Pluto: using a sigma-delta audio output

G. Goavec-Mérou, J.-M Friedt

February 28, 2019

The investigation of the PlutoSDR hardware, combining an AD9363 radiofrequency frontend and a Zynq processor, aims at exploiting at best all functionalities of the platform. In https://www.github.com/oscimp/PlutoSDR/doc we focused the investigation on the Zynq by demonstrating the integration of GNU Radio on the processing system (PS) of the Zynq as provided by the buildroot framework, hence providing a flexible software environment for exploiting at best the embedded ARM processor. In the current investigation, we focus on the programmable logic (PL) and aim at demonstrating how to add functionalities to the FPGA without interfering with the existing AD9363 communication blocks. Interacting with these blocks will be the topic of another tutorial. In the context of a fully autonomous commercial broadcast FM radio receiver, we wish to add sound generation capability to the PL by integrating a sound card capability. The final demonstration will be data acquisition from the AD9363 using ADi's processing blocks, transfer to the PS in charge of FM demodulation and sound generation, communication with the kernel module emulating ALSA compatibility, and sending the resulting audio stream back to the PL embedding the sound card capability. Hence, this tutorial aims at extending the previous demonstration of processing the FM signal on the PS and streaming the resulting audio signal to the computer using the 0MQ framework by removing the need for the PC acting as a sound card and embedding this audio generation capability in the PL.

Since the PS has no sound controller, the PL design must be adapted to add the IPs needed to provided such a functionality. Furthermore, communicating to the IP in charge of sound generation through the ALSA framework requires updates to the PlutoSDR devicetree describing the hardware configuration.

1 Sound card implementation

Since no hardware codec is available on the PlutoSDR, we consider using two GPIO connected to the PL for generating the left and right audio signals.

1.1 Hardware

From a hardware perspective, two GPIOs will be toggled by the PL to generate PWM signals. The PWM to DAC conversion is taken care of by a RC low pass filter bridge connected to each GPIO output as shown in Fig. 2. The resulting voltage can be directly fed to headphones.

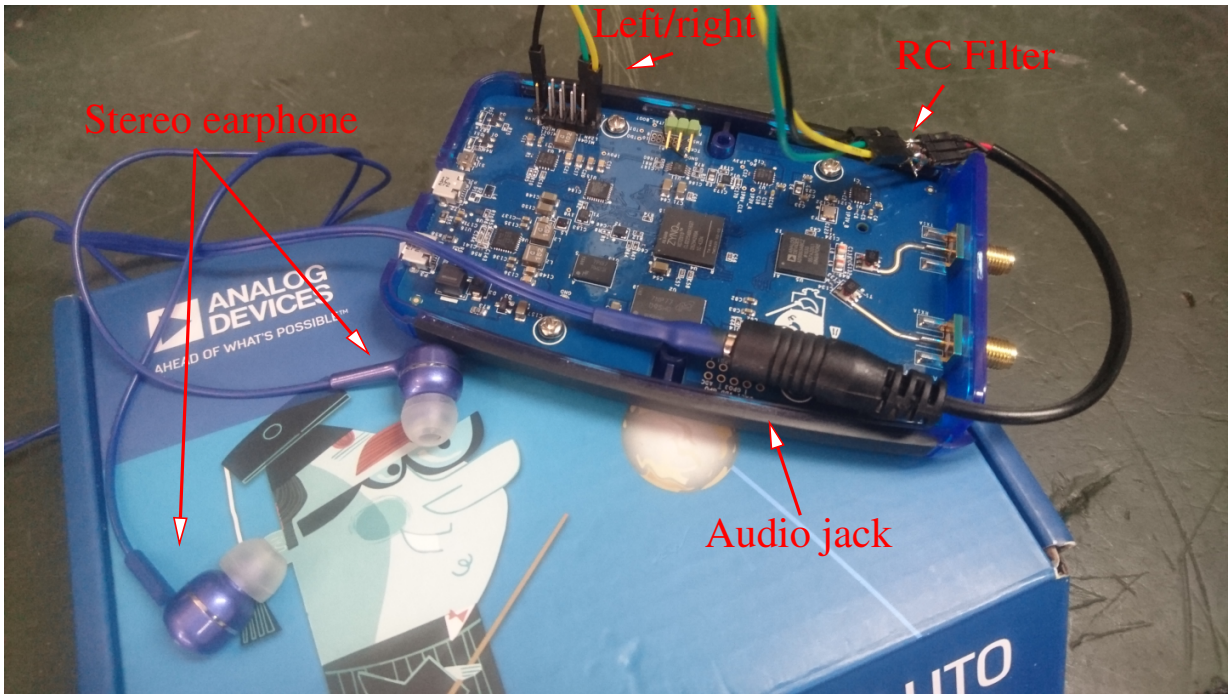

Figure 1: Objective of the demonstraiton: convert a PlutoSDR in a fully autonomous FM radio receiver providing a sound output of the radiofrequency signal demodulated by the PS using GNU Radio processing blocks.

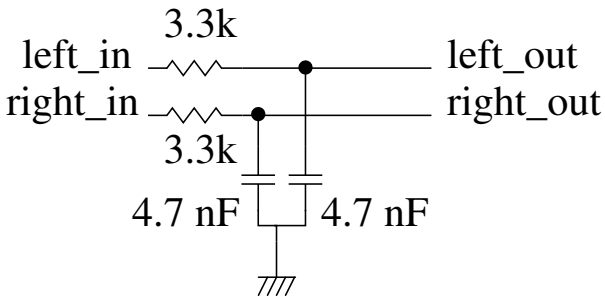

Figure 2: RC filter with a cutoff frequency of 10.26kHz for converting the $\Sigma\Delta$ output to a smooth audiofrequency signal driving the headphones.

1.2 HDL sound card implementation

The PL will be in charge of toggling the GPIO outputs so that they act as DACs. Such a result will be achieved by

- using FIFO based DMA memory to device transfer to collect the samples representing the sound signal

- generate a $\Sigma\Delta$ signal with a mean value equal to the resulting voltage
- configuring the clock signals to meet the timing requirements expected from standard sampling rate of sound cards (24, 48 or 96 kHz).

Such results are achieved by updating bitstream generated from the various IPs available on ADi's repository. Modifying the PlutoSDR bitstream provided by ADi requires first cloning their repository:

```
git clone https://github.com/analogdevicesinc/hdl # clone repository
git checkout 401395cdd1980827fd1f7043ce1a10770f666c64 # select hash of current official git
```

To generate the design there are two possibilities:

1. using the script and the `Makefile` provided in `design` directory. Such a path requires exporting the `ADI_HDL_DIR` variable to the correct repository location (see end of this section for pin mapping and bitstream generation);
2. adding mandatory IPs in an existing PlutoSDR project (`hdl/project/pluto`)

Only the latter case requires detailed instructions as given below. The file `system_bd.tcl` must be adapted, with:

```
variable fpga_ip $::env(OSCIMP_DIGITAL_IP)
# update ip_repo with IPs present in OSCIMP_DIGITAL_IP $
set_property ip_repo_paths [list ${fpga_ip} ${lib_dirs}] [current_project]
update_ip_catalog
```

included at the begining of `system_bd.tcl` to update the list of repositories known to Vivado, and

```
# audio
# axi_deltaSigma needs DMA0 ack/req
ad_ip_parameter sys_ps7 CONFIG.PCW_USE_DMA0 1

# left/right channels
create_bd_port -dir O bit_left_o
create_bd_port -dir O bit_right_o

# clk to generate base time
ad_ip_instance clk_wiz clk_wiz_0

ad_ip_parameter clk_wiz_0 CONFIG.USE_LOCKED false
ad_ip_parameter clk_wiz_0 CONFIG.USE_RESET true
ad_ip_parameter clk_wiz_0 CONFIG.CLKOUT1_REQUESTED_OUT_FREQ 24.576
ad_ip_parameter clk_wiz_0 CONFIG.USE_PHASE_ALIGNMENT false
ad_ip_parameter clk_wiz_0 CONFIG.RESET_TYPE ACTIVE_LOW
ad_ip_parameter clk_wiz_0 CONFIG.PRIM_SOURCE No_buffer

ad_connect clk_wiz_0/clk_in1 sys_ps7/FCLK_CLK0
ad_connect clk_wiz_0/resetn sys_rstgen/peripheral_aresetn

ad_ip_instance axi_deltaSigma axi_deltaSigma
ad_cpu_interconnect 0x43C00000 axi_deltaSigma

```

```
25 ad_connect clk_wiz_0/clk_out1 axi_deltaSigma/data_clk_i
26 ad_connect bit_right_o axi_deltaSigma/bit_right_o
27 ad_connect bit_left_o axi_deltaSigma/bit_left_o
28 ad_connect sys_ps7/FCLK_CLK0 axi_deltaSigma/DMA_REQ_TX_ACLK
29 ad_connect sys_ps7/FCLK_CLK0 sys_ps7/DMA0_ACLK
30 ad_connect sys_ps7/DMA0_REQ axi_deltaSigma/dma_req_tx
31 ad_connect axi_deltaSigma/dma_ack_tx sys_ps7/DMA0_ACK
32
33 ad_connect sys_rstgen/peripheral_aresetn axi_deltaSigma/DMA_REQ_TX_RSTN
```

appended near the end, to add all functionalities needed for audio output.

Some aspects warrant a more detailed description:

- in line 3 the PS configuration is modified to enable the Zynq PL330 DMA-controller control interface. These signals are used for requesting and acknowledging sample tranfers through the DMA between the PS memory and the audio controller FIFO, through an AXI lite bus;
- in lines 10–17 a `clk_wiz` is added and configured to generate a 24.576 MHz frequency. This clock is not used for clocking the $\Delta\Sigma$ signal generation but to provide a slow clock with a rate, independent of the FPGA clock frequency, multiple of commonly used audio frequencies. This is used by a internal prescaler whose maximum value is provided by the driver.

Two ports (lines 6–7) were added to the design, for left and right signals, so the constraint file (`system_constr.xdc`) must be updated by adding the associated pin mapping:

```
# audio
set_property -dict {PACKAGE_PIN M12 IOSTANDARD LVCMOS18} [get_ports bit_right_o]
set_property -dict {PACKAGE_PIN R10 IOSTANDARD LVCMOS18} [get_ports bit_left_o]
```

- Pin `M12` corresponds to `L12N` (`PL_GPIO1`) in connector silkscreen (Fig. 3);
- Pin `R10` corresponds to `L24N` (`PL_GPIO2`) in connector silkscreen (Fig. 3).

Figure 3: Top left: left channel. Bottom: right channel.

The design is now ready to be synthesized with `make`.

Once the bitstream has been generated, the original PlutoSDR bitstream is overwritten with the new one using:

```
cp pluto.runs/impl_1/system_top.bit /somewhere/PlutoSDR/board/pluto
```

2 Software updates

2.1 Buildroot

According to https://github.com/oscimp/plutosdr, a buildroot master must be configured with `zynq_pluto_gnuradio_defconfig`. Since `gr-audio` is enabled by default, no particular configuration must be done for the buildroot part. It is obviously possible to modify the default configuration, through `make menuconfig` to suppress some package. Since `python` weights about half the rootfs image size and is mandatory for executing GNU Radio flowgraphs generated using GNU Radio Companion on the host PC, in addition to GNU Radio SWIG, removing some minor packages will hardly improve the image size.

2.2 Linux kernel configuration

Two modifications must be brought to the kernel configuration.

First adding some nodes to the devicetree (`arch/arm/boot/dts/zynq-pluto-sdr-revb.dts`) is needed to add support for audio devices. An additional entry must be added to the devicetree part beginning with `/ {` and finishing with `};`

```
1   audio_clock: audio_clock {
2       compatible = "fixed-clock";
3       #clock-cells = <0>;
4       clock-frequency = <24576000>;
5   };
6
7   codec_out: fake_codec {
8       #sound-dai-cells = <0>;
9       compatible = "ggm,fake_codec";
10      clocks = <&audio_clock>;
11      clock-names = "mclk";
12
13      status = "okay";
14  };
15
16  axi_deltaSigma_0: axi-deltaSigma@43c00000 {
17      #sound-dai-cells = <0>;
18      compatible = "ggm,axi-deltaSigma";
19      reg = <0x43c00000 0x1000>;
20      dmas = <&dmac_s 0>;
21      dma-names = "tx";
22      clocks = <&clkc 15>, <&audio_clock>;
23      clock-names = "axi", "ref";
```

```
24      };
25
26      pluto_sound {
27          compatible = "simple-audio-card";
28          simple-audio-card,name = "PlutoSDR PWM";
29          simple-audio-card,widgets = "Line", "Line Out";
30          simple-audio-card,routing =
31              "Line Out", "LOUT",
32              "Line Out", "ROUT";
33
34          simple-audio-card,dai-link@0 {
35              format = "i2s";
36              cpu {
37                  sound-dai = <&axi_deltaSigma_0>;
38                  frame-master;
39                  bitclock-master;
40              };
41              codec {
42                  sound-dai = <&codec_out>;
43              };
44          };
45      };
```

Again, some details must be provided. According to devicetree rules, the `.dts` must be a description as close as possible to reality. Hence, the reference clock frequency is provided by a clock driver (`fixed-clock`). The node `pluto_sound` is used to connect the `cpu` part (our IP) and the `codec` representing the peripheral that converts the numeric format, provided by CPU, to an analog signal. In the case of the demonstration given later, a fake driver will be used only because the codec is mandatory.

The second modification focuses on the Linux kernel configuration (`make linux-menuconfig`) to add audio support and PL330 DMA engine used by the audio controller, which are not not enabled by default:

```
Device Drivers -->
    <*>Sound card support -->
        <*>   Advanced Linux Sound Architecture -->
        [...]
        [ ]   ARM sound devices
        [ ]   SPI sound devices
        [ ]   USB sound devices  ----
        <*>   ALSA for SoC audio support  --->
            <*>   Audio support for Analog Devices reference designs
            <*>     AXI-I2S support
            <*>   ASoC Simple sound card support
    [*] DMA Engine support  --->
        [...]
        <*>   DMA API Driver for PL330
```

(Analog Devices devices support is only to have access to the PCM DMA engine).

After having modified the `dts` and the kernel configuration, `make linux` in the Buildroot root directory will compile a new kernel image as well as device tree binary `.dtb` file including the new functionalities.

3 Demonstration

3.1 GNU Radio Companion generated Python scripts

The first demonstration of the proper operation of the sound system embedded on the Pluto SDR relies on a basic sine wave emission on both left and right channels. Using GNU Radio, a `signal source` configured to generate a sine wave at 1 kHz is connected to an `audio sink` (Fig. 4, top). The result, displayed on an oscilloscope (Fig. 4, bottom) exhibits the proper operation of the processing chain and hence the compatibility of the sound source embedded in the PL with the Linux ALSA framework.

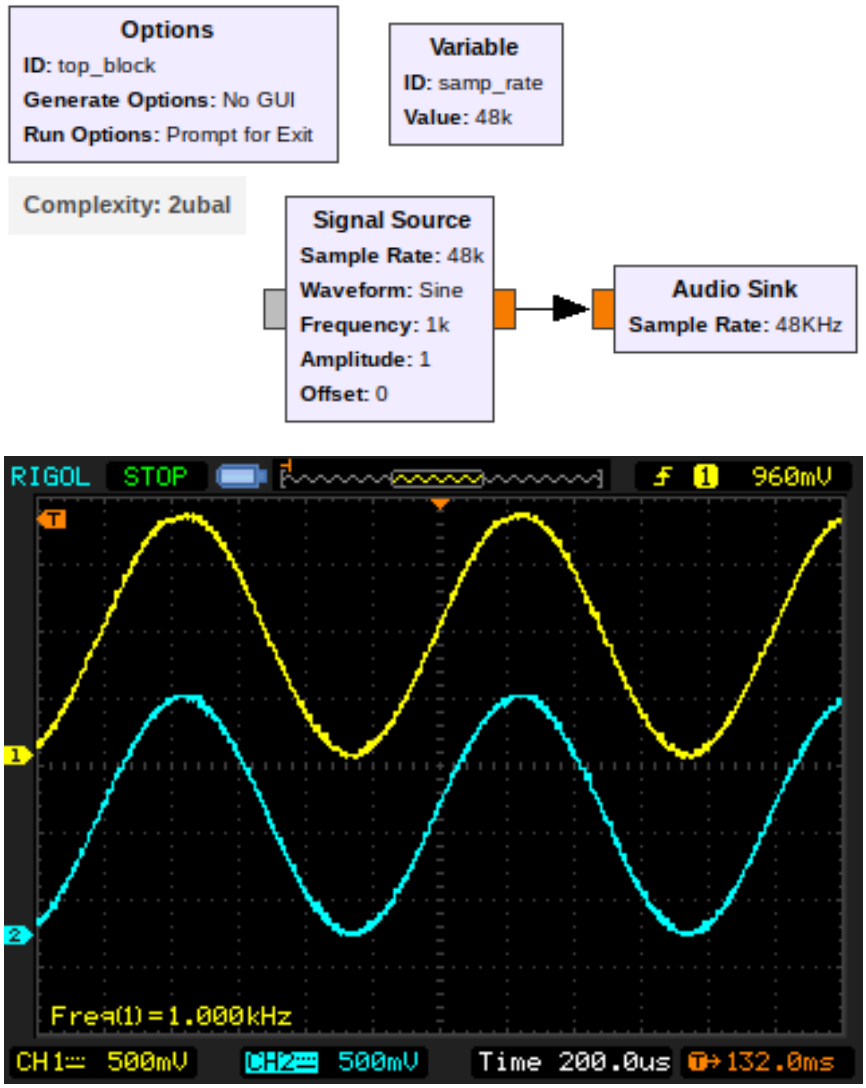

Figure 4: Top: flowgraph generating a 1 kHz signal connected to an audio sink. Bottom: signal observed and displayed through an oscilloscope

A more complex and representative demonstration relies on fetching a radiofrequency

datastream from the AD9363 radiofrequency frontend, transfer the I and Q coefficients through the PL to the PS, feed the PlutoSDR acquisition block to provide the data collected at a FM broadcast channel to the WBFM block after low pass filtering and decimating, and sending the resulting sound signal to the Audio Sink block as demonstrated earlier (Fig. 5).

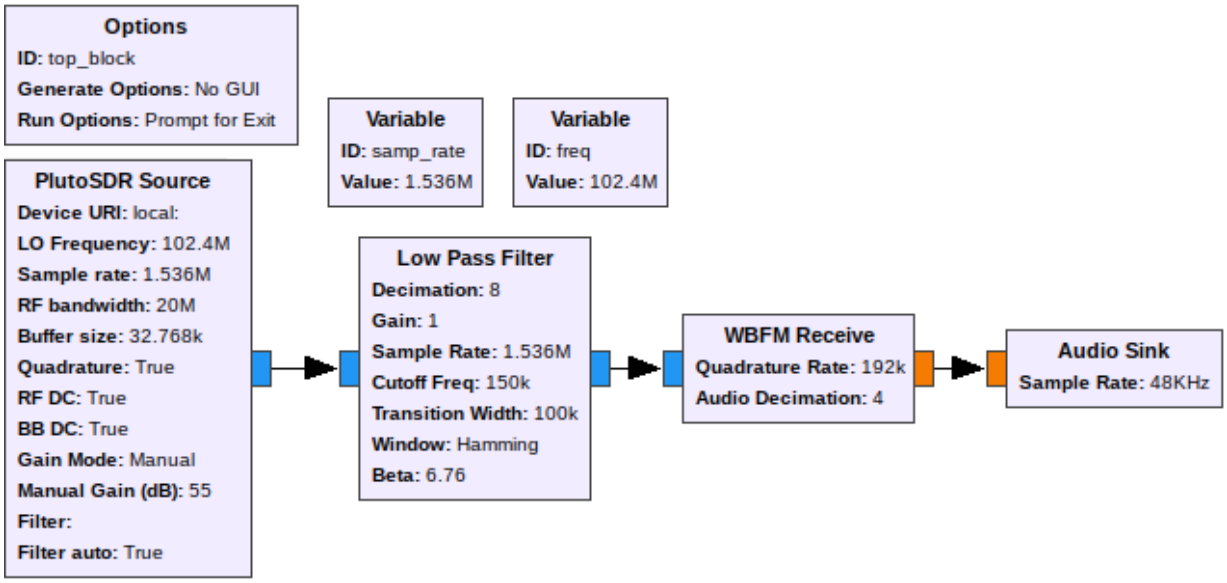

Figure 5: FM broadcast radio signal demodulation on the PlutoSDR PS and playing the resulting audio signal on the embedded sound card.

3.2 C++ call to GNU Radio functions

Python induces a huge footprint on the size of the binary image loaded as RAMdisk. Avoiding Python while accessing GNU Radio functionalities is achieved by calling such functions from C++. As an example of the source to sound card example, the following program allows for generating the continuous tone without requiring Python on the PS.

```
#include <iostream>
#include <gnuradio/top_block.h>
#include <gnuradio/blocks/file_source.h>
#include <gnuradio/blocks/throttle.h>
#include <gnuradio/analog/sig_source_waveform.h>
#include <gnuradio/analog/sig_source_f.h>
#include <gnuradio/msg_queue.h>
#include <boost/make_shared.hpp>
#include <boost/thread/thread.hpp> //sleep
#include <boost/program_options.hpp>
#include <gnuradio/audio/sink.h>
#include <math.h>

int main(void)
```

```
{double samp_rate = 48000; // audio sampling rate
 gr::top_block_sptr top_block;
 top_block = gr::make_top_block("Acquisition demonstration");
 boost::shared_ptr<gr::analog::sig_source_f> source = gr::analog::sig_source_f::make(samp_rate,
       gr::analog::GR_SIN_WAVE, 1000, 1, 0);
 boost::shared_ptr<gr::audio::sink>audio_sink = gr::audio::sink::make(samp_rate, "", true);
 top_block->connect(source, 0, audio_sink, 0);
 top_block->start(); // Start threads and wait
 printf("Hit any key to stop the application\n");
 getchar();
 top_block->stop();
 top_block->wait();
 return EXIT_SUCCESS;
}
```