Redpitaya: second exemple, from ADC to PS

G. Goavec-Mérou, J.-M Friedt

February 4, 2021

This document is a sequel to the previous tutorial on which it is based. It concludes with not only copying the ADC measurements to the DAC, but also with allowing the user to collecte the sample values from the PS for storage or further processing (Fig. 1).
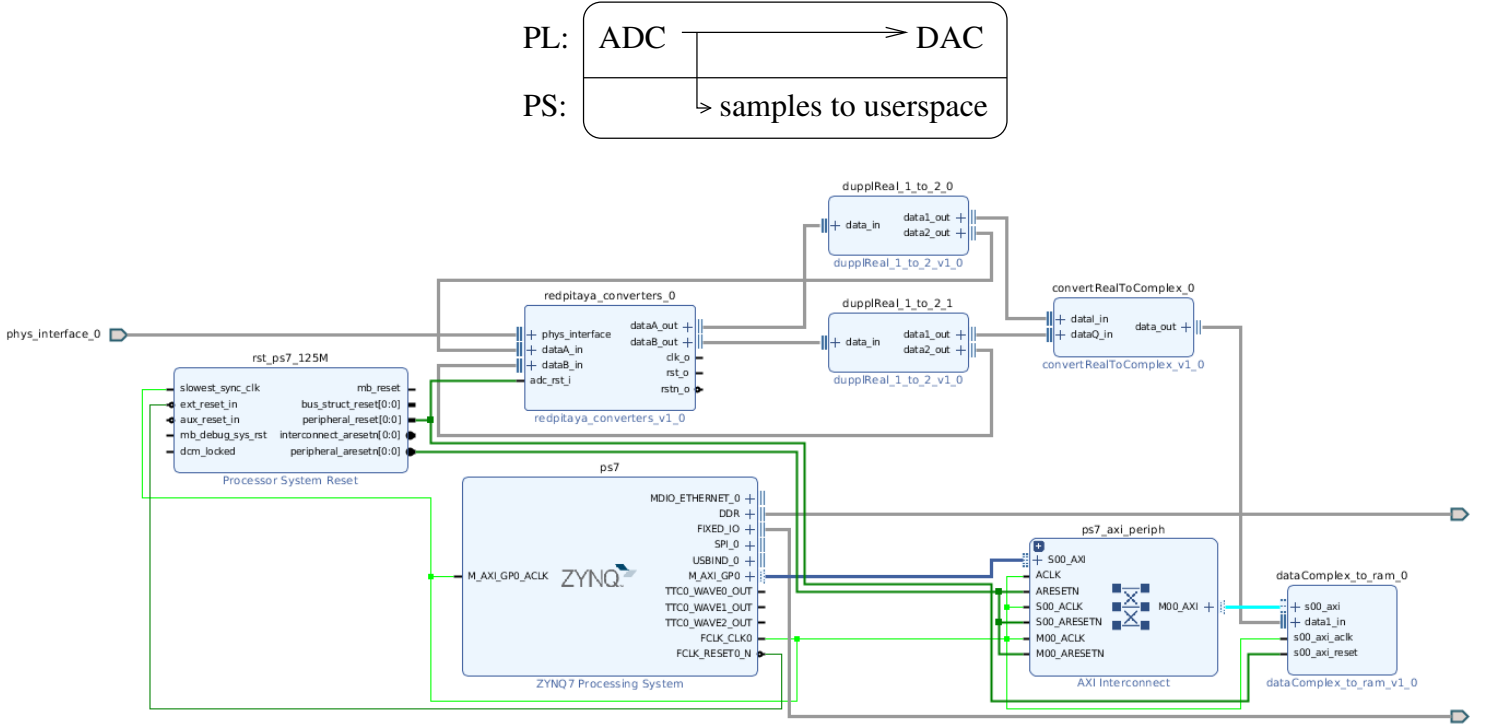

Figure 1: Objective (top) of this tutorial and final schematic of the processing chain described in this document.

1   Sending data to the PS: the PL side

Transfering data towards the PS *in addition* to sending the stream to the DAC requires duplicating the data on the one hand, and interleaving the two streams from the two ADCs on the other hand.

Doing so is achieved by:

1. add `dupplReal_1_to_2`, `convertRealToComplex` (converting two real data streams to one complex) and then `dataComplex_to_ram` (renamed as `data1600`

2. Double click on `dupplReal_1_to_2` to configure to 14 bit (or 16-bit if using the 16-bit Redpitaya) data (doing so on the two stream doublers). Same for convertRealTo-Complex

3. Link `data1_out` of each `dupplReal` to `data1_in` and `data2_in` respectively of `convertRealToComplex`

4. Cut the wires linking the ADC and DAC, and use the two free outputs of the `dupplReal_1_to_2` blocks

5. Connect `dataA_out` and `dataB_out` of the ADCs to the two free inputs of the `dupplReal_1_to_2` blocks

6. `dataComplex_to_RAM`: configure `Data Size` to 14 bits (or 16-bit if using the 16-bit Redpitaya), `Nb Input` to 1 and `Nb Sample` to the number of samples to be transfered. For example, defining 4096 **sample pairs** (complex numbers) each encoded as 16-bit values, or a total of 16384 bytes available to the PS.

Once these blocks have been defined and connected, execute `Run Connection Automation` for connecting to the AXI bus.

2 Sending data to the PS: the Linux kernel side

The driver needed to fetch data on the PS from Linux will be `data_to_ram_core`. Compiling this kernel module requires exporting the variables

```
export BOARD_NAME=redpitaya
export BR_DIR=${HOME}/buildroot-2018.08.1/
```

Compilation is achieved from the `$OSCIMP_DIGITAL_DRIVER/data_to_ram_core` directory by running
`make install`
which will install the `.ko` in `$OSCIMP_DIGITAL_NFS/$BOARD_NAME/modules`.

We have briefly introduced the `devicetree overlay` in `1-PL`. In the context of this tutorial, where we must communicate with an IP, the `overlay` approach is mandatory. This file provides both bitstream name and, through a sub-node, information used by the Linux kernel to know which driver must be probed, and the base address of the memory segment shared between PS and PL, allocated for communications between the IP and the associated driver.

The plugin to the devicetree is generated thanks to the `module_generator` tool located in `$OSCIMP_DIGITAL_APP/tools/module_generator`: this tool is designed to get the definition of all resources from an XML file written manually. In this example, the configuration file is

```xml
<?xml version="1.0" encoding="utf-8"?>
<project name="tutorial3" version="1.0">
  <options>
    <option target="makefile" name="DONT_USE_LIB">1</option>
  </options>
  <ips>
    <ip name ="dataComplex_to_ram" >
      <instance name="data1600" id = "0"
        base_addr="0x43c00000" addr_size="0xffff" />
    </ip>
  </ips>
</project>
```

with the `name` tag including the name of the bitstream without the `.bit.bin` extension nor the "_wrapper" part of its name (notice that the default name `tutorial3` is given by Vivado to generate `tutorial3_wrapper.bit.bin`). The `ip` tag informs on the IP (module_generator uses this information to deduce which driver is used), while `base_addr` and `addr_size` provide the address starting point and range as provided in each IP attribute by Vivado (Fig. 2). `option` node tell `module_generator` to add `DONT_USE_LIB` in the Makefile. This variable is used to avoid application to be linked to the liboscimp_fpga (see. 4-FIR for library introduction).

This XML file may be written manually or by using a script :

```
vivado −mode batch −source $OSCIMP_DIGITAL_IP/scripts/gen_module_generator_xml.tcl −tclargs →
    ↪. project_name bd_name
```

Where the first argument is the directory where .xpr is stored, the second is the name of the project and the last is the name of the block design.

This script generate a file called project_name.xml in the upper directory.

Figure 2: Address range used by each IP able to communicate between the PL and PS through the AXI bus.

This XML is used to generate devicetree file, script to load drivers and apply the overlay, and a Makefile dedicated to compile the application, the dtbo and to install all files in `$OSCIMP_DIGITAL_NFS/tutorial3`.

This task is done by this command:

`/somewhere/oscimpDigital/app/tools/module_generator/module_generator my_file.xml`

After generation, a new directory, called `app` is present in the current directory, containing all files. The Makefile is basic since it provides the application name and simply includes `$OSCIMP_DIGITAL_APP/Makefile.inc`. `make` will compile the application (`tutorial3_us`)

and the dtbo (`tutorial3.dtbo`) while `make install` will 1/ create
`$OSCIMP_DIGITAL_NFS/$BOARD_NAME/tutorial3`, and 2/ copy binary files in a sub-directory
called `bin`.

The rest needs some explanations.

2.1   Devicetree overlay

For the current design the overlay looks like:

```
/dts−v1/;
/plugin/;

/ {
  compatible = "xlnx,zynq−7000";

  fragment0 {
    target = <&fpga_full>;
    #address−cells = <1>;
    #size−cells = <1>;
    __overlay__ {
      #address−cells = <1>;
      #size−cells = <1>;

      firmware−name = "tutorial3_wrapper.bit.bin";

      data1600: data1600@43c00000{
        compatible = "ggm,dataToRam";
        reg = <0x43c00000 0xffff>;
      };

    };
  };
};
```

Compared to the previous tutorials, this devicetree provides a subnode to declare a driver. This node is describe by a name used, at runtime, for the pseudo-file in `/dev`, an attribute `compatible` used by Linux to match between this entry and the `compatible` attribute of the core driver and, finally, an attribute `reg` wich provides the base address in memory and size of this slice.

2.2   Loader script

The second file created by `module_generator` is a script called `tutorial3_us.sh`:

```
CORE_MODULES_DIR=../../modules

mkdir −p /lib/firmware
cp ../bitstreams/tutorial3_wrapper.bit.bin /lib/firmware
DTB_DIR=/sys/kernel/config/device−tree/overlays/fpga
rmdir $DTB_DIR
mkdir $DTB_DIR
```

```
cat tutorial3.dtbo > $DTB_DIR/dtbo

insmod ${CORE_MODULES_DIR}/data_to_ram_core.ko
```
this script copies the bitstream to `/lib/firmware`, applies the overlay and loads the core driver.

2.3 Source code

This file is not provided by `module_generator` and must be created/written manually.
```c
#include <stdio.h>
#include <stdint.h>
#include <fcntl.h>
#include <unistd.h>

int main()
{int k,fi,fo; char c[16384];
 fi=open("/dev/data1600",O_RDWR); fo=open("/tmp/data.bin",O_WRONLY|O_CREAT);
 for (k=1;k<5;k++) {read(fi,c,16384); write(fo,c,16384); }
 close(fi); close(fo);
}
```
where we open `/dev/data1600` to read 5 time 16384 samples and write these values in an other file in binary format.

3 On the Redpitaya ...

Having completed all compilation and installation steps, we have in `$OSCIMP_DIGITAL_NFS/$BOARD_NAME`:

1. `tutorial3.dtbo`, `tutorial3_us.sh` and `tutorial3_us` in `tutorial3/bin` directory;

2. `tutorial3_wrapper.bit.bin` in `tutorial3/bitstreams`;

3. `data_to_ram_core.ko` in `modules/`

The next task, before using the application is to load the bitstream, the overlay and the driver. This is done by the command: `sh tutorial3_us.sh`.

If all goes well, the blue LED on the Redpitaya board will light up (the bitstream has been used to configure the FPGA) and the kernel module is loaded. The last step is to fetch data from userspace: `./tutorial3_us`

whos execution will generate a binary data file loaded in GNU/Octave with
```
f=fopen('data.bin')
d=fread(f,inf,'int16');
plot(d(2:2:end));
```
providing a result as exhibited in Fig. 3 in which the input of the ADC is directly connected to the non-differential to differential amplifier input in order to exploit aliasing on purpose (bypassing the low-pass filter).
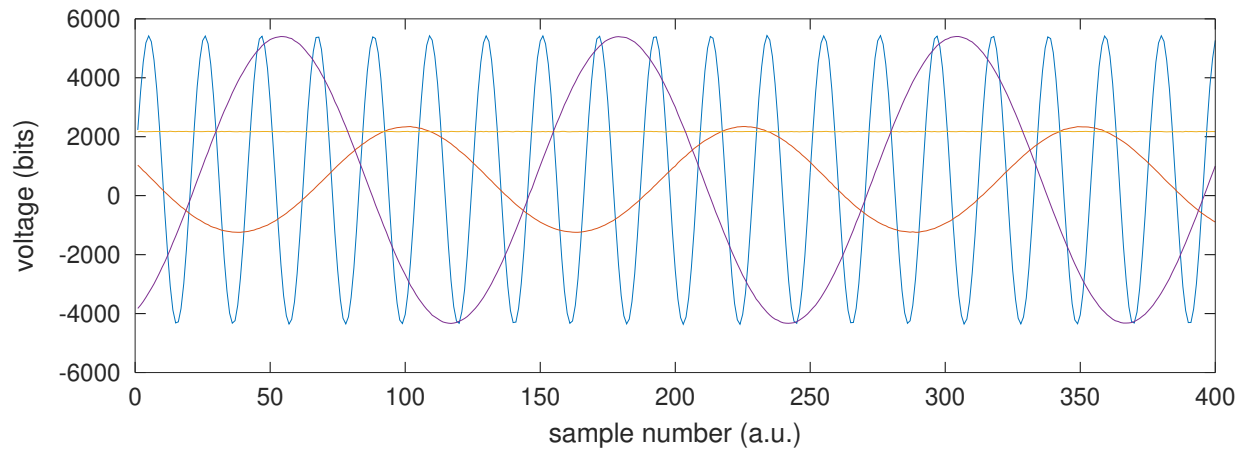
Figure 3: Acquisitions from FPGA by the userspace program communicating with the kernel through the `/dev/data1600` device: sine waves the generated at a level of +6 dBm at 1 MHz, 6 MHz, 124 MHz (aliased to weaker 1 MHz) and 125 MHz (aliased to the DC signal).