

VIDEO PROCESSING WITH INTEL FPGAS

WEBINAR SERIES – Q4'2020

Session 1.1 – DisplayPort Loopback Design

Francisco Perez
Intel FPGA Field Applications Engineer
v.1 – October 2020

Contents

1. Introduction	3
1.1. Introduction	3
1.2. Requirements	3
1.3. References.....	3
1.4. Implementation diagram.....	4
1.5. Hardware Setup.....	5
2. Generating and testing the design.....	6
2.1. Generating the design flow	6
2.2. Testing the design	6
3. Architecture Details	8
3.1. Block Diagram.....	8
3.1.1. Core System Components	9
3.1.2. DisplayPort RX PHY Top and TX PHY Top Components.....	10
3.1.3. Top Level Common Blocks.....	10
3.2. Clocking Scheme.....	12
3.2.1. Top Level Clock Diagram.....	12
3.2.2. Clock Signal Descriptions	13
4. Detailed Design Walkthrough	15
4.1. Design Example Generation	15
4.2. Review the generated project	21
4.3. Modifying the Software Application	28
5. Summary	38

1. Introduction

1.1. Introduction

In this lab manual we are explaining the steps, you need to follow, to implement a working DisplayPort application, able to receive video stream generated from a source like computer, laptop or Blu-ray player, process it and display on an external monitor.

We are explaining the process to generate the hardware design for the FPGA fabric, and how to build the software application for an embedded processor like Nios II. The processor is used to configure all the modules and handle all the link training procedures with both the DisplayPort video source and with the DisplayPort compliant monitor.

The Intel FPGA DisplayPort IP core is compliant with *VESA DisplayPort Standard version 1.4*.

NOTE: There are no project files associated with this manual. We are generating the project from scratch.

1.2. Requirements

For this implementation, we are using the following setup:

- Cyclone® 10 GX Development Kit
https://www.intel.com/content/www/us/en/programmable/products/boards_and_kits/dev-kits/altera/cyclone-10-gx-development-kit.html
- Bitec DisplayPort daughter card rev.11
<https://bitec-dsp.com/product/fmc-displayport-daughter-card-revision-11/>
- Intel® Quartus Pro ACDS 20.3
<https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/overview.html>
- CentOS 7.6 (but other Linux distros as well as Windows are supported)

1.3. References

The purpose of this document is to guide you through the process of creating the different building blocks, and pull all together, to assembly a working application. We are not explaining in detail all the different settings and options available. We also assume you know how to get all the development tools (Quartus and Nios2 EDS) currently installed in your preferred OS. For more detailed information you can use the dedicated User Guides:

- Intel FPGA DisplayPort IP User Guide
<https://www.intel.com/content/www/us/en/programmable/products/intellectual-property/ip/interface-protocols/m-alt-displayport-megacore.html>
- Cyclone 10 GX DisplayPort Design Example User Guide
<https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug-dex-dp-c10gx.pdf>
- AN745-Design Guidelines for DisplayPort Interface
https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/an/an_745.pdf
- Quartus Prime Pro Installation Guide

https://www.intel.com/content/dam/altera-www/global/en_US/pdfs/literature/manual/quartus_install.pdf

- NiosII EDS installation

https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/nios2/n2sw_nii5v2gen2.pdf

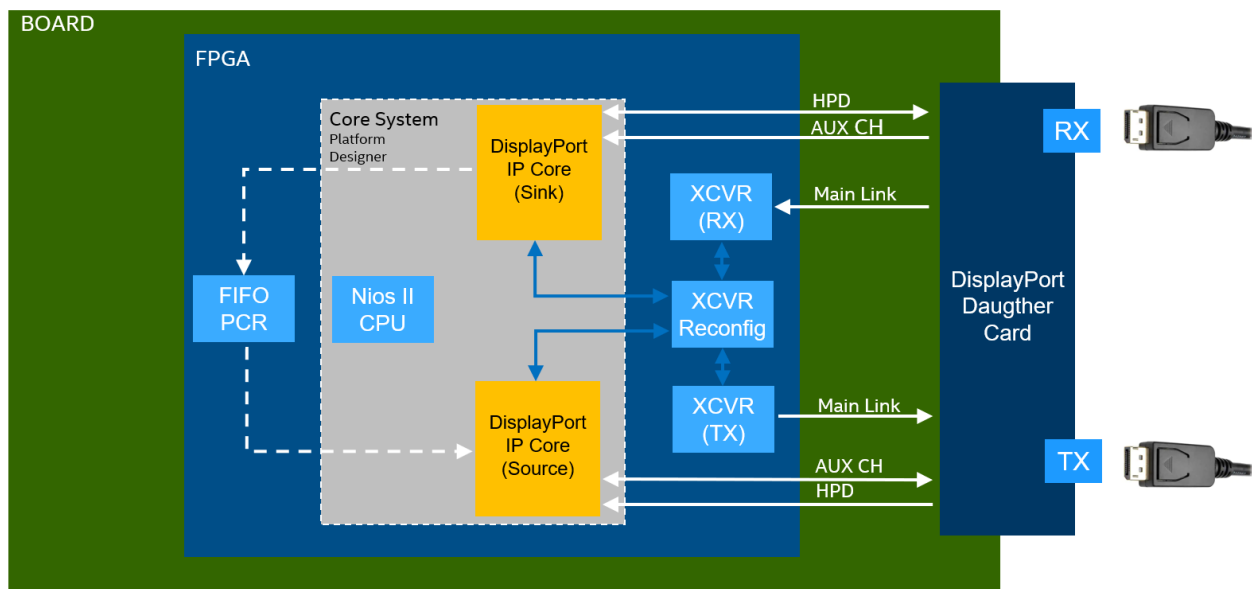
- Embedded Design Handbook

https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/nios2/edh_ed_handbook.pdf

1.4.Implementation diagram

Find in the below figure, a high-level block diagram with the hardware implementation. Inside the FPGA, we are configuring a set of high-speed transceivers to receive and transmit the DisplayPort video streams in serialized form, acting as the physical layer. Attached to them, we have the DisplayPort IP cores for Sink and Source implementation, these are our link layer blocks.

The video packets received by the Sink are passed through a Dual-Clock FIFO and sent to the DisplayPort Source IP core to re-transmit. Inside the same block, we are also performing Pixel Clock Recovery from the link clock.



For implementing this system, we are using an included feature in our DisplayPort IP cores: **Example Design Generation**. We'll go into full details on how to generate it later in the document, but for now, let's discuss a bit on the architecture.

The example design script generates the following modules:

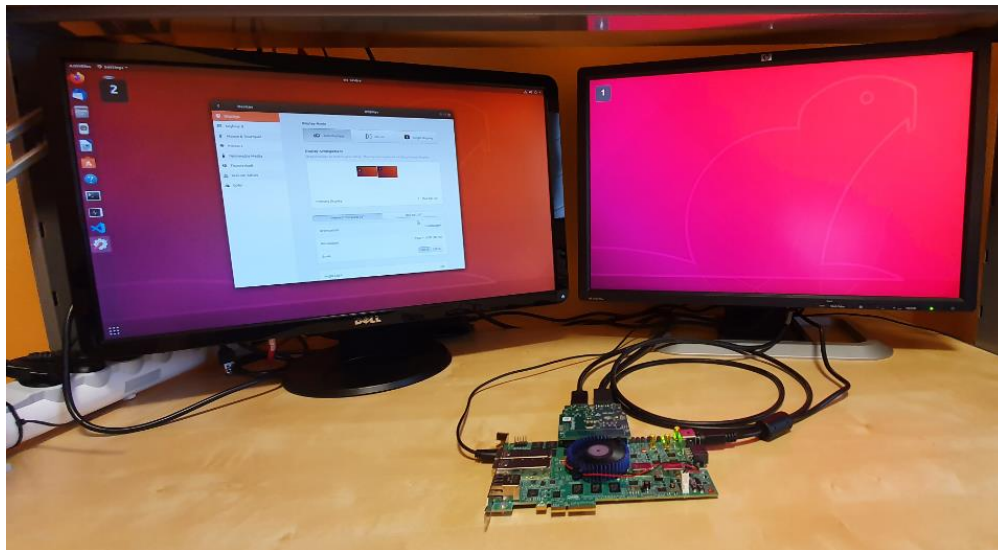
- **Core System** - The core system consists of the Nios II Processor and its necessary components, DisplayPort RX and TX core sub-systems.
- **DisplayPort RX subsystem (Sink)** – Includes the parameterized DisplayPort RX core (link layer)
- **DisplayPort TX subsystem (Source)** - Includes the parameterized DisplayPort TX core (link layer)
- **XCVR (RX)** – Transceiver Native PHY that deserializes high speed data (physical layer)
- **XCVR (TX)** – Transceiver Native PHY that serializes high speed data (physical layer)

- **XCVR Reconfig** – Transceiver arbiter and reconfiguration control to support different link rates (RBR, HBR, HB2, HBR3)
- **FIFO PCR** – The design uses the pixel recovery clock (PCR) to recover the pixel clock according to the received MSA information from the sink and converts the RX parallel video interface to the standard VSYNC/HSYNC/DE interface. The PCR output drives the source video interface and encodes to the DisplayPort main link before transmitting to the monitor.

1.5. Hardware Setup

The DisplayPort Intel FPGA IP design example performs a loop-through for a standard DisplayPort video stream.

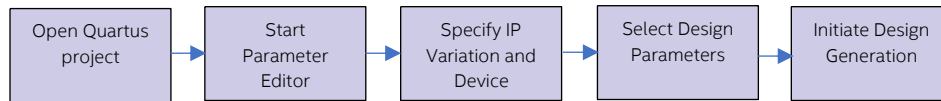
1. To run the hardware test, connect a DisplayPort-enabled source device to the DisplayPort FMC daughter card sink input.
2. The DisplayPort sink decodes the port into a standard video stream and sends it to the clock recovery core.
3. The clock recovery core synthesizes the original video pixel clock to be transmitted together with the received video data.
4. The clock recovery core then sends the video data to the DisplayPort source and the Transceiver Native PHY TX block.
5. Connect the DisplayPort FMC daughter card source port to a monitor to display the image.
6. You can check the On-board User LED in the devkit for status information



2. Generating and testing the design

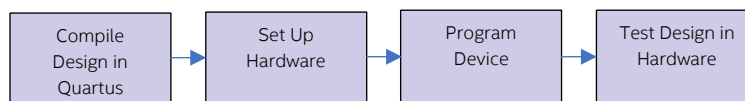
We are using the DisplayPort Intel FPGA IP parameter editor in the Intel Quartus Prime Pro Edition software to generate the design example.

2.1. Generating the design flow



1. Click **Tools** ➤ **IP Catalog**, and select Intel Cyclone 10 GX as the target device family.
Note: The design example only support Intel Cyclone 10 GX devices.
2. In the IP Catalog, locate and double-click **DisplayPort Intel FPGA IP**. The **New IP Variation** window appears.
3. Specify a top-level name for your custom IP variation. The parameter editor saves the IP variation settings in a file named `<your_ip>.ip`.
4. Click **OK**. The parameter editor appears.
5. Configure the desired parameters for both TX and RX.
Note: The Nios II software has the capability to read and print out the DisplayPort Main Stream Attribute (MSA) information in the Nios II terminal. To read or print the MSA information, turn on the **Enable GPU Control** parameter.
6. On the **Design Example** tab, select **DisplayPort SST Parallel Loopback With PCR**.
7. Select **Synthesis** to generate the hardware design example.
8. For **Target Development Kit**, select **Cyclone 10 GX FPGA Development Kit**. If you select the development kit, then the target device changes to match the device on the development kit. For **Cyclone 10 GX FPGA Development Kit**, the default device is 10CX220YF780E5G.
9. Click **Generate Example Design** to generate the project files and the software Executable and Linking Format (ELF) programming file.

2.2. Testing the design



To compile and run a demonstration test on the hardware example design, follow these steps:

1. Ensure hardware example design generation is complete
2. Launch the Intel Quartus Prime Pro Edition software and open `<project directory>/quartus/c10_dp_demo.qpf`.

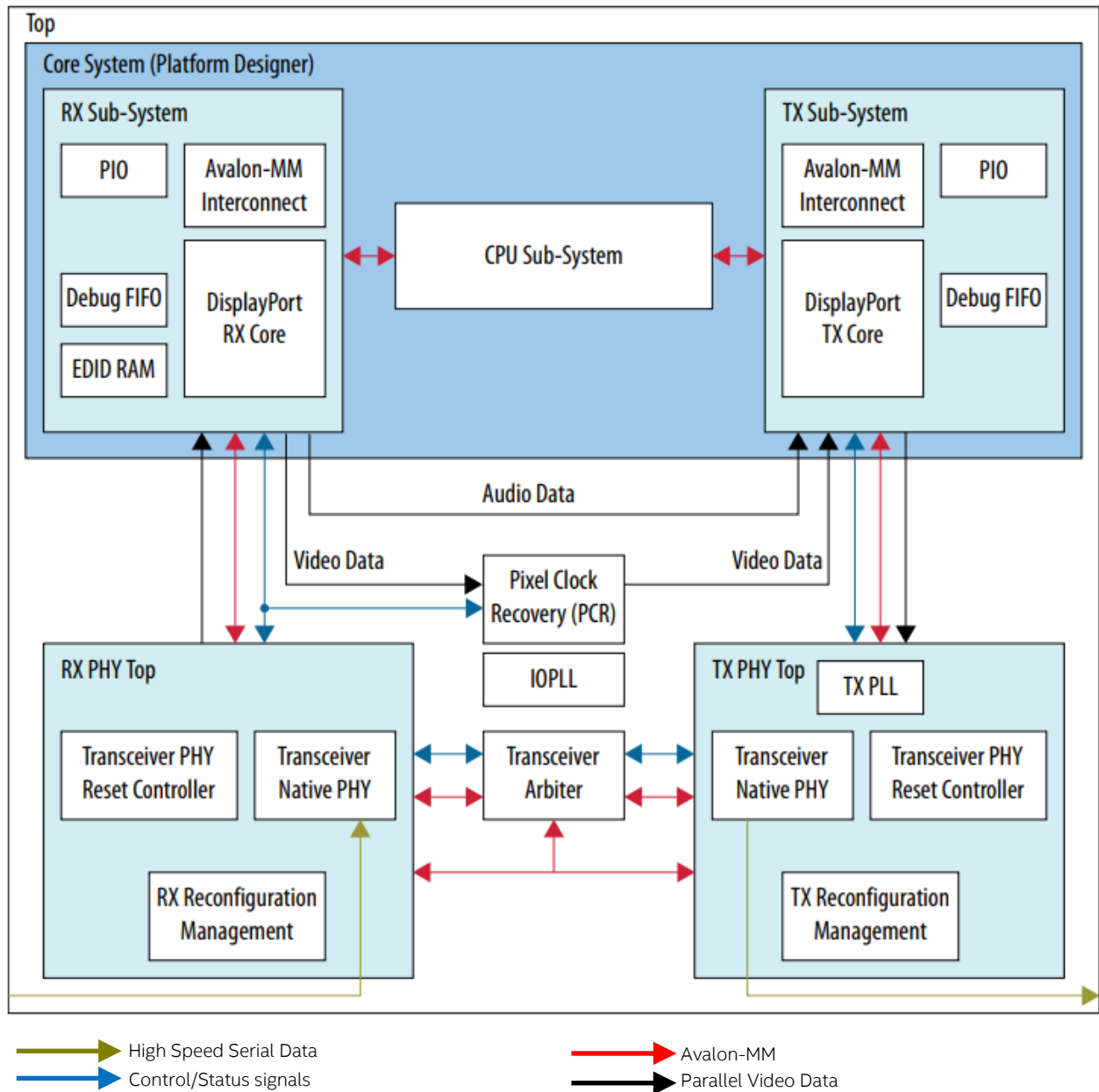
3. Click **Processing ► Start Compilation**.
4. After successful compilation, the Intel Quartus Prime Pro Edition software generates a `.sof` file in your specified directory.
5. Connect the DisplayPort RX connector on the Bitec daughter card to an external video source, such as the graphics card on a PC.
6. Connect the DisplayPort TX connector on the Bitec daughter card to a video analyzer or a DisplayPort sink device, such as a PC monitor.
7. Ensure all switches on the development board are in default position.
8. Configure the selected Intel Cyclone 10 GX device on the development board using the generated `.sof` file (**Tools ► Programmer**).
9. The DisplayPort sink device displays the video generated from the video source

By default, the ELF file for the embedded CPU is generated when you generate the dynamic design example. The software application gets compiled and an internal RAM, in the FPGA fabric used as a program memory, is initialized to hold the executable.

3. Architecture Details

3.1. Block Diagram

In the figure below we are including a more detailed architecture of the generated design showing the type of signals that communicate across the different components



3.1.1. Core System Components

Module	Description
Core System (Platform Designer)	<p>The core system consists of the Nios II Processor and its necessary components, DisplayPort RX and TX core sub-systems.</p> <p>This system provides the infrastructure to interconnect the Nios II processor with the DisplayPort Intel FPGA IP (RX and TX instances) through Avalon memory-mapped (Avalon-MM) interface within a single Platform Designer system to ease the software build flow.</p> <p>This system consists of:</p> <ul style="list-style-type: none"> • CPU Sub-System • RX Sub-System • TX Sub-System
RX Sub-System (Platform Designer)	<p>The RX sub-system consists of:</p> <ul style="list-style-type: none"> • Clock Source—The clock source to the DisplayPort RX core. This sub-system has two clock sources integrated: 100 MHz and 16 MHz. • Reset Bridge—The bridge that connects the external signal to the sub-system. This bridge synchronizes to the respective clock source before it is used. • DisplayPort RX Core—DisplayPort Sink IP core, <i>VESA DisplayPort Standard version 1.4</i>. • Debug FIFO—This FIFO captures all DisplayPort RX auxiliary cycles, and prints out in the Nios II Debug terminal. • PIO—The parallel IO that triggers the MSA capture, and prints out when the on-board push button (PB) is pressed. • Avalon-MM Pipeline Bridge—This Avalon-MM bridge interconnects the Avalon-MM interface between components within the RX sub-system to the Nios II processor in the Core sub-system. • EDID—The EDID RAM is only used to store the desired EDID value in the RAM and connect to the DisplayPort Sink IP core. This component is only used when you disable the Enable GPU Control option in the RX core.
TX Sub-System (Platform Designer)	<p>The TX sub-system consists of:</p> <ul style="list-style-type: none"> • Clock Source—The clock source to the DisplayPort TX core. This sub-system has two clock sources integrated: 100 MHz and 16 MHz. • Reset Bridge—The bridge that connects the external signal to the sub-system. This bridge synchronizes to the respective clock source before it is used. • DisplayPort TX Core—DisplayPort Source IP core, <i>VESA DisplayPort Standard version 1.4</i>. • Debug FIFO—This FIFO captures all DisplayPort TX auxiliary cycles, and prints out in the Nios II Debug terminal. This component is only used when the <code>TX_AUX_DEBUG</code> parameter is turned on. • PIO—The parallel IO that triggers the DPTX register update in software (<code>tx_utils.c</code>). • Avalon-MM Pipeline Bridge—This Avalon-MM bridge interconnects the Avalon-MM interface between components within the TX sub-system to the Nios II processor in the Core sub-system.

3.1.2. DisplayPort RX PHY Top and TX PHY Top Components

Module	Description
RX PHY Top	<p>The RX PHY top level consists of the components related to the receiver PHY layer.</p> <ul style="list-style-type: none"> Transceiver Native PHY (RX)—The transceiver block that receives the serial data from an external video source and deserializes it to 20-bit or 40-bit parallel data to the DisplayPort sink IP core. This block supports up to 8.1 Gbps (HBR3) data rate with 4 channels. Transceiver PHY Reset Controller—The RX Reconfiguration Management module triggers the reset input of this controller to generate the corresponding analog and digital reset signals to the Transceiver Native PHY block according to the reset sequencing. RX Reconfiguration Management—This block reconfigures and recalibrates the Transceiver Native PHY block to receive serial data in the supported data rates (RBR, HBR, HBR2, and HBR3).
TX PHY Top	<p>The TX PHY top level consists of the components related to the transmitter PHY layer.</p> <ul style="list-style-type: none"> Transceiver Native PHY(TX)—The transceiver block that receives 20-bit or 40-bit parallel data from the DisplayPort Intel FPGA IP and serializes the data before transmitting it. This block supports up to 8.1 Gbps (HBR3) data rate with 4 channels. <p><i>Note:</i> You must set the TX channel bonding mode to PMA and PCS bonding and the PCS TX Channel bonding master parameter to 0 (default is auto).</p> <ul style="list-style-type: none"> Transceiver PHY Reset Controller—The TX Reconfiguration Management module triggers the reset input of this controller to generate the corresponding analog and digital reset signals to the Transceiver Native PHY block according to the reset sequencing. TX Reconfiguration Management—This block reconfigures and recalibrates the Transceiver Native PHY and TX PLL blocks to transmit serial data in the required data rates (RBR, HBR, HBR2, and HBR3). TX PLL—The transmitter PLL block provides a fast serial fast clock to the Transceiver Native PHY block. For the DisplayPort Intel FPGA IP design example, Intel uses transmitter fractional PLL (FPLL).

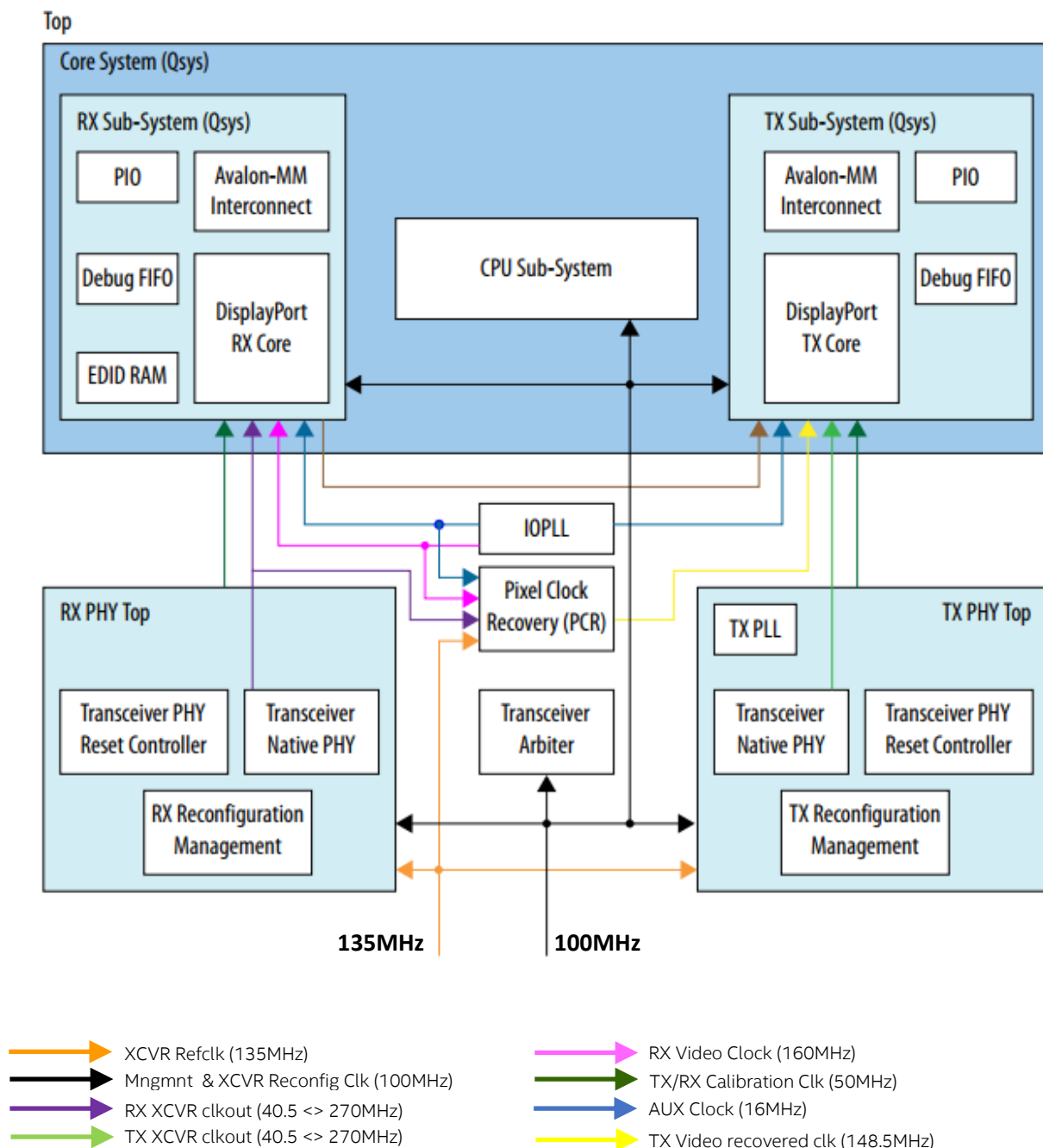
3.1.3. Top Level Common Blocks

Module	Description
Transceiver Arbiter	<p>This generic functional block prevents transceivers from recalibrating simultaneously when either RX or TX transceivers within the same physical channel require reconfiguration. The simultaneous recalibration impacts applications where RX and TX transceivers within the same channel are assigned to independent IP implementations.</p> <p>This transceiver arbiter is an extension to the resolution recommended for merging simplex TX and simplex RX into the same physical channel. This transceiver arbiter also assists in merging and arbitrating the Avalon-MM RX and TX reconfiguration requests targeting simplex RX and TX transceivers within a channel as the reconfiguration interface port of the transceivers can only be accessed sequentially. The transceiver arbiter is not required when only either RX or TX transceiver is used in a channel.</p> <p>The transceiver arbiter identifies the requester of a reconfiguration through its Avalon-MM reconfiguration interfaces and ensures that the corresponding <code>tx_reconfig_cal_busy</code> or <code>rx_reconfig_cal_busy</code> is gated accordingly.</p>
IOPLL	<p>IOPLL generates common source clock: <code>dp_rx_vid_clkout</code> and <code>clk_16</code> (16 MHz) for the DisplayPort system.</p> <ul style="list-style-type: none"> <code>dp_rx_vid_clkout</code>—used as RX core video clock of the video data stream input clock. <code>clk_16</code>—Used as DisplayPort auxiliary clock reference clock.

PCR	<p>The design uses the pixel recovery clock (PCR) to recover the pixel clock according to the received MSA information from the sink and converts the RX parallel video interface to the standard VSYNC/HSYNC/DE interface.</p> <p>The PCR output drives the source video interface and encodes to the DisplayPort main link before transmitting to the monitor.</p> <p>The recovered clock drives the TX video clock.</p>
-----	--

3.2. Clocking Scheme

3.2.1. Top Level Clock Diagram



3.2.2. Clock Signal Descriptions

Clock	Signal Name in Design	Description
TX PLL Refclock	tx_pll_refclk	135 MHz TX PLL reference clock, that is divisible by the transceiver for all DisplayPort data rates (1.62 Gbps, 2.7 Gbps, and 5.4 Gbps). <i>Note:</i> The reference clock source of the TX PLL refclock is located at the HSSI <code>refclk</code> pin.
TX Transceiver Clockout	gxb_tx_clkout	TX clock recovered from the transceiver, and the frequency varies depending on the data rate and symbols per clock.
TX PLL Serial Clock	gxb_tx_bonding_clocks	Serial fast clock generated by TX PLL. The clock frequency is set based on the data rate.
RX Refclock	rx_cdr_refclk	135 MHz transceiver clock data recovery (CDR) reference clock, that is divisible by all DisplayPort data rates (1.62 Gbps, 2.7 Gbps, and 5.4 Gbps). <i>Note:</i> The reference clock source of the RX refclock is located at the HSSI <code>refclk</code> pin.
RX Transceiver Clockout	gxb_rx_clkout	RX clock recovered from the transceiver, and the frequency varies depending on the data rate and symbols per clock.
Management Clock	rx_rcfg_mgmt_clk tx_rcfg_mgmt_clk	A free running 100 MHz clock for both Avalon-MM interfaces for reconfiguration and PHY reset controller for transceiver reset sequence.
Audio Clock	dp_audio_clk	DisplayPort audio clock.
16 MHz Clock	clk_16	160 MHz clock used to encode and decode auxiliary channel in the DisplayPort Intel FPGA IP source and sink IP cores. This clock is also used as a reference clock in the Pixel Clock module for fractional calculation.
Calibration Clock	dp_rx_clk_cal dp_tx_clk_cal	A 50 MHz calibration clock input that must be synchronous to the Transceiver Reconfiguration module's clock. This clock is used in the DisplayPort Intel FPGA IP core's reconfiguration logic.
RX Video Clock	dp_rx_vid_clkout	Video clock for DisplayPort sink to clock video data stream. If <code>MAX_LINK_RATE = HBR2</code> and <code>PIXELS_PER_CLOCK = Dual</code> , video clock uses 300 MHz. Otherwise, fixed to 160 MHz.
TX Video Clock	tx_vid_clk	Recovered video clock from the PCR module that reflects the actual video clock frequency. Used when DisplayPort source's <code>TX_SUPPORT_IM_ENABLE=0</code> .

TX Transceiver Clockout and RX Transceiver Clockout frequencies are based on link rate and specified symbol per clock as per the following table:

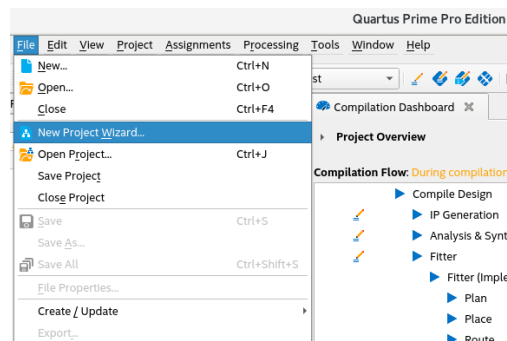
Data Rate	Symbols per Clock	Frequency (MHz)
RBR (1.62 Gbps)	2 (dual)	81
	4 (quad)	40.5
HBR (2.7 Gbps)	2 (dual)	135
	4 (quad)	67.5
HBR2 (5.4 Gbps)	2 (dual)	270
	4 (quad)	135
HBR3 (8.1 Gbps)	4 (quad)	202.5

4. Detailed Design Walkthrough

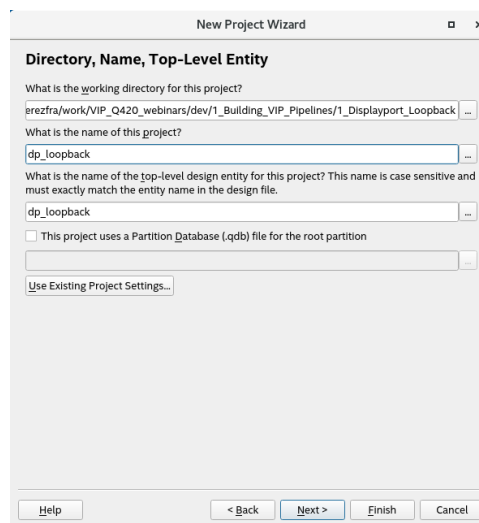
In this chapter we will go across the different steps needed to have our system working in hardware. We will start creating a DisplayPort IP variation file, generate the example design and compile the FPGA bitstream. Then we will explore the generated software application and learn how to import all the files into an Eclipse for NiosII project to modify, compile and debug in the board.

4.1. Design Example Generation

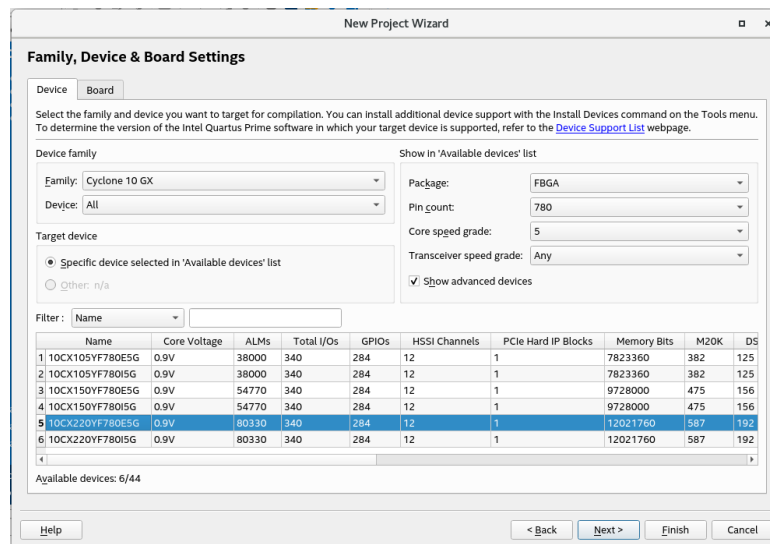
1. Open Quartus and create a new project
 - **File -> New Project Wizard**



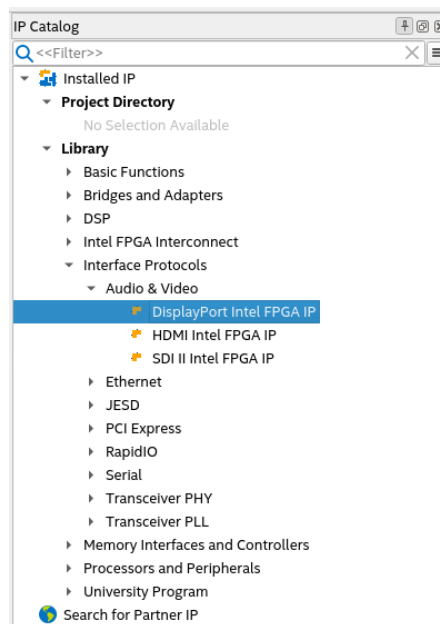
- Select working directory and name for the project. i.e. *dp_loopback*



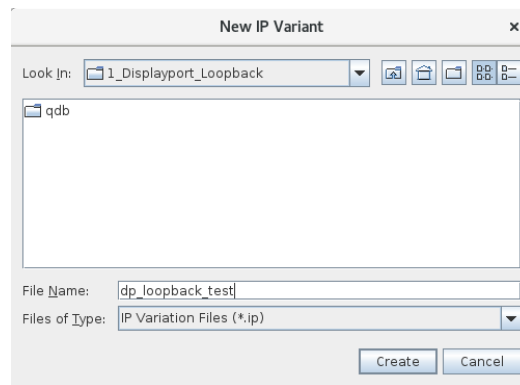
- Click **Next** until **Family, Device & Board Settings** tab.
 Apply the shown filters below and select: **10CX220YF780E5G**, this is the device we have in the kit.



- Click **Next** until the **Summary** tab and then **Finish** to create the project
- 2. Once the project is created, we will start the generation of our example design. By going to the **IP Catalog**, double click on **DisplayPort Intel FPGA IP** under **Library->Interface Protocols->Audio & Video section**

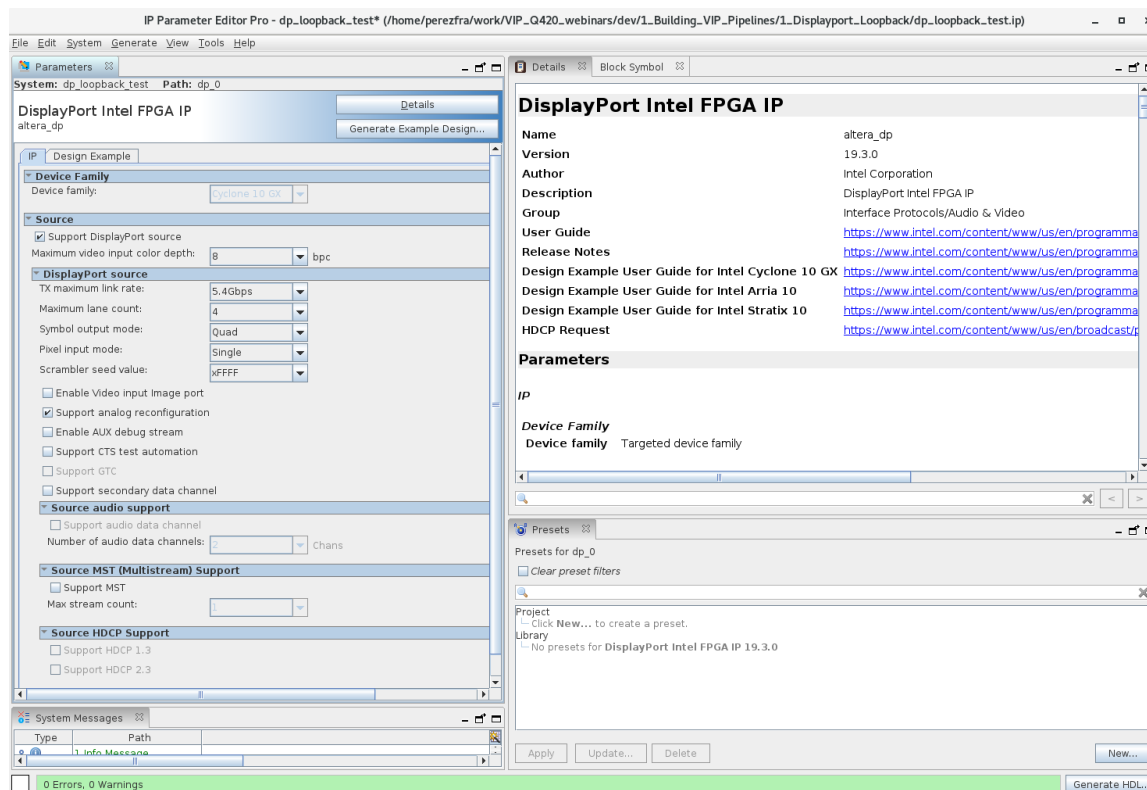


- 3. The **IP Parameter Editor** will open and request a name for the **New IP variant**. Let's use `dp_loopback_test`.



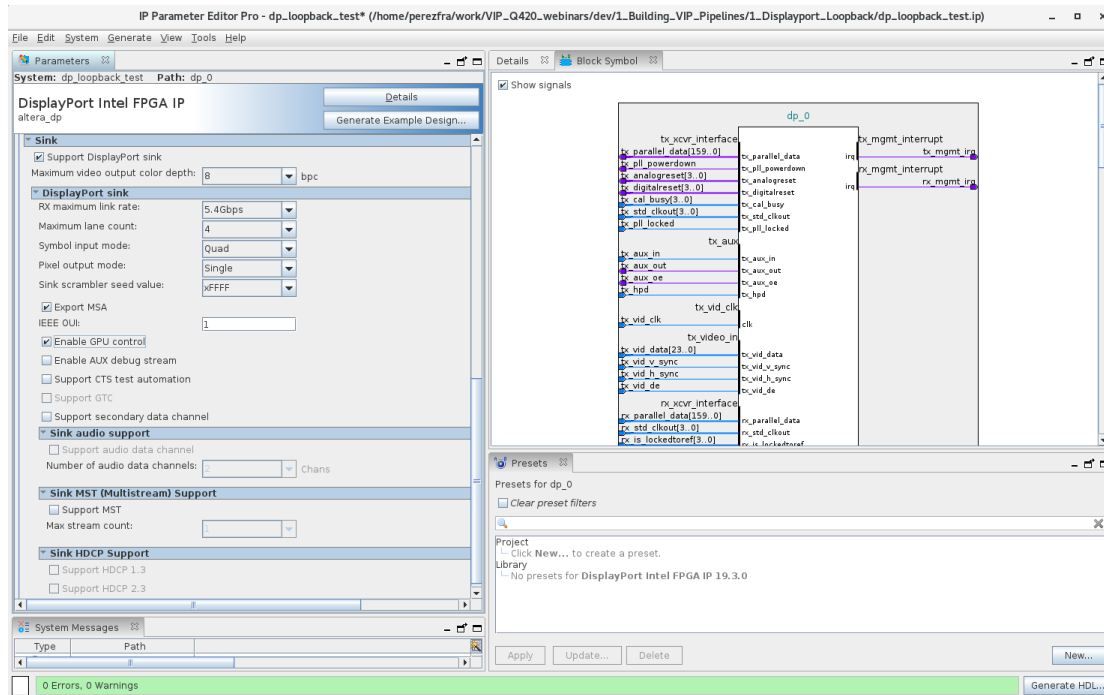
4. Press Create and the IP Parameter Editor will appear, so we can start configuring our IP variation

- We start configuring the DisplayPort Source parameters. For this application we are setting the following values (see below screenshot).
- We are keeping things simple by selecting 8 bpc and single Pixel input mode (enough for 1080p60 resolution using a video clock = 148.5MHz)

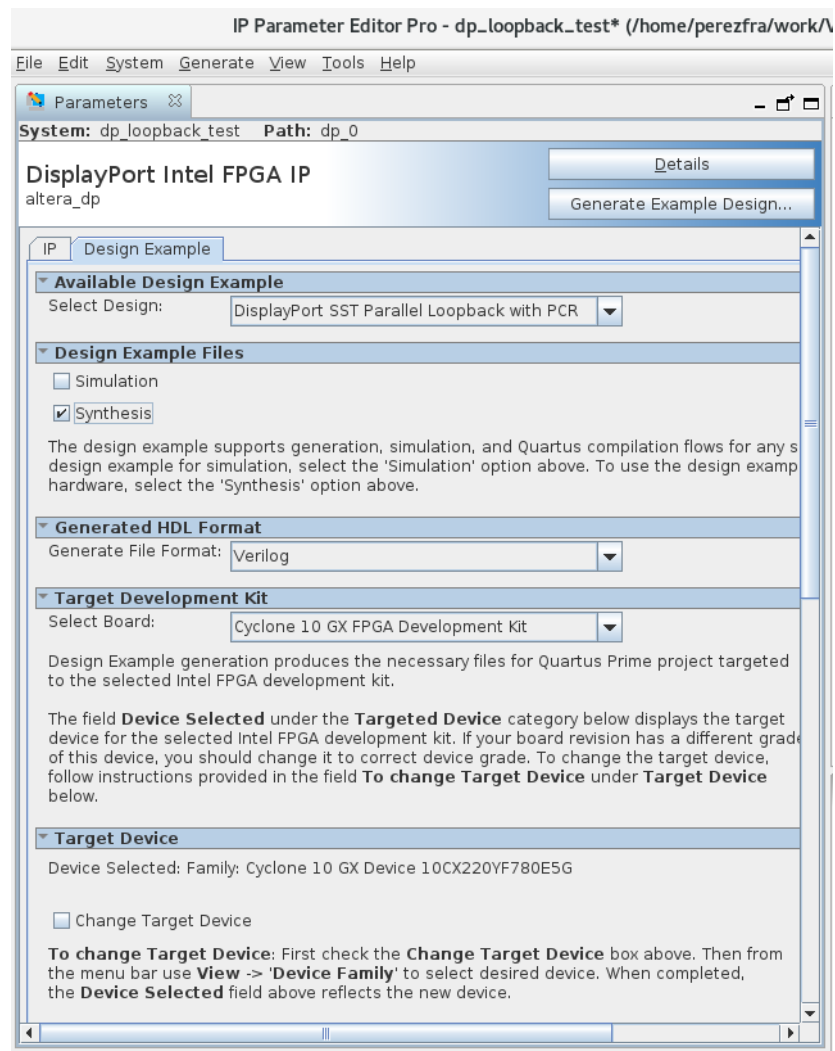


- In the **Details** tab you have links to access all the relevant documentation, as well as description and allowed values for the different parameters.
- We continue with the **DisplayPort Sink** interface, where we are matching the relevant parameters with the Source.

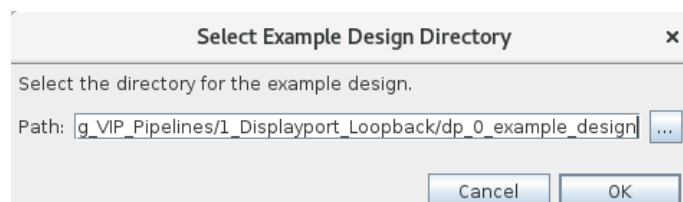
- We enable to **Export MSA** (Main Stream Attribute) data, as this is used in the PCR block to recover the pixel clock frequency.
- We also **Enable GPU control**, as we will use an embedded CPU to control the Sink and the EDID management.

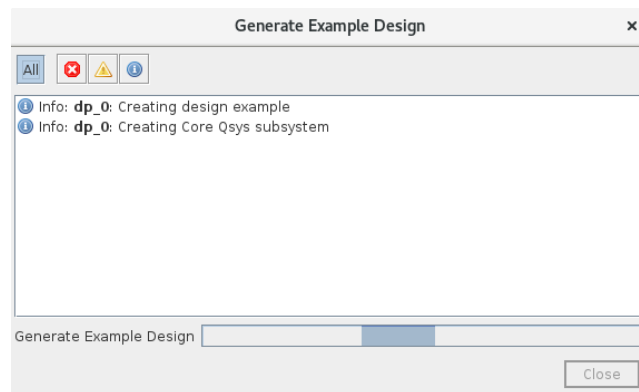


- You can have a look at the different interfaces exported by the IP, viewing the content in the **Block Symbol** tab.
- Navigate to **Design Example** tab to configure the design example generation
 - **Select Design:** DisplayPort SST Parallel Loopback with PCR
 - **Synthesis files generation.** You can also enable simulation if you wish.
 - **HDL Format:** Verilog
 - **Target Development Kit:** Cyclone 10 GX

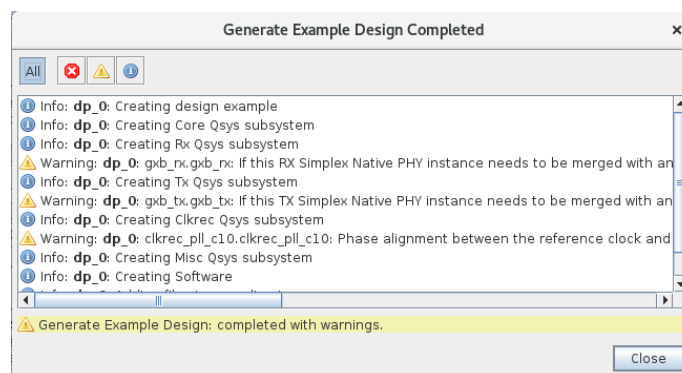


- Click on **Generate Example Design** to kick off the process. You will be asked to select target Directory. Keep it as default (*quartus_project_dir/dp_0_example_design*). Click **OK**

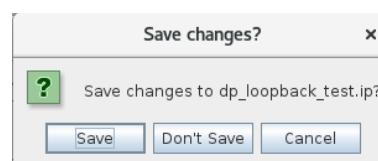




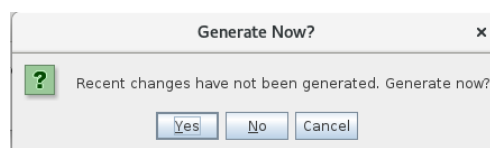
- After some minutes we have our example design files generated. You can safely ignore the warnings, as these are indications already addressed in the implementation.



- We can now close the dialog box and exit the **DisplayPort IP** parameter editor. You will be prompted to save the IP variant. It's recommended to save it, so you can open again to launch a similar design example generation if needed.



- You don't need to Generate the HDL files for this IP variant, we will open the generated Quartus project with the example design and work from there.

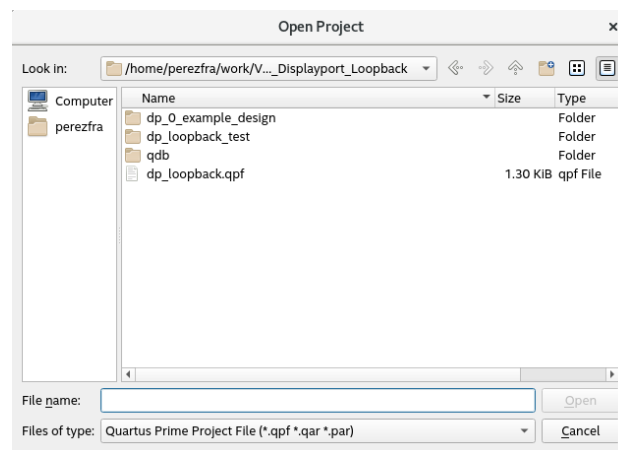


4.2.Review the generated project

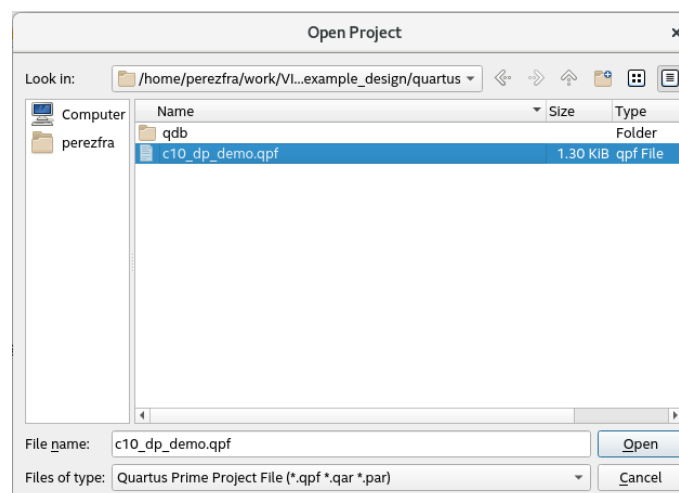
In this section we are opening the generated Quartus project and analyze what modules have been included and the connections done.

1. Open the generated Quartus project

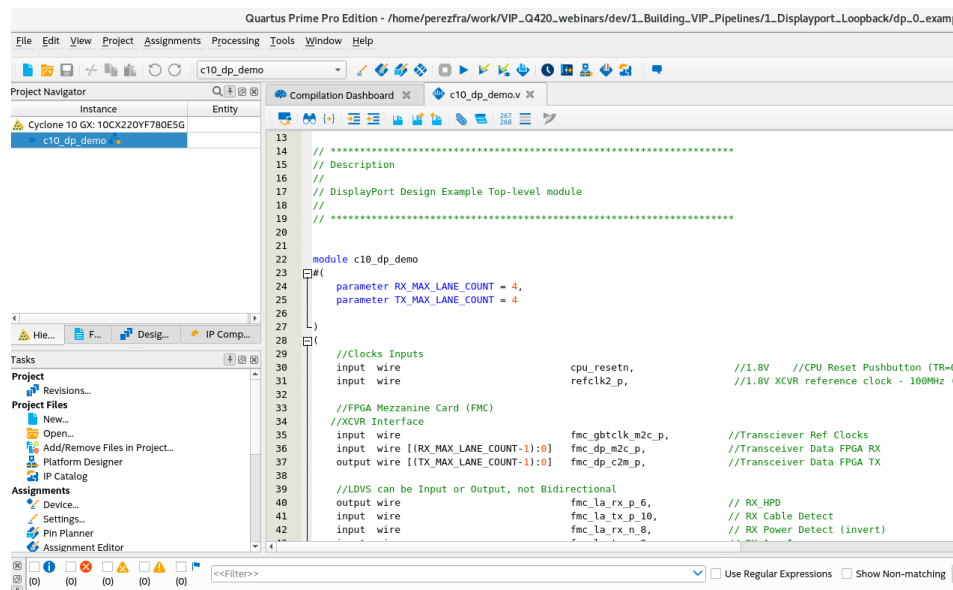
- In Quartus, go to **File->Open Project**. If we are still at the initial project we created, we will see a new folder *dp_0_example_design* which contains our target files.



- Navigate to *dp_0_example_design/quartus* and select **c10_dp_demo.qpf**. Click **Open**



- After opening the project, we see the top level entity *c10_dp_demo.v* already loaded.

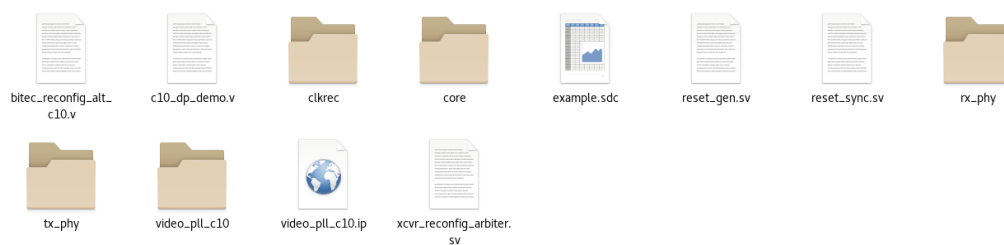


2. Review the generated files

The generation script will build the following folder tree



- generation.log
 - o Contains all the info messages and commands issued along the whole process
- quartus
 - o Contains the Quartus project and setting files (device, ip paths, pin locations, ...)
- rtl



- o Contains all the rtl source files for the project
 - **c10_dp_demo.v** - Top Level
 - **clkrec** – Pixel Clock Recovery module
 - **core** – Contains the Platform Designer modules
 - **dp_core**
 - **dp_rx**
 - **dp_tx**

- **rx_phy** – Transceiver Native PHY instantiation for Sink
- **tx_phy** – Transceiver Native PHY instantiation for Source
- **video_pll_c10** – PLL used to generate AUX_CLK (16MHz) and RX_VIDEO_CLK (160MHz)
- **example.sdc** – Timing constraints and analysis file
- script
 - Includes a couple of files for IP regeneration and software recompilation
- software

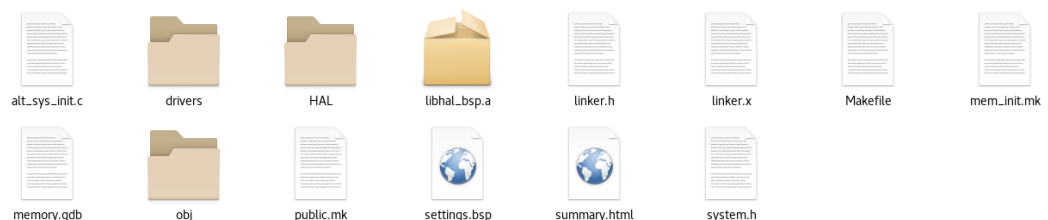


- Contains all the necessary source files and libraries to build the control application for the Nios II processor.
 - **btc_dprx_syslib**, **btc_dptxll_syslib** & **btc_dptx_syslib** are provided as encrypted libraries to handle the configuration of DisplayPort IP sink and source blocks, as well as to manage all the link training process and EDID management with external devices.
 - **dp_demo** is the software application for this example design.



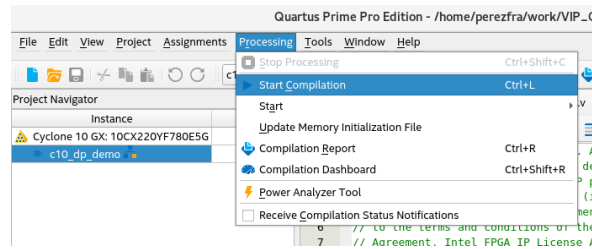
Please note that, as part of the example design generation, the software application has been already compiled and the executable generated (**dp_demo.elf**). Inside `mem_init` folder a **cpu_onchip_mem.hex** has been also auto generated to initialize the internal RAM block assigned to hold the program memory for the Nios II CPU. The generated bitstream to configure the FPGA will include already the program instructions file for the CPU, so can start execution right after FPGA configuration.

- **dp_demo_bsp** is the Nios II Board Support Package for this specific Platform Designer configuration including all the device drivers and HAL (Hardware Abstraction Layer) support



3. Generate FPGA programming file

We start the bitstream generation launching **Processing->Start Compilation** from the main toolbar.



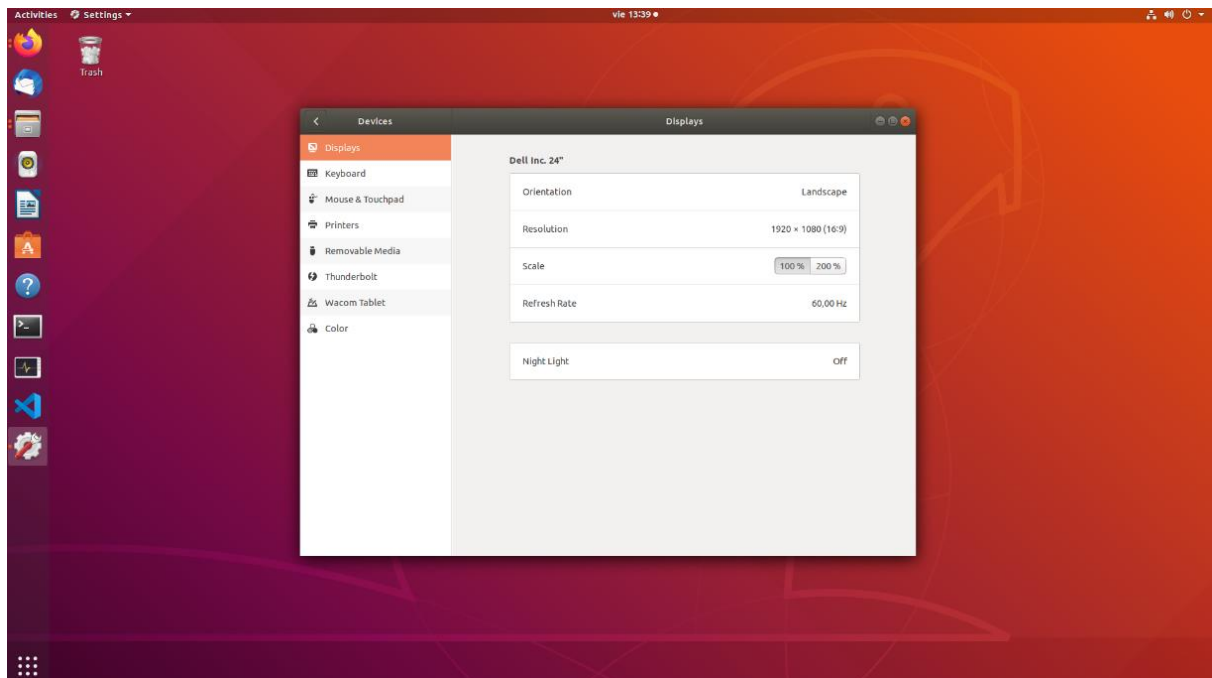
After a few minutes, depending on your machine configuration, the full compilation is done and the FPGA configuration file (**c10_dp_demo.sof**) gets generated in the *dp_0_example_design/quartus* folder.

4. FPGA configuration

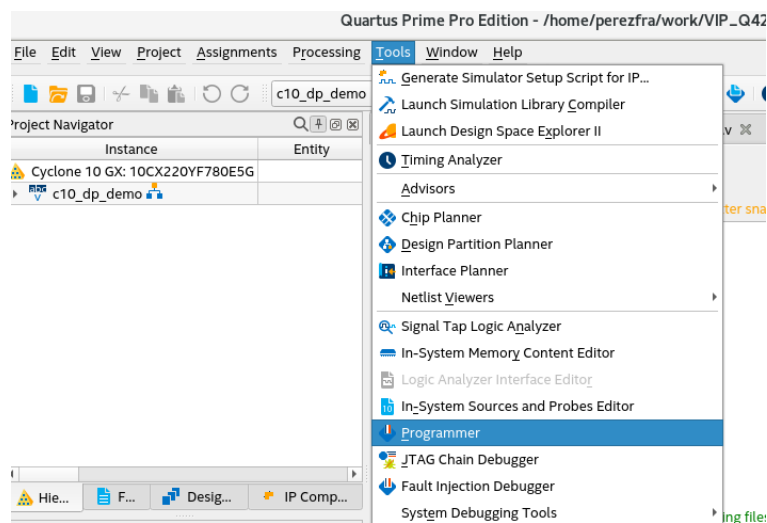
- Hardware Setup.
 - Connect the power supply, USB Blaster and DisplayPort Input/Output cables to the kit and switch it on.



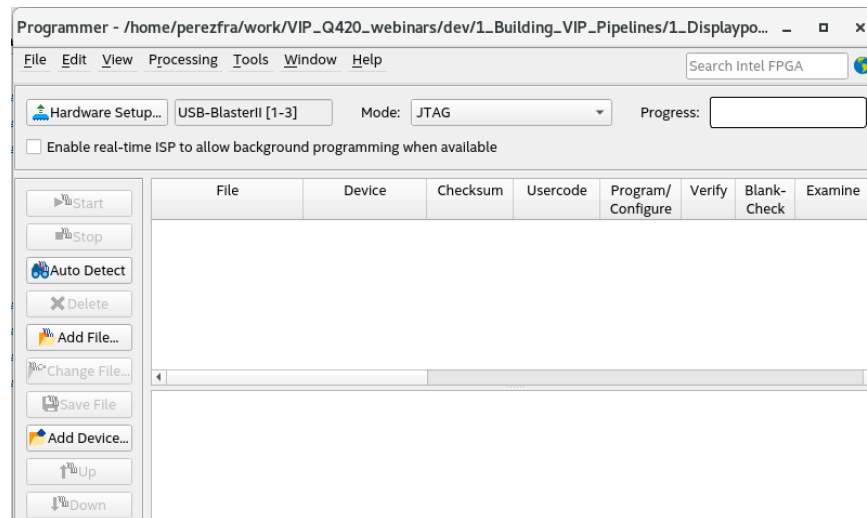
- In this demo, we are using an external PC as DisplayPort Source and a LCD Display as Sink. Please note that, before configuring the FPGA, there is only one display detected by the graphics card driver.



- Open Quartus Programmer. **Tools->Programmer**



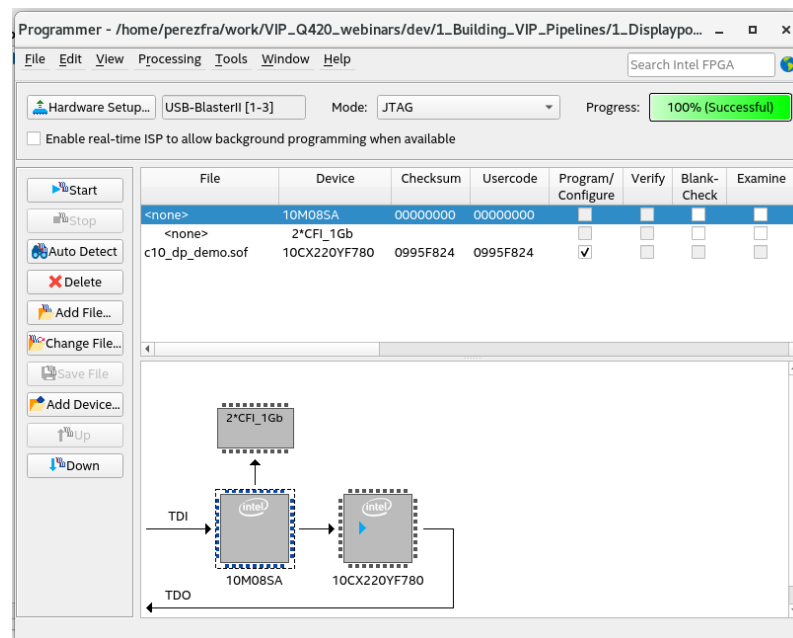
- Make sure you select the right download cable under **Hardware Setup** and click on **Auto Detect** to perform an enumeration of the different devices connected to the JTAG chain on the board.



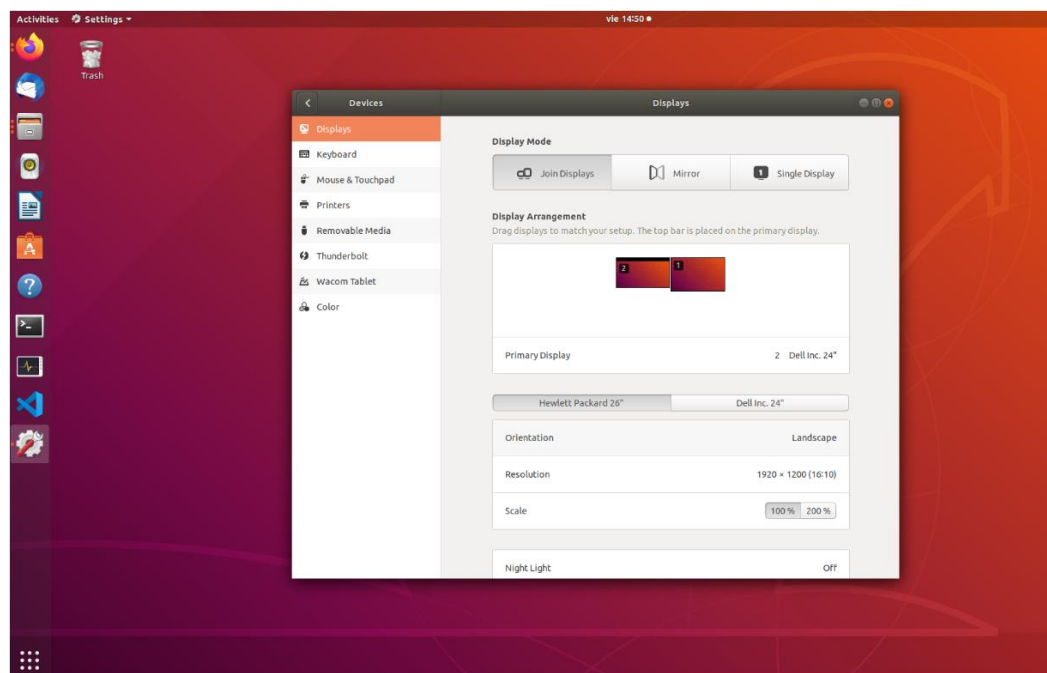
- Once **Auto Detect** has started, you will be prompted to select the right devices among different options. Select **10M08SA** and **10CX220Y** when asked. You will see the following topology on the Programmer GUI

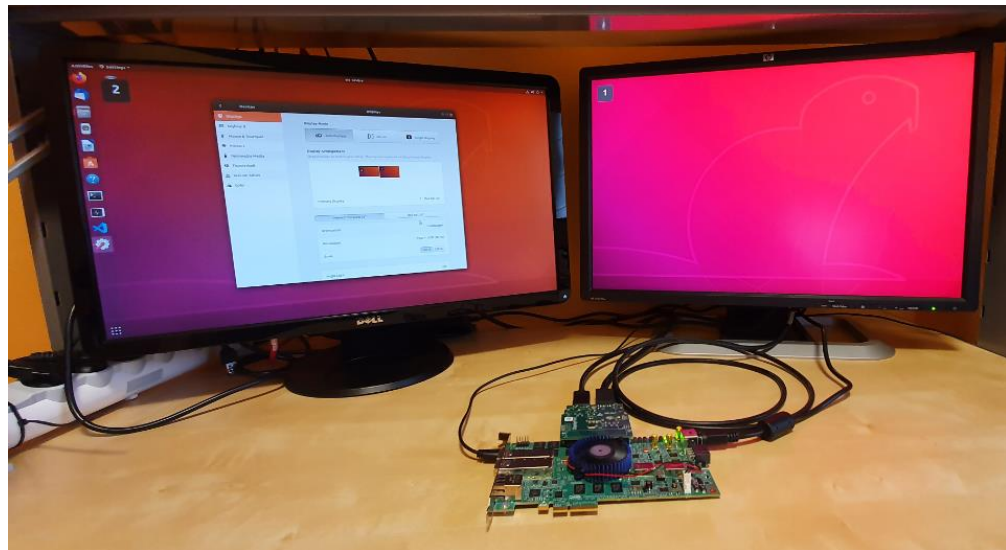


- Select the 10CX220Y device and click on **Change File** to attach the generated **.sof** file.
- Navigate to *dp_0_example_design/quartus* and select **c10_dp_demo.sof**. Enable **Program/Configure** and you are ready to go. Press **Start** to trigger configuration.



- Once the **Progress** bar reaches 100%, the design is loaded onto the FPGA and the link training procedure starts with the connected Sink and Source devices. As result you can see a secondary monitor detected attached to the DP output displaying content generated from the computer. Please note that we are using EDID Passthrough on this configuration, so the identified monitor name and resolution coincides with what we have attached to the source in the devkit. We can modify this behavior in the software application to include our proprietary EDID if needed.





In the above picture the monitor identified as “2”, is connected directly to the host PC running Ubuntu OS and the monitor “1” is driven by the FPGA devkit. The secondary output of the host PC is connected to the input on the FPGA card and the OS can detect it as an additional display. The FPGA board is just doing a loopback off the content, so showing an extended desktop here.

4.3.Modifying the Software Application

Although there is a script located in `dp_0_example_design/script/build_sw.sh` that compiles all the BSP and application to build new *.elf file, we are showing here how you can create an Eclipse project to manage your files and launch a debug session from the GUI.

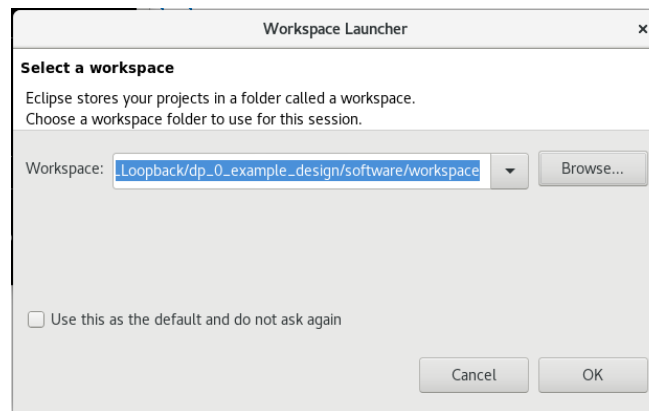
1. Creating Eclipse project for Nios II application and BSP.

Please note that the recommendations here are not mandatory and you might want to build a different setup.

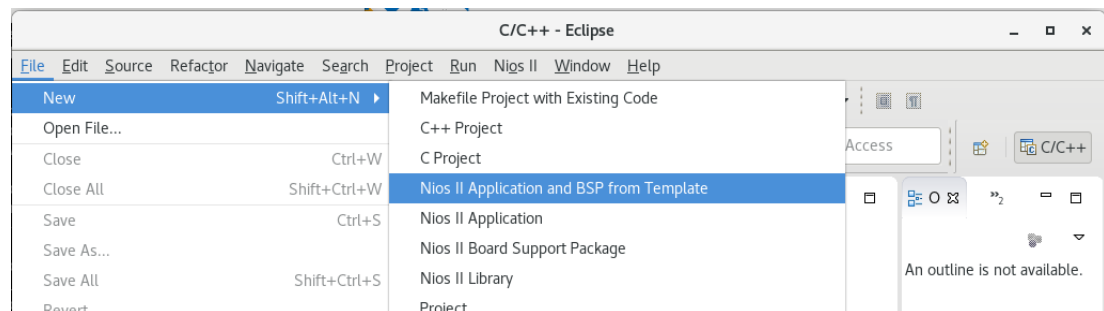
- Create a workspace folder in `dp_0_example_design/software`



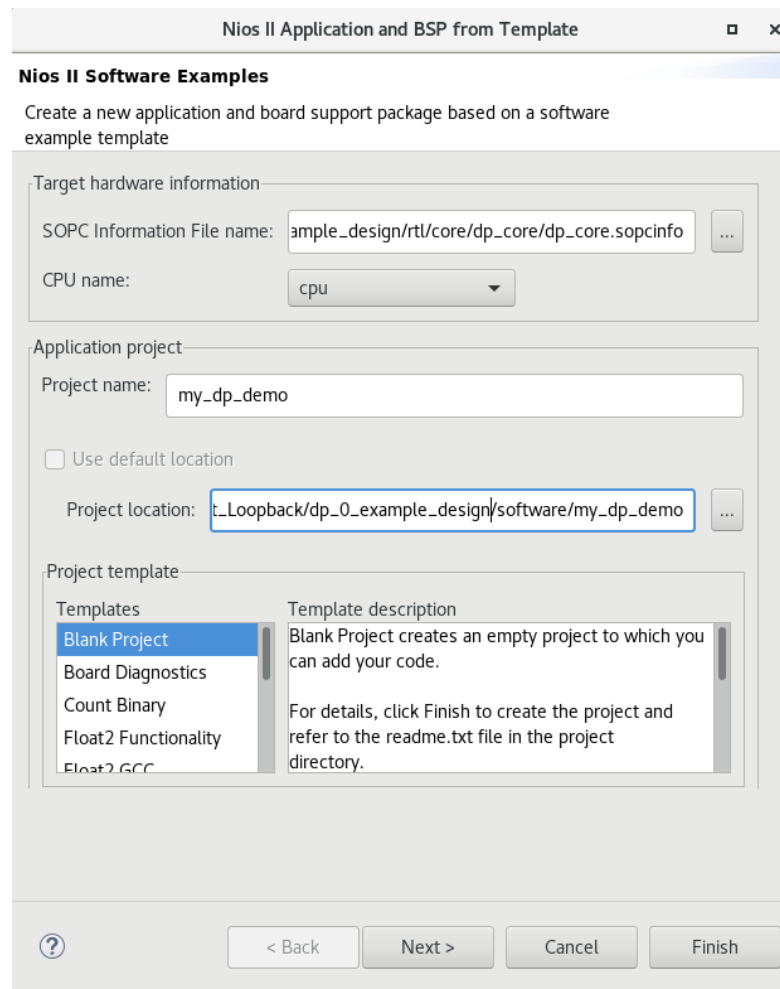
- Launch `eclipse-nios2` from your terminal with the correct environment variables. When asked for a **Workspace** for the session, select the folder you have just created (or the one of your choice)



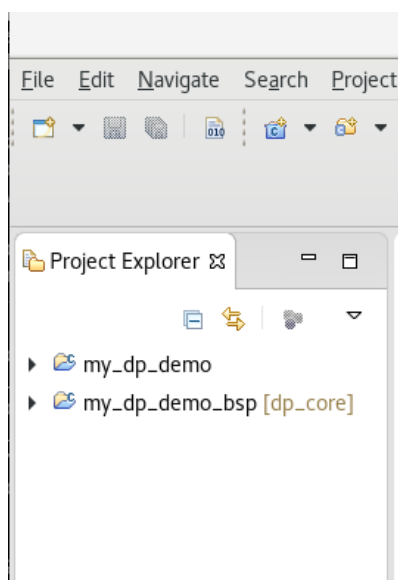
- Use **File->New->Nios II Application and BSP from Template** to create your new project



- Under **Target hardware information->SOPC Information File name**, you need to select the `*.sopcinfo` file that contains your Nios CPU. In our case is located in `dp_0_example_design/rtl/core/dp_core/dp_core.sopcinfo`
- Under **Application project->Project name** : `my_dp_demo`
- Make sure that **Project location** is set to `dp_0_example_design/software/my_dp_demo`, by default gets located at `dp_0_example_design/rtl/core/dp_core/software/my_dp_demo`
- Select **Blank Project** as **Templates** and Click **Finish**

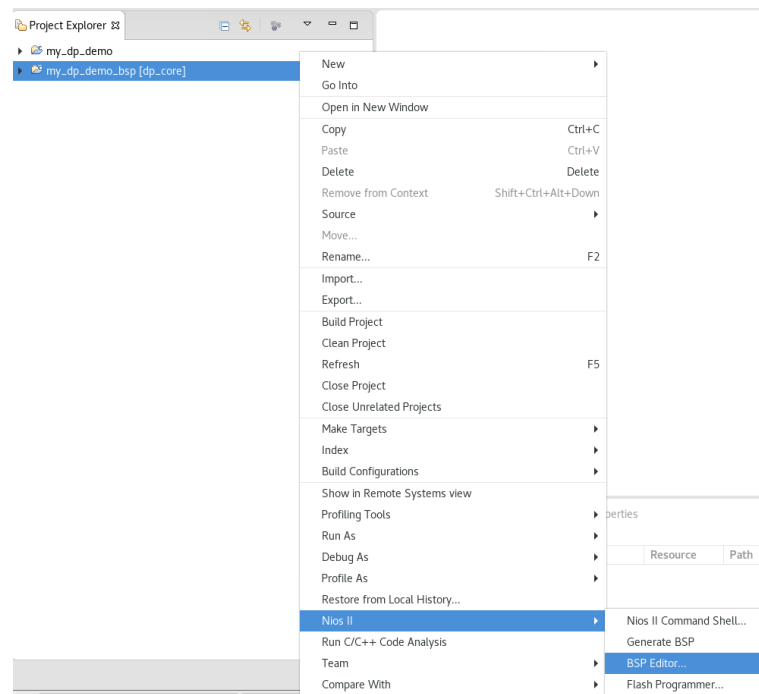


- After generation, you can see the **my_dp_demo** application and **my_dp_demo_bsp** in the **Project Explorer** tab

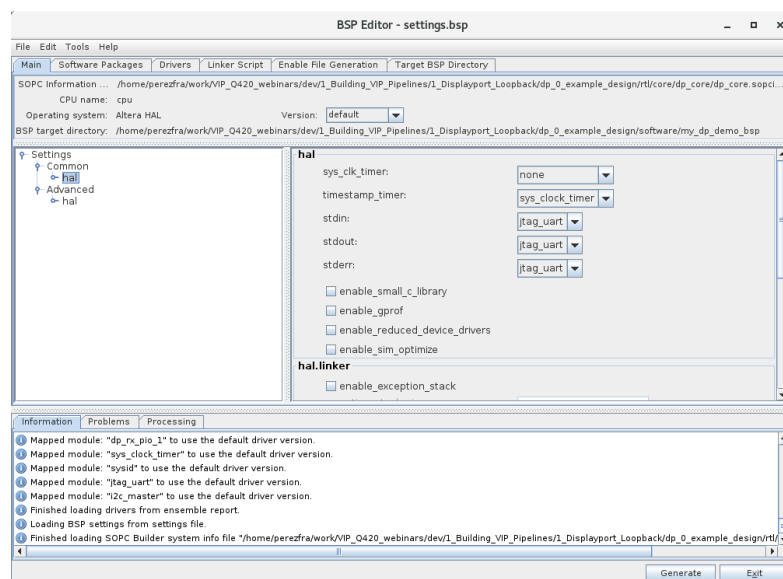


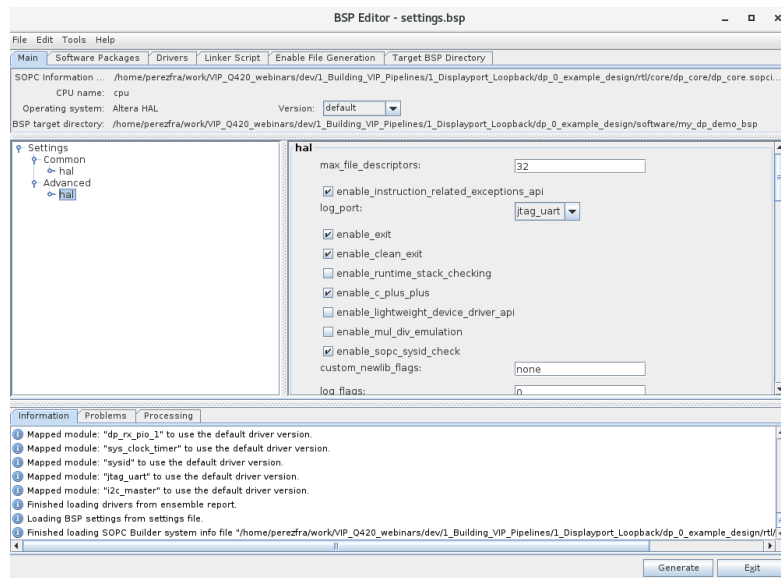
2. Adjusting the BSP settings and generating

- Next, we need to modify the `bsp` to match with our hardware implementation and enable some features.
- Right-click **my_dp_demo_bsp** and select **Nios II -> BSP Editor**



- The **BSP Editor** opens, make the following changes:
 - **Settings->Common->hal->sys_clk_timer:** none
 - **Settings->Common->hal->timestamp_timer:** sys_clock_timer
 - **Settings->Advanced->hal->log_port:** jtag_uart





- Click on **Generate** then **Exit**

3. Adding files to the application and build it

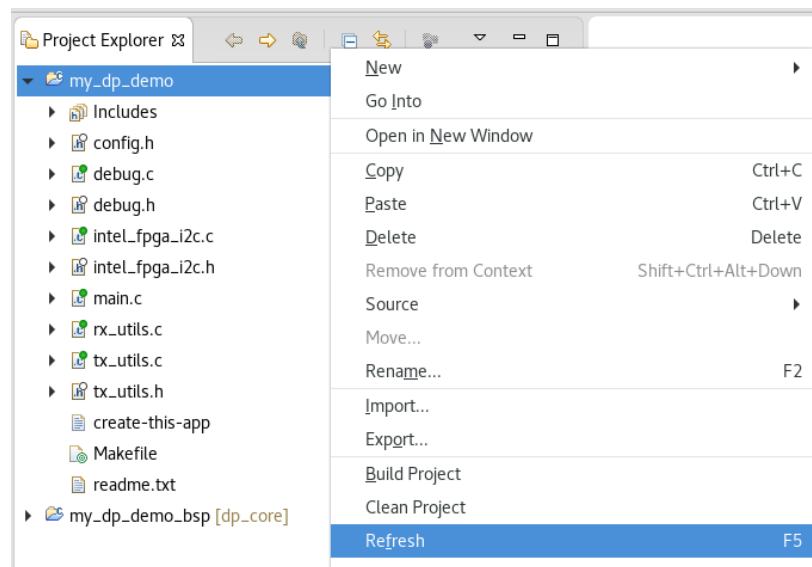
If we open now our `dp_0_example_design_folder/software`, we will find 2 new items added, corresponding to `my_dp_demo`, `my_dp_demo_bsp`



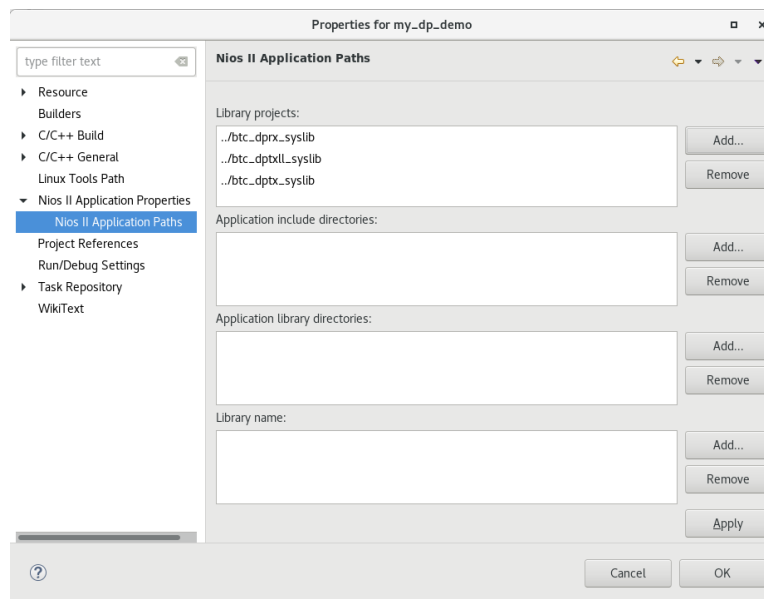
- Next, we are copying all the source files (.c, .h) from `dp_demo` >> `my_dp_demo`



- In the **Project Explorer**, right-click on **my_dp_demo** and select **Refresh**, to see the newly added files. Also, the managed *Makefile* gets updated with the changes.



- Now we need to add the encrypted DisplayPort libraries. Right-click the **my_dp_demo** application and select **Properties**. In the dialog box, go to **Nios II Application Properties->Nios II Application Paths**. Under **Library projects**, click **Add** and select *btc_dprx_syslib*, *btc_dptxll_syslib* & *btc_dptx_syslib* from the *dp_0_example_design/software* folder.



- Click **Apply & OK** to close the dialog
- Let's do a small modification to the code, to make sure we are running our compiled application.
 - o Open *main.c* in the editor and look for the **main** function
 - o Around line 141, you will find the following code

```

131
132 int main()
133 {
134     // Force non-blocking jtag uart
135     int res = 0;
136     res = fcntl(STDOUT_FILENO, F_SETFL, O_NONBLOCK);
137     res = fcntl(STDIN_FILENO, F_SETFL, O_NONBLOCK);
138     if (res == -1) {
139         printf("FCNTL Failed\n");
140     }
141     printf("Started...\n");
142 }

```

- Let's modify it by

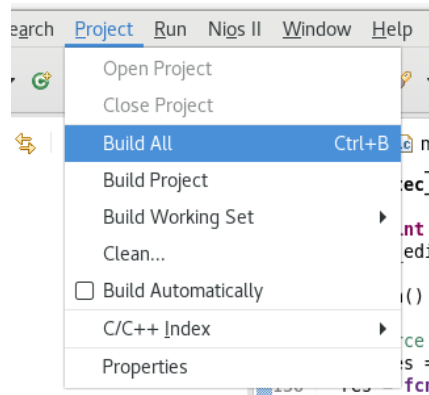
```

132 int main()
133 {
134     // Force non-blocking jtag uart
135     int res = 0;
136     res = fcntl(STDOUT_FILENO, F_SETFL, O_NONBLOCK);
137     res = fcntl(STDIN_FILENO, F_SETFL, O_NONBLOCK);
138     if (res == -1) {
139         printf("FCNTL Failed\n");
140     }
141     printf("My Demo Started...\n");
142 }

```

- Save the file

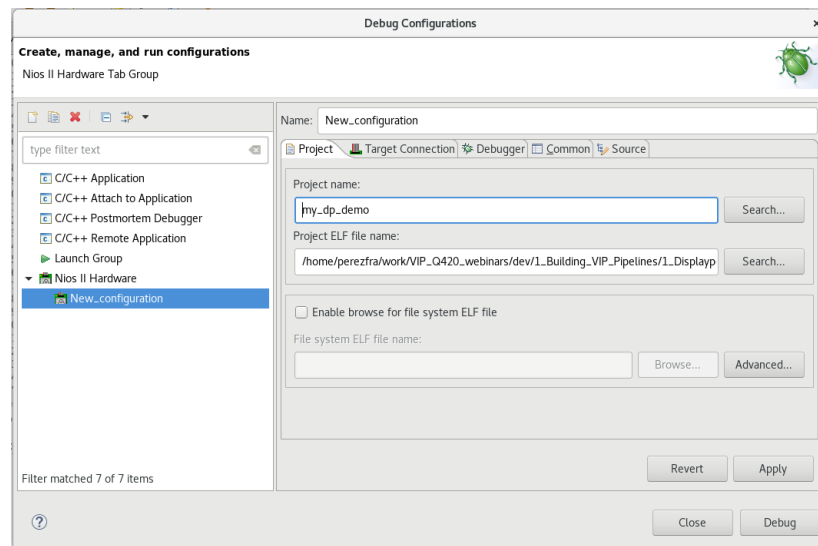
- Go to **Project->Build All** to start the BSP and application compilation



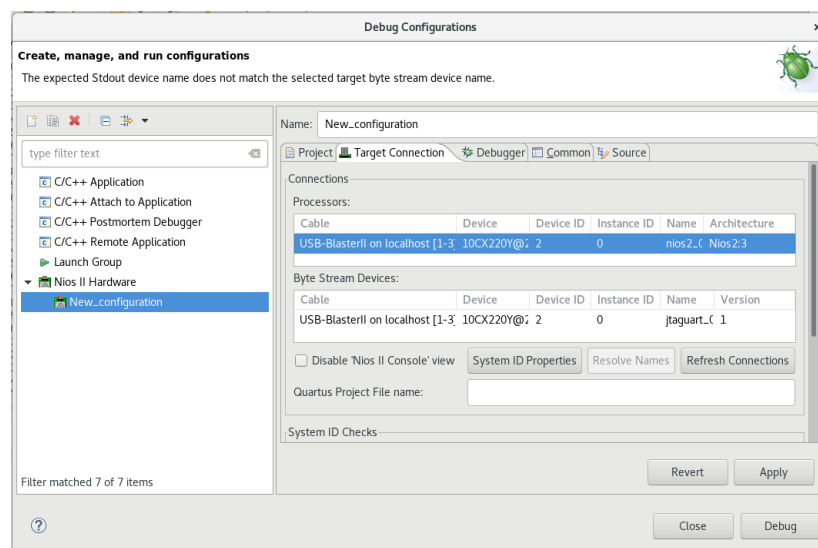
- After completion, you will see the generated executable **my_dp_demo.elf** which we will use to download and debug on the target

4. Download to target and debug the code

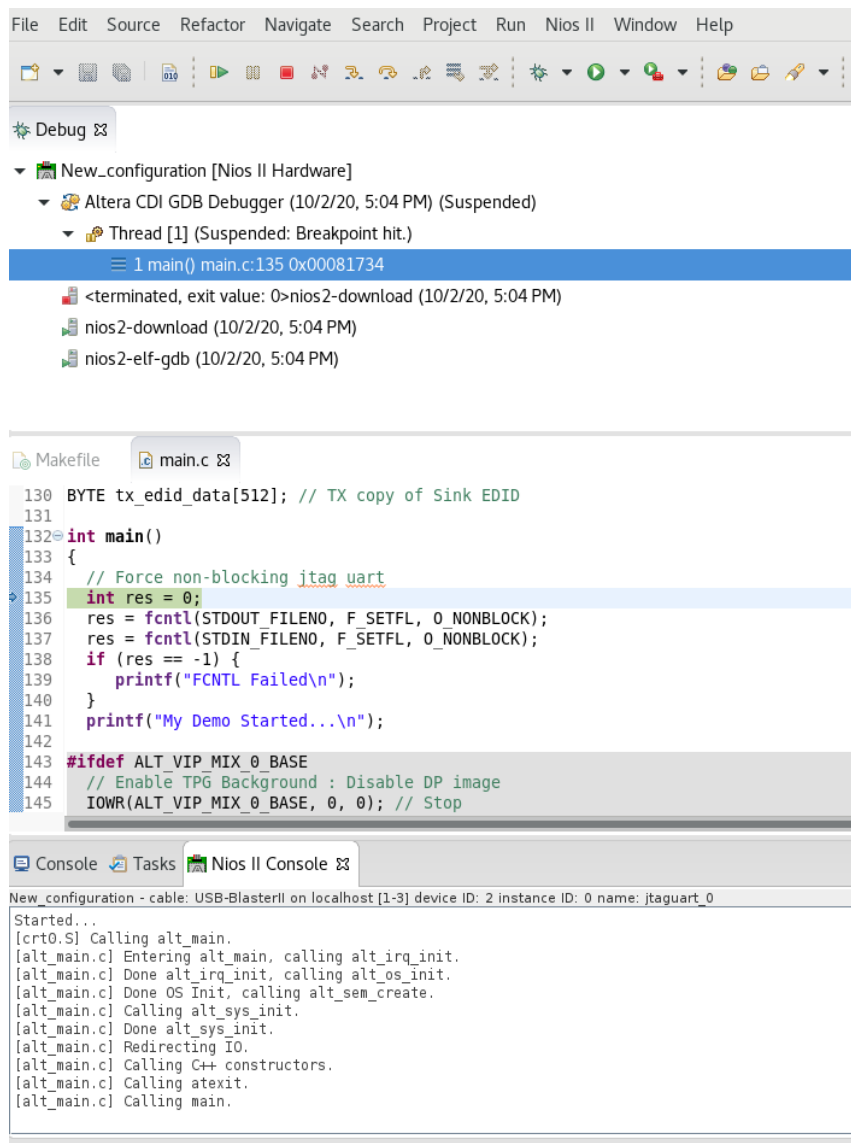
- Go to **Run->Debug Configurations** to open the dialog to manage the configurations. Double click on **Nios II Hardware** to create a *New_configuration*.
- Verify that the **Project Name** and **ELF file** are correct in the **Project** tab



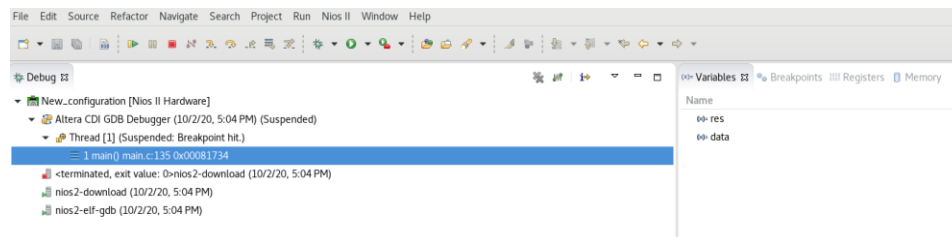
- You might want also check that, in **Target Connection** tab, you are using the right USB BlasterII adapter over the target NiosII processor.




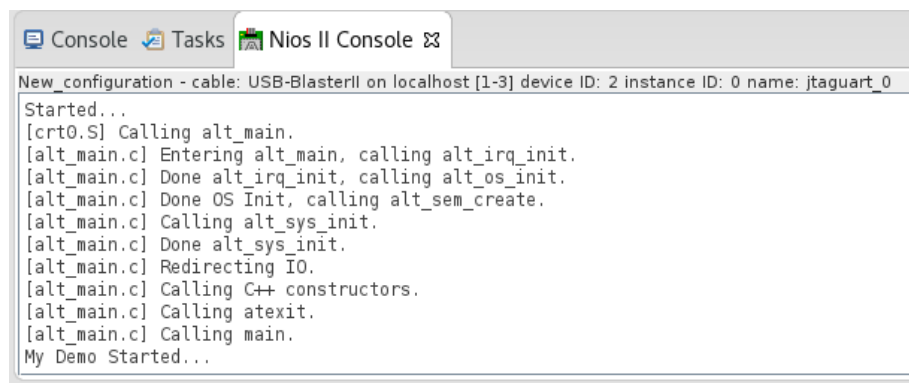
- Just click **Apply & Debug** to download the code to the target processor and start the debug session.
- The program is downloaded into the CPU program memory and initialized. The debugger stops at `main`. As we have enabled the **debug_log_port** in the **BSP (hal->Advanced)**, we see in the **Nios II Console** messages issued by the HAL in the runtime initialization.



- We can now debug our code using typical debug facilities like: resume, step into, step over, check variables, insert breakpoints, inspect registers & Memory, ...



- If we press the **Resume**  button, the program gets executed and will run the main loop (while(1)) until we stop it.
- We can verify in the **Nios II Console** that our modify version is being executed



The screenshot shows a software window titled "Nios II Console" with tabs for "Console", "Tasks", and "Nios II Console". The console output displays the initialization steps of a demo application, starting with "Started..." and ending with "My Demo Started...".

```
New_configuration - cable: USB-BlasterII on localhost [1-3] device ID: 2 instance ID: 0 name: jtaguart_0
Started...
[crt0.S] Calling alt_main.
[alt_main.c] Entering alt_main, calling alt_irq_init.
[alt_main.c] Done alt_irq_init, calling alt_os_init.
[alt_main.c] Done OS Init, calling alt_sem_create.
[alt_main.c] Calling alt_sys_init.
[alt_main.c] Done alt_sys_init.
[alt_main.c] Redirecting I/O.
[alt_main.c] Calling C++ constructors.
[alt_main.c] Calling atexit.
[alt_main.c] Calling main.
My Demo Started...
```

5. Summary

In this lab, we have built a complete working application to retransmit DisplayPort video content.

- We have generated an example design using the capabilities provided by the DisplayPort IP core for Intel FPGAs.
- We have learnt how to modify the software application running on the embedded Nios II Processor