DGN-1 Digital Guitar Effects Processor

Tomasz Kaczmarczyk tomasz.kaczmarczyk@teamovercrest.org
Tomasz Henisz tomasz.henisz@teamovercrest.org
Dominik Stożek dominik.stozek@teamovercrest.org

Revision 1 Feature overview and basic technical documentation

Abstract:
DGN-1 is an electric guitar effect chain implemented on an Altera DE2 prototyping board. It chains together six different, fully configurable digital effects implemented entirely in hardware. The complete system delay is kept well under one sample, resulting in a zero-delay audio path and allowing true real-time sound processing capabilities. The implemented effects are parameterized and can be easily customized or bypassed by the user with the use of a rich graphical animated interface presented below. The software also boasts an audio analyzing module rendering a dual-channel oscilloscope-like transfer function plot and VU output power meters.

Fig 1 User Interface layout

Table of Contents

System overview

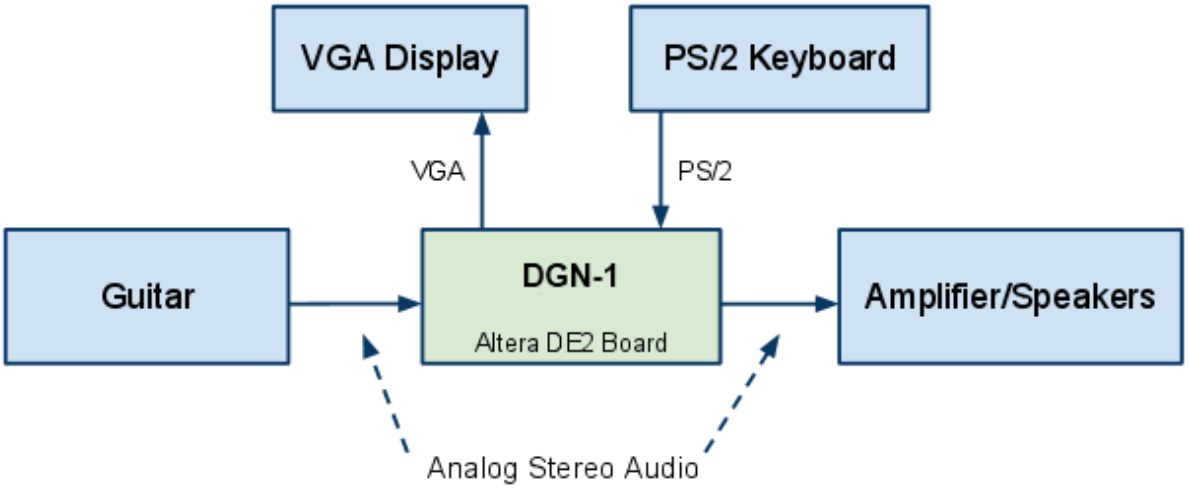


Fig 2 DGN-1 typical application in an audio chain system

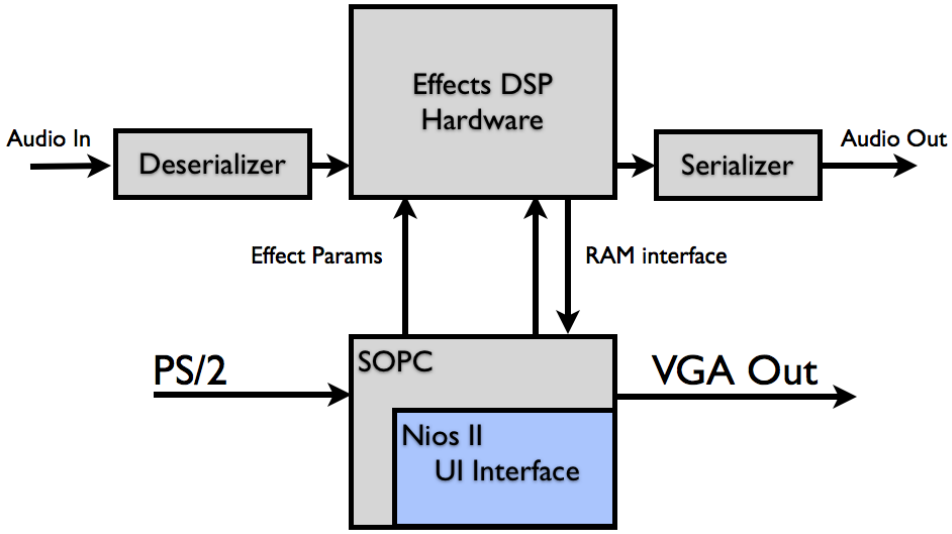


Fig 3 DGN-1 high-level component architecture

The DGN-1 guitar effects processor takes analog input from a guitar or a similar device as an input. It is then sampled at 48 kHz and converted to a digital signal which is processed by a hardware audio path consisting of 6 distinct guitar effects.

User interface

DGN-1 uses a custom animated graphical interface to allow for easy parameterization of the audio path components. The audio components are represented as the artistic impressions of front panels of imagined audio-rack elements. The interface also provides the user with a clear visual feedback of the characteristics of the system. Output signal power is indicated by a set of animated VU meters whereas the current transfer characteristic of the setup is calculated and displayed in real time by a software sound analyzer.

Fig 4 UI live shot

The user may change the properties of the system by selecting any of the available 21 animated knobs and buttons and adjusting its value to suit his needs. The user navigates the interface by using the arrow keys of a standard PS/2 keyboard. A pleasant animated cursor in the form of a blue wiggling arrow indicates the currently selected parameter. Stylized digital displays and positions of knobs, switches and buttons depicture the current values of different effect parameters.

Rack elements

„Blues Rider" distortion

The distortion effect creates a non-linear response of the system. High-amplitude signals have less gain than lower-amplitude signals. The effect is instantaneous, so the shape of the signal gets deformed (*distorted*). New harmonics are added to the sound, creating a more interesting guitar timbre. The effect resembles the sound of an overdriven tube or transistor, hence the second name of the effect: **overdrive**.

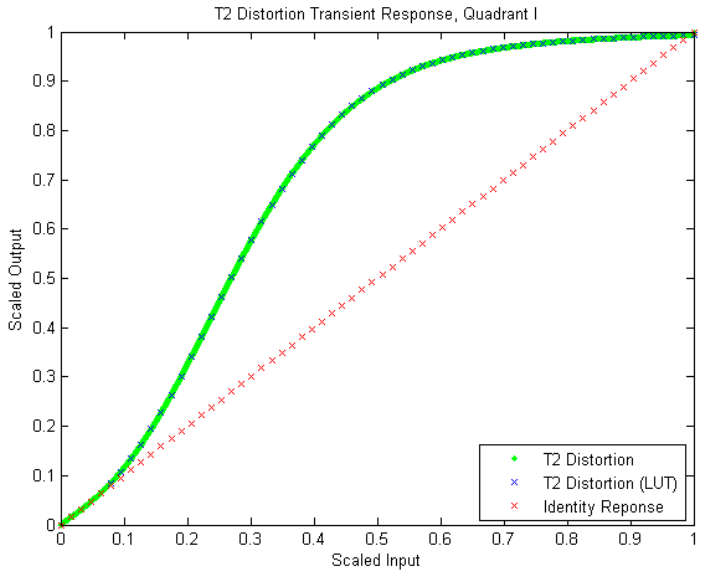


Fig 5 Simulated distortion transfer characteristic

The distortion effect can be customized using the following parameters:

- **Asym** (asymmetric) creates additional distortion by halving the gain of negative half-periods of the signal.
- **Drive** regulates the input volume of the effect, changing the working point of the non-linear characteristic.
- **Volume** sets the effect's output volume.
- **Tone** determines how much of the signal is mixed through a lowpass tone filter, which muffles the sound but eliminates unwanted chirp.

"Waveringer" panning tremolo

The tremolo effect introduces a trembling variation in the amplitude of the output signal. The gain of the internal amplifiers changes cyclically up and down resulting in a slowly oscillating output volume of the signal. It is also possible to set counter-phase trembling of the left and right channels, which causes one channel to be amplified while the other is attenuated, broadening the perceived audio source.

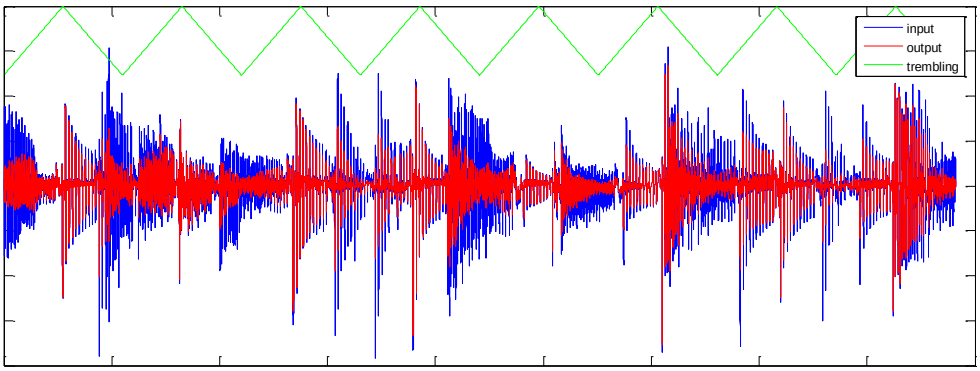


Fig 6 Tremolo – simulated waveform

The tremolo effect can be customized using the following parameters:

- **Depth** controls the maximum amplification and attenuation, effectively controlling the influence of the tremolo on the signal
- **Rate** controls the frequency of the trembling, changing whether the variation in output signal power happens rapidly or gradually
- **Join** controls whether the two channels should be affected counter-phase or together, resulting in either a stereo panner or a standard tremolo

"Frosch" compressor

The compressor effect is a subtle self-adjusting gain controller. It uses a pseudo-RMS averaging algorithm to establish the recent input signal power and uses this information to dynamically attenuate loud inputs while leaving the lower volume ones undistorted.

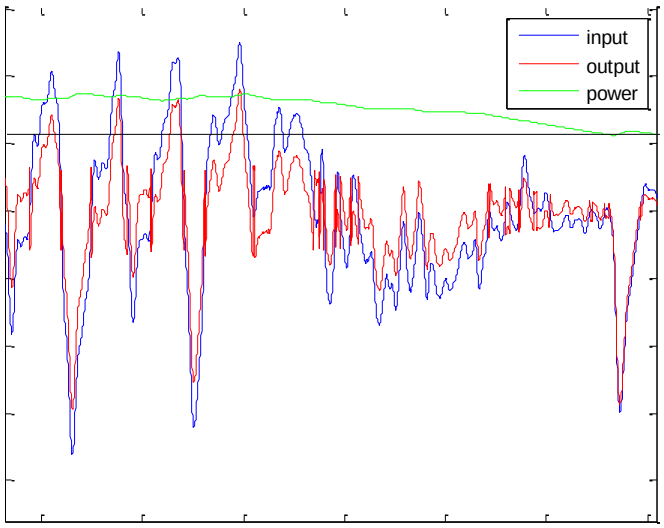
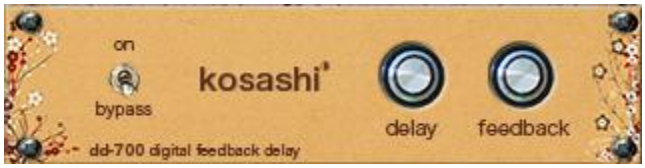


Fig 7 Compressor waveform – the horizontal black line is the threshold level

The compressor effect can be customized using the following parameters:

- **Gain** controls the amplification of the signals passing through the compressor
- **Threshold** controls the signal power level at which the compression kicks in and the compressor starts attenuating the input

"Kosashi" delay



The delay effect buffers a fragment of the audio signal in memory and mixes it back into the output signal. The effect's output is fed back into the delay unit, creating a decaying, infinite response.

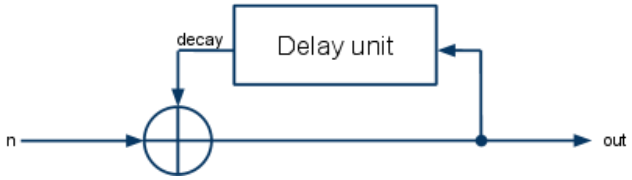


Fig 8 Feedback delay effect - signal diagram

The feedback effect can be customized using the following parameters:

- **Delay** specifies the time delay of the effect
- **Feedback** determines the gain of the signal coming out of delay unit

"Octavarium" octaver

The octaver effect halves the frequency of the input, resulting in a sound which is shifted by an octave from the original signal. Frequency halving is achieved by approximating the incoming signal with a square wave expanded properly to achieve the required frequency and feeding it through a low-pass filter to smoothen the resultant waveform. The effect is a distinct, deep metallic sound.

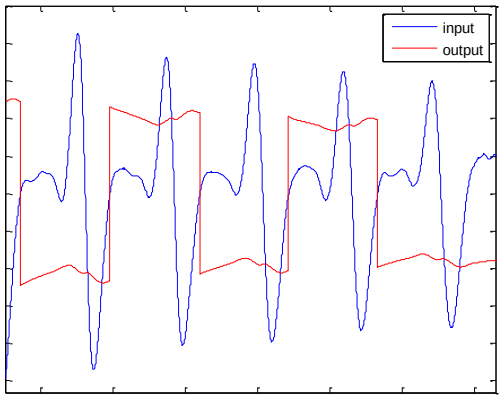


Fig 9 Octaver simulation waveform

The octaver effect can be customized using the following parameters:

- **Mix** controls the amount of the transformed signal which is fed into the output. Maximum value means that only the transformed signal is sent, minimum means that the output is completely undistorted.

"TU-101 Tupolev" bitcrusher

The experimental bitcrusher effect alters the volume resolution of the signal, making it seem as if the sound was synthesized on an 8-bit game console. Currently, two flavors – algorithms of degenerating the signal – are implemented. One performs simple bit shifting and shuffling, discarding some signal bits and realigning others. The other discards a number of lower bits of the signal, adaptatively changing the number of discarded bits to match the average power of the input signal.

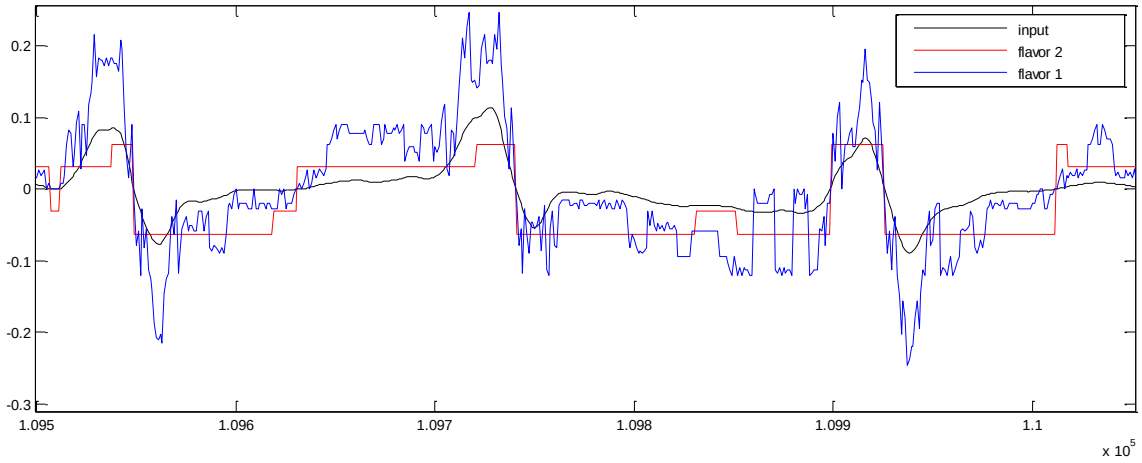


Fig 10 Bitcrusher simulation waveforms

The bitcrusher effect can be customized using the following parameters:

- **DryWet** controls the amount of the transformed signal which is fed into the output. Maximum value means that only the transformed signal is sent, minimum means that the output is completely undistorted.
- **Tone** determines how much of the signal is mixed through a lowpass tone filter, which muffles the sound but eliminates unwanted chirp.
- **Flavor** decides which algorithm is used to degenerate the input signal.

"Elizabeth II" output amplifier

The amplifier takes care of adjusting the output power level to suit the needs of the user. It also gives him graphical feedback in the form of the two animated VU meters showing the current volume of the signal.

"Puppetech Vetch-Pro" transfer characteristic analyzer

The analyzer module samples the input and output signals and plots the resultant transfer characteristic in real time, much like a typical two-channel oscilloscope. The curves and loops visible on the screen can then serve as a basis for judging the characteristics of the currently setup system, intuitively indicating gain various nonlinearities and distortions. Functional keys on the keyboard serve as scaling switches, making it possible to zoom in or zoom out plot axes independently of each other.

Contents of the package

Main project

The package contains the complete, annotated and commented hardware/software project:

- */Project/*
  contains the Quartus II hardware project. All the diagrams are annotated and VHDL source codes contain comments explaining the algorithms and the implementation details. DGN-1.bdf diagram is a good entry point when starting to explore the project.
- */Project/software/*
  contains the NIOS II EDS software projects. The projects should be imported into a new NIOS II EDS workspace.

Auxiliary files

Some additional files are also included in the package:

- */Auxiliary/Archived/*
  contains the compressed versions of the hardware and software projects.
- */Auxiliary/Binaries/*
  contains the compiled project files. You can use the .sof and the .elf files to directly program the development board without recompiling everything. The next section explains how to do it.
- */Auxiliary/Graphics/*
  contains the raw binary picture data to be uploaded to the FLASH memory on the DE2 board. The memory offsets at which they should be stored and their source images are also provided.
- */Auxiliary/Simulation/*
  contains Mathworks Matlab scripts simulating some of the effects used in the project.

Deployment

Required software

- **Altera Quartus II** (preferably version 9.1 or later)
  http://www.altera.com/education/univ/software/quartus2/unv-quartus2.html
- **Altera NIOS II Development Suite** (preferably version 9.1 or later)
  https://www.altera.com/download/software/nios-ii
- **Altera DE2 Control Panel**
  (available on the CD provided with the board)

- Altera University Program Design Suite 9.0
  http://university.altera.com/materials/tools/upds/?quartusII=9.0&hdl=VHDL&board=DE2
  (*to resolve the issue with the installer claiming that Quartus is not installed if you're using a newer version of Quartus, set an environment variable SOPC_KIT_NIOS2 to the path to your Nios II EDS installation folder e.g. C:\Altera\10.0\nios2eds, then run the installer*)

Required hardware

- Altera DE2 Board
- D-sub capable VGA screen (or a multimedia projector)
- PS/2 Keyboard (or a USB keyboard with an USB/PS2 adapter)
- Speakers/headphones
- Guitar (or your different sound source of choice)

Deployment procedure

Before first starting the program, FLASH memory on the DE2 board needs to be pre-populated with the graphics used by DGN-1. It's a onetime process and you won't need to repeat it later (unless the images are changed). The procedure is as follows:

0. Pre-populate FLASH
   a. Use Quartus II Programmer Tool to upload the Altera DE2 Control Panel .sof file
   b. Start DE2 Control Panel software and connect to the board
   c. Erase the FLASH chip using the "Chip Erase" command
   d. Upload the .rgb565 files you'll find in the */Auxiliary/Graphics* folder
      i. Check the "File Length" box
      ii. Input the offset found in *offsets.xls* into the "Address" box
      iii. Write the corresponding file to FLASH
      iv. Repeat until you run out of .rgb565 files to upload

DGN-1 can then be deployed to the development board either by opening the projects in appropriate Altera software (Quartus II and NIOS II EDS) or directly from NIOS II EDS command line. The command line approach is described in NIOS II Software Developer Handbook (chapter I.3) http://www.altera.com/literature/lit-nio2.jsp and can be used to deploy the compiled binaries available in */Auxiliary/Binaries/* folder.

To deploy DGN-1 using Quartus II and NIOS II EDS the procedure is as follows:

1. Start Quartus II and use it to open the hardware project
   a. Use the Programmer Tool to deploy the .sof file to the development board. Keep the dialog box about using time limited megafunctions open while running the project, otherwise you won't be able to upload the .elf software file or use the UI.
2. Start NIOS II EDS and create a new workspace
   a. Import the software projects contained in the */Project/Software/* folder.
   b. Compile the projects (you might need to regenerate the BSP – right click the GuitarrHjalteUI_BSP project in the Project Explorer window and select "Nios II > Generate BSP". Consult the final section of this document if you need to do this.)

   c. After the compilation is successful and the hardware has been deployed you can run the software (select "Run As > Nios II Hardware". You can choose to ignore system ID and timestamps if necessary).
3. Set the hardware flip switches as described in the next section of this document
   a. Switch 0 – UP
   b. Switch 1 – DOWN
   c. Switch 16 - UP
   d. Switch 17- UP

Using DGN-1

Hardware switches

DGN-1 uses a number of hardware flip switches to control a few deeper functions which didn't find their way into the GUI.

- Switch 0 – Audio Enable
  During normal operation should be in the UP position. The switch can be used to quickly mute all sound output if necessary.
- Switch 1 – Master Bypass
  During normal operation should be in the DOWN position. Switching it to the UP position temporarily bypasses all the effects. It can be used to quickly compare pure, vanilla guitar signal with whatever you have configured in the effect chain.
- Switch 16 – Mono Select
  If the source audio signal is stereo should be in the DOWN position. If you're using a mono source, switching it to the UP position will cause the right channel to be copied from the left.
- Switch 17 – Reset
  During normal operation should be in the UP position. A quick flip DOWN and UP again resets the system.

Keyboard input

The currently selected parameter is indicated on the UI by the wiggling blue animated cursor. You can change the selection by pressing the arrow keys. The value of the selected parameter can be changed by pressing the 'a' and 'z' keys.

Other than that, you can just plug in your guitar and start riffing away.

Exploring and modifying the project

The sources of DGN-1 are thoroughly commented and the hardware schematics are decorated with colorful annotations. We hope that it will make the experience of exploring the project a pleasant and educative one.

Exploring

The hardware project can be opened by pointing Quartus II to the DGN-1.qpf file in the */Project/* folder of the package. The software projects can be opened by importing the GuitarrHjalteUI projects you'll find in the */Project/software/* folder into a new NIOS II EDS workspace.

Hardware

Start out by getting yourself acquainted with the hardware architecture of DGN-1. All the diagrams and VHDL codes are thoroughly annotated and commented so this document will only provide a brief overview of the most notable features of the hardware. All the details are best explained inside the project itself. Fire up Quartus II, open the project and head on to DGN-1.bdf schematics.

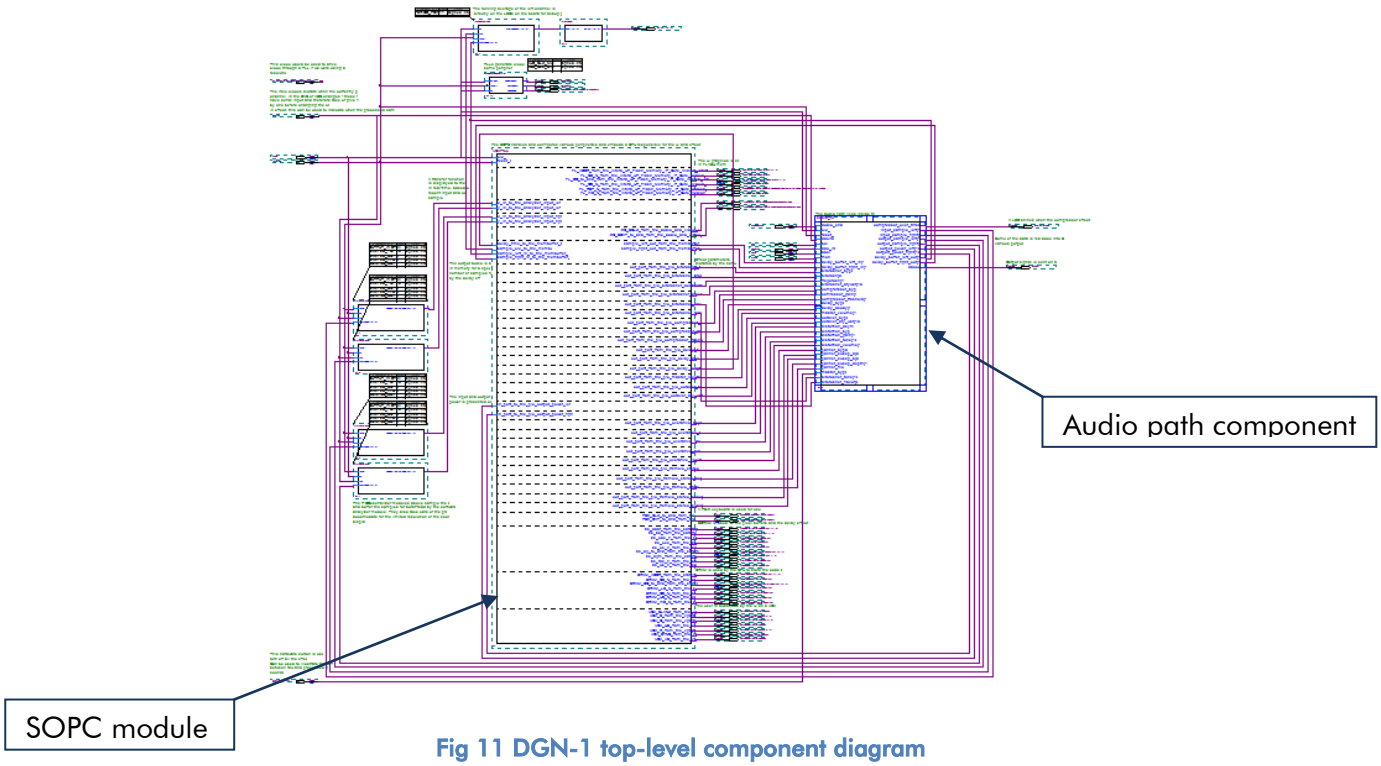


Fig 11 DGN-1 top-level component diagram

The diagram is dominated by an SOPC module which takes care of creating the NIOS II CPU (on which the software is run), handling all the DE2 peripherals used by the system and feeding effect parameters (which are configured by the user) to the hardware via PIOs (parallel input/outputs).

You will also notice a few external modules which take care of buffering data for the sound analyzer module and configuring the PLL (phase locked loop) to drive some peripheral clocks. The annotations and comments in the project explain it in more detail.

The audio path component houses all the effects and the input and output handling modules. It consumes all the inputs concerning sound processing - the ADC (analog-to-digital converter) signals, effect parameter values and hardware switch inputs (like the master bypass or mono selector). It outputs data to be fed to DAC (digital-to-analog converter) as well as some signals which are fed back to the SOPC (either to display on the sound analyzer module, VU meters or to store in a memory buffer).

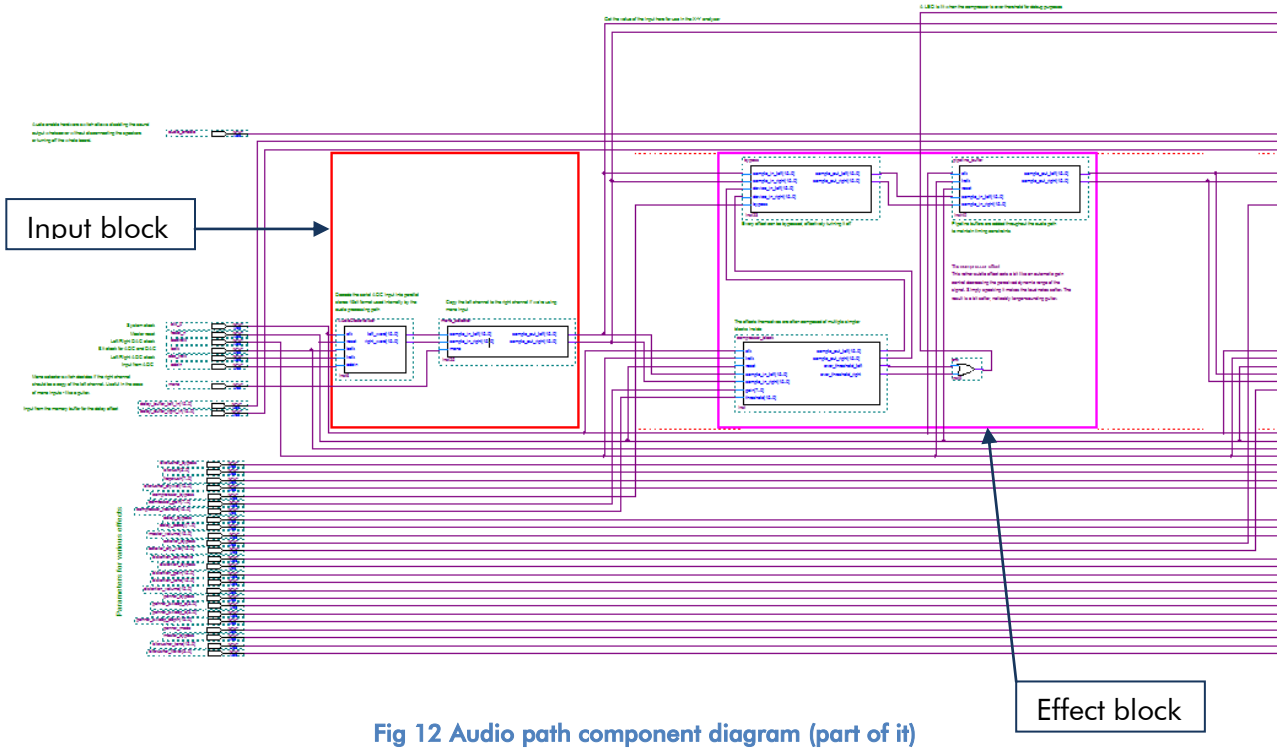


Fig 12 Audio path component diagram (part of it)

Inside it you will find blocks representing the input/output blocks and the six implemented effects. The I/O blocks take care of translating the signal between a serial form understood by ADC/DAC and the parallel 16 bit stereo format used internally inside the audio path. The I/O blocks are outlined in red, the effect blocks are purple. Everything is annotated thoroughly.

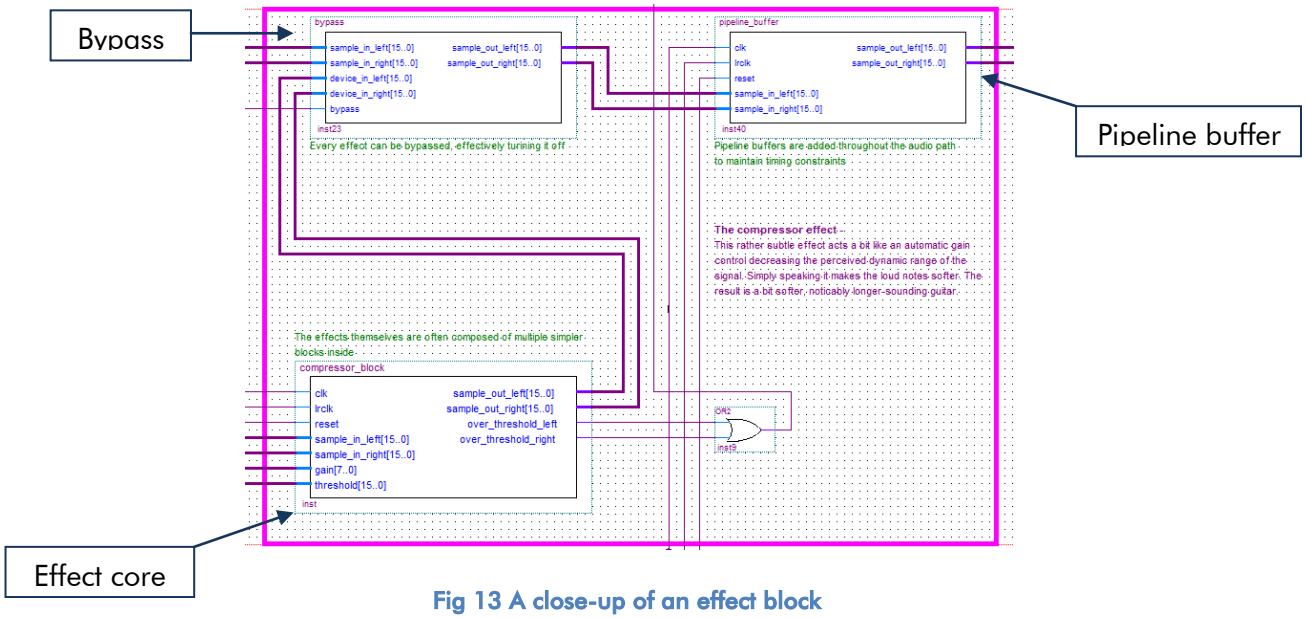


Fig 13 A close-up of an effect block

All the effect blocks have a similar structure. They are in most cases composed of three characteristic modules.

Bypass acts as a select switch which can be used to ignore the signal fed by the effect core, effectively excluding the effect from the effect chain.

A pipeline buffer is introduced after each effect to take care of timing violations. In some of the more complex effect cores additional pipeline buffers are introduced between sub-stages of the effect. All these buffers are however clocked with the 50 MHz system clock whereas the sound is sampled at 48 kHz. This means that roughly 1000 of those would need to be introduced into the system to cause a system latency of over one sample.

The effect core performs the actual signal transformation. The implementation of every core is different, some are monolithic VHDL modules and some are composed of smaller reusable blocks inside.

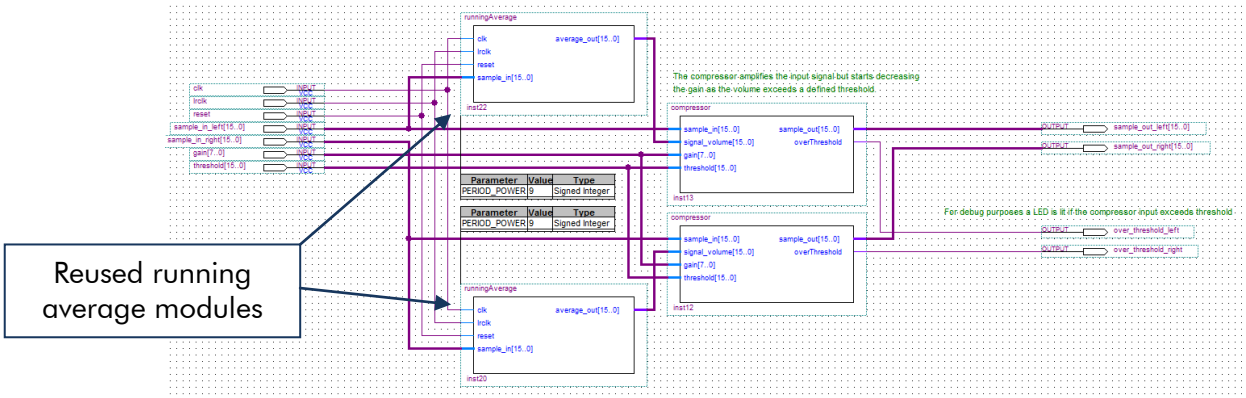


Fig 14 Internal structure of the compressor core

Software

The software behind DGN-1 takes care of displaying the UI and handling all the effect parameters. The program is divided into well defined modules taking care of particular features. The source code is richly decorated with comments which explain all the implementation details. Therefore this document only provides a brief outline of the structure of the software.

- UserInterface
  is the starting point of the software. It contains the main function which initializes all the other modules and runs the main program loop. The loop periodically signals the other modules to update their state and perform their respective jobs.
- Time
  keeps a global time counter. In the current implementation the time module just naively counts the number of iterations of the main program loop. If a need for higher time resolution would arise, it could be easily rewritten to poll a hardware counter instead, but in the current setup it is entirely unnecessary.
- Input
  handles the PS/2 keyboard. It processes all the input from the user as he presses the keys and takes care of properly debouncing the input. It indicates a key-press event to other modules first when the user actually presses the button and then repeatedly at decreasing intervals if he keeps the button pressed. This means that the user gets both high input accuracy (by pressing and releasing the key quickly he only triggers a single key-press event) and high rate of change when necessary (by keeping the key pressed he starts triggering the event multiple times, faster and faster).
- Effects
  handles interactions with the effect parameters. It takes care of incrementing/decrementing the currently selected parameter, sending the parameter values to the PIOs and deciding which parameter to select next when the user navigates the UI.
- EffectViews
  takes care of rendering the various widgets representing the parameters. The effects are rendered as text fragments, knobs or switches. The knobs are rendered as simple revolving dials, indicating the current value of the parameter. The buttons are rendered by blitting a graphics representing their current state (on or off) onto an appropriate position in the screen.
- CreateEffects
  configures all the effect parameters and widgets. It contains the definitions of the expected behavior of the parameter (maximum, minimum values, destination PIO, optional custom handlers for changing the value and feeding it to the hardware) and the type and position of the widget used to represent the parameter on the GUI.
- TextShards
  wraps around the Altera UP Character Buffer IP Core and is used to render text-based widgets on the screen.
- Graphics
  wraps around the Altera UP Pixel Buffer IP Core. It contains optimized functions for blitting regions of memory onto the screen (with support for rudimentary transparency), drawing arbitrary lines and additively blending pixels onto the background.

- Graphs
  handles the VU meters and the sound analyzer plots. The transfer function on the sound analyzer plot is generated in real-time based on a number of last input and output samples. To increase the fidelity of the plot, the sound analyzer module is fed 8 pairs of samples per update. It then blends pixels corresponding to the sample pairs onto the GUI and takes care of fading the older pixels.
- Cursor
  renders the animated wiggling cursor. The cursor is re-blitted onto the background every frame and makes use of the transparency implemented in the Graphics module.

Modifying

This section should be understood not only as a guide for introducing changes into the project but primarily as a description of the process which was involved in creating the DGN-1. Even if you're not planning on introducing your own changes to the project, the following sections may give you a clearer understanding of the internal workings of this system.

Required IP Cores

The project uses SOPC IP Cores available as part of the Altera University Program Design Suite. The complete bundle can be downloaded here:
http://university.altera.com/materials/comp_org/ip_cores/?hdl=VHDL&board=DE2&quartusII=9.0

Modifying the effects

Modifying or substituting the existing effects with your own ideas is a relatively simple procedure. It might however prove to be a bit time-consuming due to rather long hardware project recompilation process.

Prototyping

To cut down on the number of recompilations necessary to achieve the desired effect, it proves worthwhile to spend some time simulating and testing the effect thoroughly. The simulations for the effects present in the default setup of DGN-1 were conducted in Mathworks Matlab environment. All the scripts used for this purpose are available in the */Auxiliary/Simulation/* folder. The simulation scripts load a prerecorded guitar sample, feed it through the tested algorithm, plot the resultant waveforms and play the transformed sound back for comparison.

This approach makes it a lot easier to spot the flaws in the algorithm you're using to implement your new effect. You can work out the details of the implementation or introduce some filtering and verify that the effect you're designing will sound the way you want it to.

If you decide to include a filter in the effect, the excellent Matlab 'fdatool' command can be used to design it, check how it'll affect the sound and eventually save it as a ready-to-use VHDL module. We suggest using IIR filters to cut down on resource utilization.

After you're satisfied with the simulation results you can fire up Quartus II and replace one of the existing effects with your own implementation by rewriting the algorithm you prepared in VHDL or Verilog.

Introducing new parameters

If you decide to parameterize your effect, you need to perform some modifications throughout all the layers of the project – the hardware, the software and the UI. This is possibly the most tedious part of introducing a new effect to the system but if carried out with care it shouldn't take more than half an hour.

In the current, naïve implementation the parameters are handled by the software and communicated to the hardware via PIOs. Therefore, to quickly introduce new parameters you may want to hijack a set of PIOs from an existing effect. This is a reasonable temporary solution if you're eager to hear your effect and don't feel like being bothered with creating your own PIOs at this stage.

However, the proper way to introduce new parameters is to create new dedicated PIOs and wiring them up appropriately. To do this you will need to do the following:

1. Double-click the SOPC module to open the SOPC Builder
    a. Create the output PIOs you need (you can copy existing parameter PIOs if you find it helpful). Pay attention to set the proper width of the PIO (it effectively decides the range of the numbers you can feed through the PIO. 16 bits is a good default).
    b. Rebuild the SOPC (after the rebuild is complete you may need to refer to the final section of this document to fix a recurring double buffering issue caused by faulty Pixel Buffer IP Core).
2. Exit the SOPC Builder and sort out the connections
    a. When you update the SOPC Module symbol, the output ports will most likely get messed up to accommodate the new PIOs.
    b. Disconnect all the messed up connections by removing a tiny segment of the wires right next to the SOPC module.
    c. Reconnect them to match the corresponding audio path component inputs. Pay special care to not overlay the wires one over the other since this will fuse the connections and introduce even more chaos.
3. Open the audio path component
    a. Add input ports corresponding to your new parameters
    b. Place the new ports below all already existing ports – this will prevent them from messing up the order of ports in the audio path symbol (the box representing the audio path component).
    c. Wire-up your new parameters.
    d. Select "File > Create/Update > Create Symbol Files" to update the symbol file.
4. Move back to the top-level schematics
    a. Right click on the audio path component and select "Update Symbol or Block"
    b. Connect the new parameter ports to the corresponding PIOs.

After you're done modifying the hardware, you may want to start the compilation before moving on to modify the software. Complete hardware compilation may take up to 20 minutes on an average-grade computer. Once the compilation finishes you may safely ignore any DDR timing verification errors. They result from a missing test configuration file.

Once you've performed these steps you can start preparing the UI and the software to accommodate your new parameters. The software modifications will be fairly straightforward in most cases.

1. Regenerate the BSP project
    a. Right click the GuitarrHjalteUI_BSP project and select "Nios II > Generate BSP"
    b. You may need to refer to the final section of this document to fix a recurring issue with the keyboard drivers from Altera UP PS/2 IP Core.
2. Open up the GHCreateEffects.c file
    a. The file is composed of several functions configuring the parameter widgets. All the functions are called at the end of the file during the initialization of DGN-1 software.
    b. Read through the comments you'll find around the first few widgets to get the idea of how to introduce your own parameters
    c. You may want to use some simple test widgets before updating the UI graphics and laying them out properly (just text-based widgets like in the bitcrusher effect placed somewhere on the screen will do well enough for testing purposes). This way you'll be able to test your effect before taking care of the details.

Modifying the UI

In the */Auxiliary/Graphics/Source Graphics/* folder you'll find the source-files of all the graphics used in the user interface. You can fire up your favorite graphics editor (Photoshop or Gimp) and create your own effect panel, complete with whatever gauges or buttons you can dream of.

Once you're done and satisfied with the outcome, save the UI as a 640x480 PNG. If you've introduced some new types of buttons or switches, also save the cropped graphics for their alternative switched state. Once you have the images saved, you need to convert them to a raw binary rgb565 format. What we used to perform the conversion was an RGB565 plugin for Paint.NET.

After successfully converting the images, you need to re-upload the graphics to FLASH. Follow the steps specified in the Deployment Procedure section of this document (remember to erase the flash chip before uploading new graphics). If you have any new button graphics, you should also upload them to some unused region of FLASH memory. Remember the offset at which it starts as it should then be used to populate the 'imageFlashMemoryOffset' field in the widget configuration in *GHCreateEffects.c*.

Known issues

Altera UP Pixel Buffer double buffering

The implementation of double buffering in the Pixel Buffer IP Core used in this project has a bug which causes it to sometimes feed the back buffer to the video output instead of the front buffer. This can cause flickering of whatever is drawn on the screen.

The glitch is reintroduced into the project every time the SOPC module is rebuilt and needs to be manually fixed afterwards. The procedure for fixing this bug is:

1. Open pixel_buffer.v file
2. Go to line 288
3. Change the condition of the if statement to
   *pixel_address_in == ((PIXELS_IN * LINES_IN) - 1)*
4. Verify that it's now the same as in line 275

The bug results from the fact that in the default implementation, the pixel buffer hardware actually swaps the buffers after finishing the frame at which the *buffer_swap* flag was raised but lowers the flag on the next beginning on a frame. This means that if the *buffer_swap* flag was raised after rendering the last pixel of the last frame but before rendering the first pixel of the current frame, the flag would be consumed without performing an actual buffer swap.

Altera UP PS/2 Keyboard handling

The drivers taking care of keyboard handling for the PS/2 IP Core have a bug causing them to fail at recognizing multi-byte make- and break-codes. This can cause the UI to seem broken and to not respond to the keyboard input as expected.

The bug is reintroduced every time you regenerate the BSP software project and needs to be manually fixed afterwards. The procedure for fixing the bug is:

1. Open altera_up_ps2_keyboard.c in the */drivers/src/* folder of the BSP project
2. Go to line 163
3. Change the function call from
   alt_up_ps2_read_data_byte
   to
   alt_up_ps2_read_data_byte_timeout

The bug results from the fact that in the original implementation the drivers tried reading the PS/2 FIFO to get all the bytes of a multi-byte code faster than the keyboard was writing to the FIFO. This caused the drivers to drop parsing the key-code after each byte because subsequent reads indicated empty FIFO, thus causing problems when parsing multi-byte codes.

Audio and Video Config SOPC Builder connection

In some cases, due to a version mismatch between the Altera UP IP Cores we used and the ones installed on your local machine, the Audio and Video Config component might be disconnected from other components in the SOPC Builder. The DGN-1 project itself will continue to function and compile as usually, however you will see an error message when you enter SOPC Builder in an attempt to modify the project.

The bug only needs to be fixed once. The procedure to fix it is:

1. Connect the audio_and_video_config_0 memory mapped slaved to the data_master bus from the CPU.
2. Assign a new base address to the slave (you can let the Builder do this for you by clicking System->Auto-assign Base Addresses).

Support and updates

This project, subsequent updates and news will be posted on dgn.teamovercrest.org. If you have any questions or run into issues with the project, please contact us at dgn@teamovercrest.org.