodes generated and introduction of backslash “\” to provide flexibility on defining filters by extending definition on next line. Due to optimizations and new instructions each dsp program requires recompilation with dspcreate utility.

xmosusb utility upgraded to better support xmosload and to print dsp program size available in dacstatus.

known bugs:
dspcreate might interpret hexadecimal numbers starting with x as labels in some cases.
delayus and delaydpus accept max 1000 microseconds

Previous versions

v160_1, v160_2, v160_3, v160_3b : preliminary distribution with DSP code enabled

Révisions & history

DATE	REV	Changes
August 25th, 2025	162	New DSP functions supported by firmware v1.62, new keywords DSPMANT, DSPCLOCK, DSPXS2, DSPSERIAL…, new scheduler, VSCode highlighter.
August 4th, 2024	161	Introduce saturatevol, outputvol and outputvolsat and 18dB volume spread across DAC and DSP. dspcreate supports backslash \ to continue syntax on next line. DSP memory extended to 38kwords to accommodate long delay lines (up to 0.18 seconds in total ~ 60 meters, at 192khz available for all channels). saturatetpdf and saturategaintpdf removed.
May 18,2024	160_2	Removing DAC8STEREODSP4 and adding technical information on core mips allocation. changes in core and section conditions.
May 7,2024	160_1	Adding headlines and table of content, and remarks about sample rate limitations for DAC8PRODSP4
May 5, 2024	160_0	first release, information compatible with DAC8 DSP firmware V1.60

Master document and latest version in gdrive (limited access):
https://docs.google.com/document/d/13nhfehX73qjWsr-vC3V8dsLwfl4rJwBqnygRnYlMKMI/edit?usp=drive_link

List of PDF document on github:
https://github.com/fabriceo/AVDSP_DAC8/tree/main/documents