# Imperix firmware IP product guide

PN116 | Posted on March 26, 2021 | Updated on July 24, 2025

Benoît STEINMANN
Software Team Leader
imperix • in

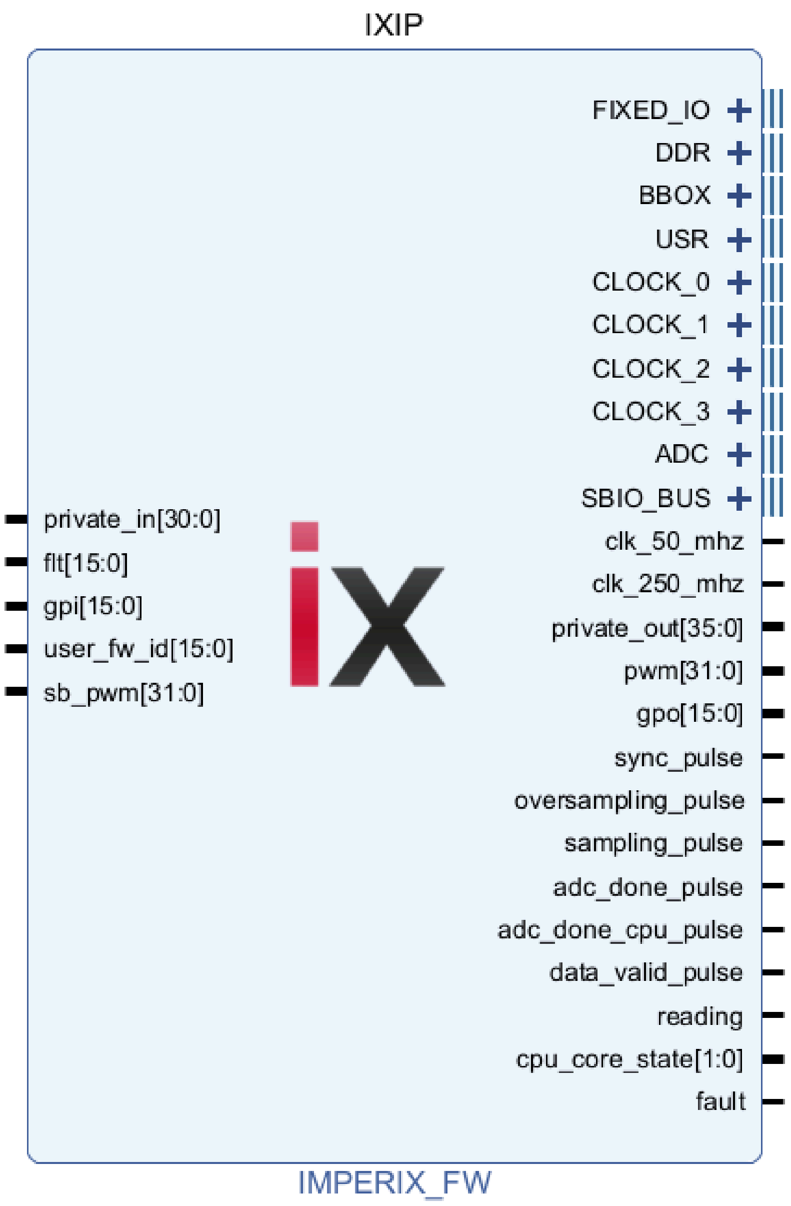
Table of Contents

This page documents the **imperix firmware IP** for Xilinx Vivado, which contains the imperix FPGA logic of the imperix controllers, namely the B-Box RCP, the B-Board PRO, the B-Box Micro and the TPI.

The **imperix firmware IP** enables the user to:

- Exchange data between the application control code running in the CPU (PS) and the logic in the FPGA (PL);
- Drive the PWM output chain, comprised of a dead-time generation system and the hardware protection mechanisms;

- Retrieve ADC conversion results as soon as they are available, directly from within the FPGA;
- Have direct access to physical I/Os pins such as USR, GPI and GPO pins.



**Prerequisite to use the imperix firmware IP**

- The getting started with FPGA page explains how to use the imperix IP in a Vivado project
- It is required to install Xilinx Vivado to use the imperix firmware IP
- To exchange data with the FPGA and configure the PWM outputs, the necessary drivers are included in ACG SDK (graphical programming) and CPP SDK (C++ programming)

The imperix firmware IP is available for download on the imperix IP download page.

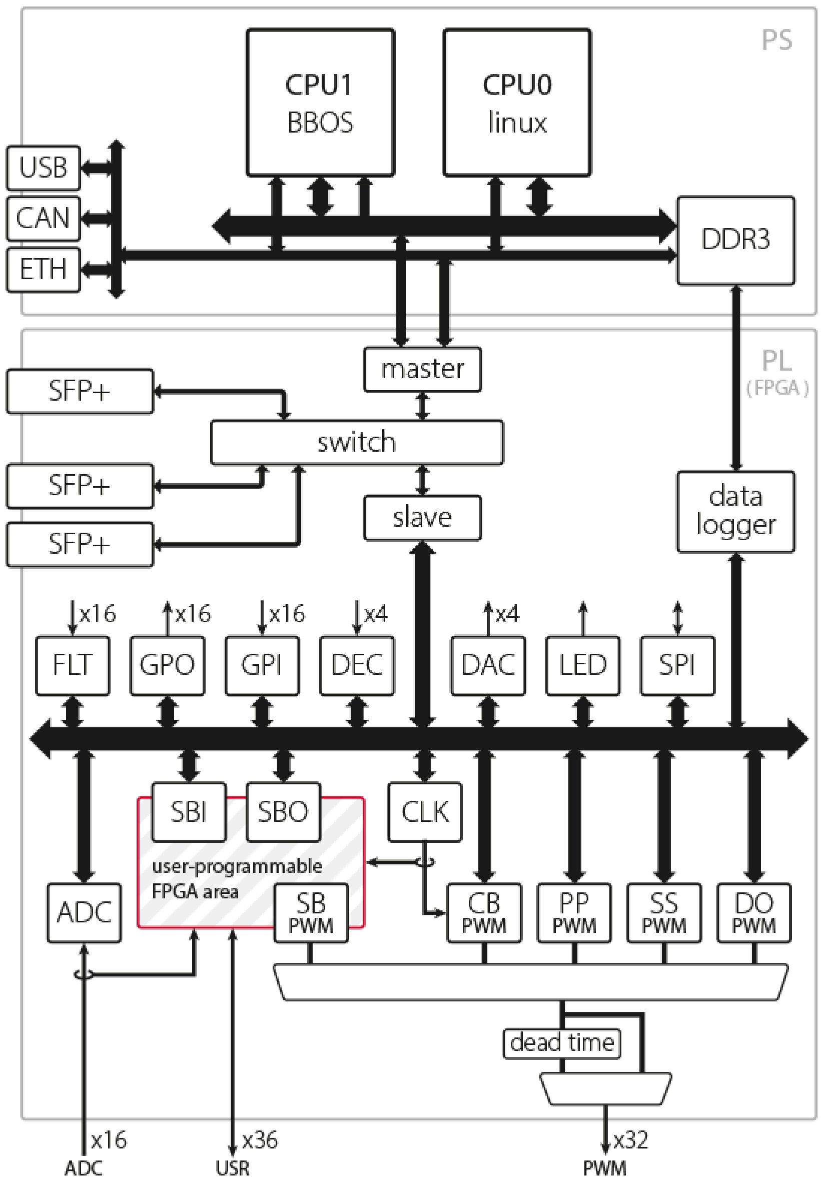
To find all FPGA-related notes, you can visit FPGA development homepage.

## The imperix firmware IP

The B-Board is based on a Xilinx Zynq XC7Z030 System-on-Chip which consists of a Processing System area (PS) with two CPU cores and a Programmable Logic (PL) area.

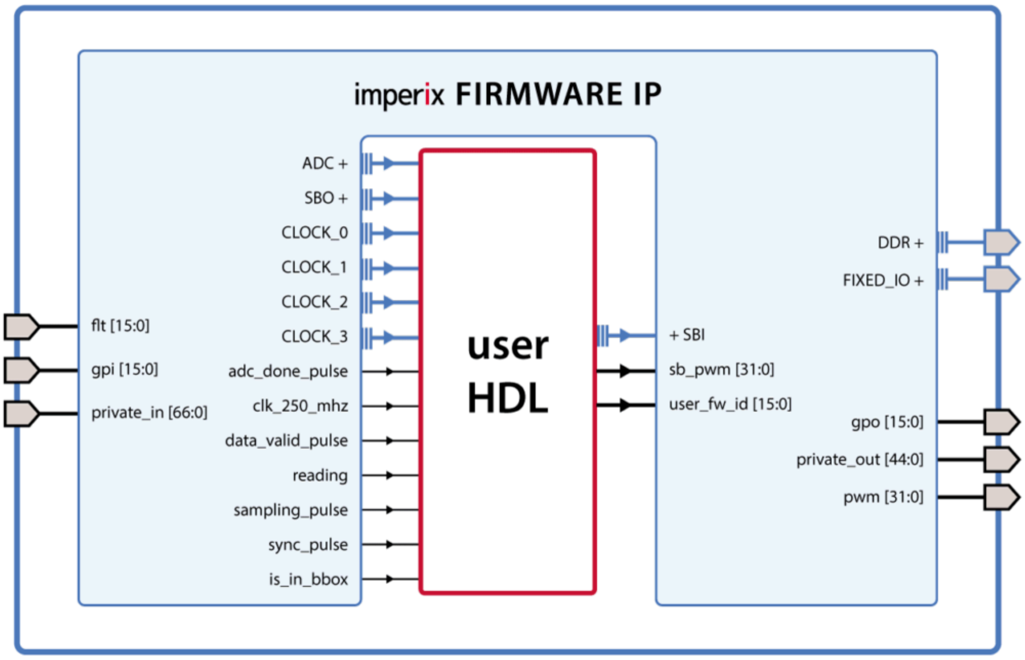
- CPU0: Running on Linux, the first core is responsible for loading the application code, supervising the system execution and managing the data logging.
- CPU1: Running on BBOS (lightweight secured proprietary operating system), the second core executes the application-level control code developed by the user.
- FPGA: The PL contains all the application-specific peripherals.

The fixed part of the FPGA firmware containing the pre-implemented peripherals is packaged in the **imperix firmware IP**. This IP also provides interfaces to the user as illustrated by the *user-programmable FPGA area* on the image.



The imperix firmware IP is designed such that it provides interfaces for implementing special **user HDL** logic in the *so-called* **sandbox**. It gives access to some of the I/Os (ADC, PWM,…) as well as ways to exchange data with the CPU (SBI, SBO).

The content of the imperix firmware IP is obfuscated and cannot be either read or edited.



## IP port descriptions

This section describes the ports of the imperix firmware IP.

## Digital I/Os drivers

These ports drive the physical-digital I/Os of the B-Board. The description of the peripherals driving these ports and the physical location of their I/Os are available in the B-Board datasheet.

| Name | Direction | Clock domain | Description |
|---|---|---|---|
| `flt[15:0]` | input | `clk_250_mhz` | Fault inputs |
| `gpi[15:0]` | input | `clk_250_mhz` | General-purpose inputs |
| `gpo[15:0]` | output | `clk_250_mhz` | General-purpose outputs |
| `pwm[31:0]` | output | `clk_250_mhz` | Pulse-width modulated outputs |

| | | | |
|---|---|---|---|
| `BBOX[51:0]` | tristate input/output | `N/A` | These pins are used internally in the B-Box RCP. They can be re-purposed when using the B-Board PRO on a custom-made PCB design, as explained in PN201. |
| `USR[35:0]` | tristate input/output | `N/A` | These 3.3V pins are made available to the user to interface external peripherals, such as an SPI ADC. They are available from the B-Board PRO Eval board and from the B-Box RCP VHDCI connector B. To use these pins, please refer to the Getting Started with FPGA page. |

## Private ports

| **Name** |
|---|
| `private_in` |
| `private_out` |
| `DDR` |
| `FIXED_IO` |

These ports are necessary to communicate with the various components on the B-Board. Modifying these connections could alter the proper behavior of the device.

## User interfaces

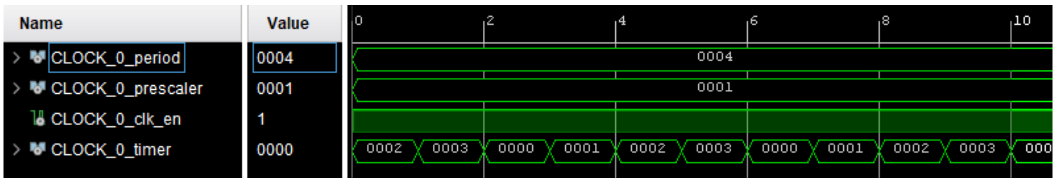
The user interfaces include data ports and timing signals to exchange data with the CPU, drive PWM outputs, and access ADC value.

| **Name** | **Direction** | **Clock domain** | **Description** |
|---|---|---|---|

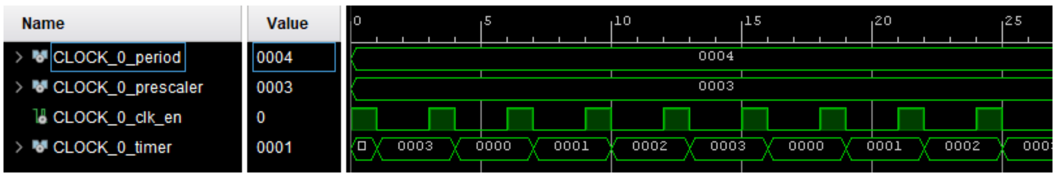
| | | | |
|---|---|---|---|
| `ADC_reg_00[15:00]` to `ADC_reg_15[15:00]` | output | `clk_250_mhz` | ADC 16-bit result in 2's complement format. If in a B-Box, registers 00 to 15 hold the results the B-Box (external) ADCs. Otherwise, registers 00 to 07 hold the results of the B-Board on-board ADCs and registers 08 to 15 return zero. |
| `SBIO_BUS` | bidirectional | `clk_250_mhz` | 16-bit memory-mapped bus allowing the CPU to addressing up to 1024 registers in the FPGA. For more details on SBIO_BUS, see the Getting Started with FPGA page. |
| `CLOCK_0_period[15:00]` to `CLOCK_3_period[15:00]` | output | `clk_250_mhz` | Indicates the timer period of the CLOCK. |
| `CLOCK_0_prescaler[15:00]` to `CLOCK_3_prescaler[15:00]` | output | `clk_250_mhz` | Indicates the prescaler division value of the CLOCK. |
| `CLOCK_0_clk_en` to `CLOCK_3_clk_en` | output | `clk_250_mhz` | Clock enable generated by the prescaler, provides the counting rate of `CLOCK_N_timer`. `CLOCK_N_clk_en` is asserted one period out of `CLOCK_N_prescaler`. |
| `CLOCK_0_timer[15:00]` to `CLOCK_3_timer[15:00]` | output | `clk_250_mhz` | Timer counting from 0 to `CLOCK_N_period` at `CLOCK_N_clk_en` rate. |

| | | | |
|---|---|---|---|
| `reading` | output | `clk_250_mhz` | Asserted for the duration of the READ phase, signaling that `SBI` registers are being read and transferred towards the CPU. |
| `sampling_pulse` | output | `clk_250_mhz` | Asserted for one clock period at each ADC sampling instant. |
| `sync_pulse` | output | `clk_250_mhz` | Asserted for one clock period at the end of the configuration phase (end of system startup sequence). It may be used as a synchronous reset. In the case where multiple B-Boards (B-Boxes) are connected, this pulse occurs simultaneously in all devices and can be used to synchronize counters. |
| `user_fw_id[15:00]` | input | `clk_250_mhz` | User firmware identification number. When a customized firmware is loaded, this number is sent to Cockpit and is available from the Target config window in the FPGA bitstream section. It can be used to ensure that the correct firmware has been loaded. |

The images below illustrate the behavior of the CLOCK interfaces.



Behavior of the CLOCK interface when prescaler=1



Behavior of the CLOCK interface when prescaler /= 1

## Understanding the real-time behavior of the firmware

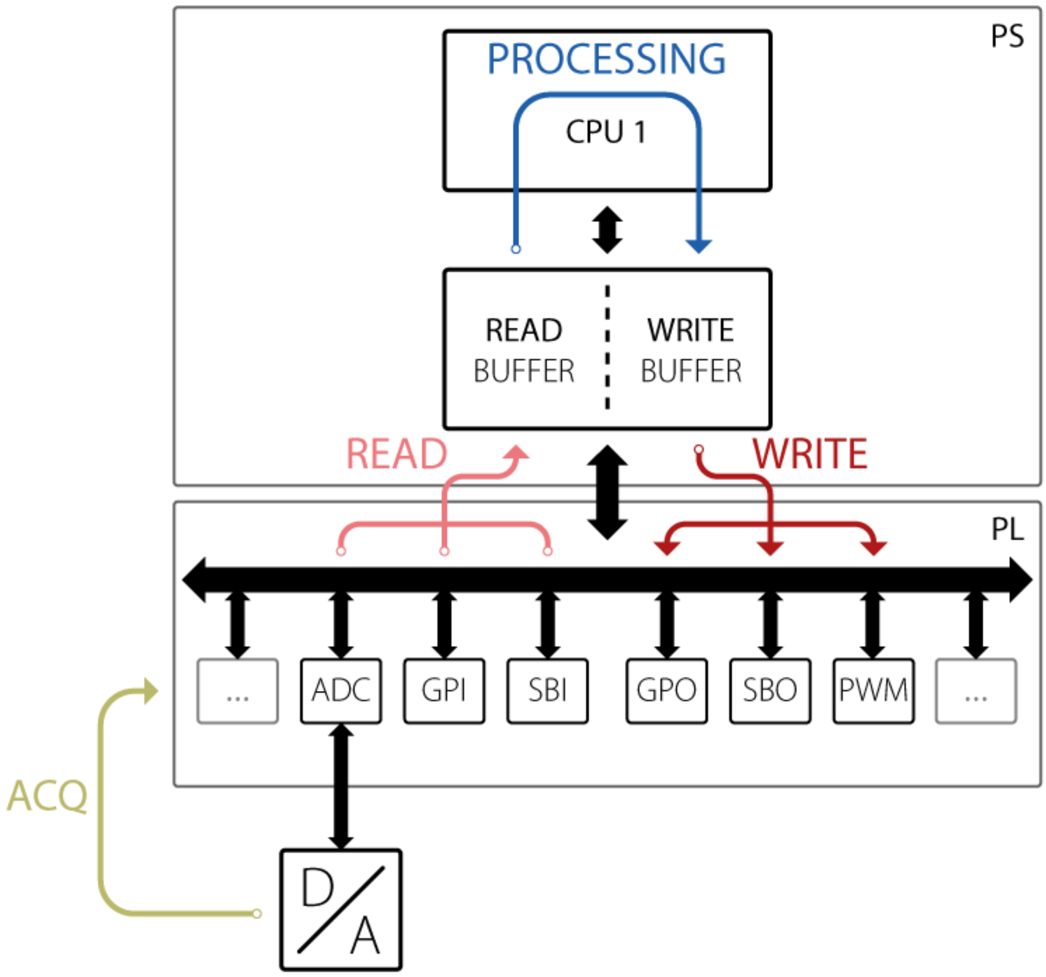
A proper understanding of the system behavior during real-time execution is necessary to correctly develop and integrate custom logic in the B-Board FPGA.

## The four phases of a control cycle

The next figure is a simplified representation of the elements involved in the real-time execution of the control algorithm on an imperix controller. It shows the four phases constituting a control cycle, which are:

- **ACQ**: Analog-to-digital conversion (physical chips) and results acquisition (transfer to FPGA);
- **READ**: Real-time DMA read, FPGA data flagged as real-time are sent to the *CPU read buffer*;
- **PROCESSING**: Execution by the CPU of the main control task, processing the user control code. At the beginning of this phase, the *CPU read buffer* is read. At the end, the *CPU write buffer* is updated.
- **WRITE**: Real-time DMA write, data flagged as real-time are transferred from the *CPU write buffer* to the FPGA peripherals (across the network of multiple B-Boards if needed).

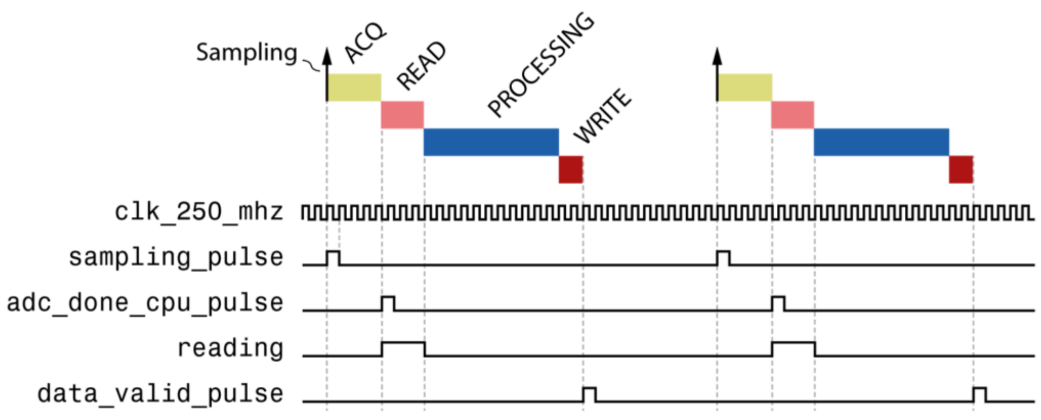
To learn more about the various delays involved when running an imperix digital controller please read the discrete control delay identification page.



The 4 phases of an imperix controller control cycle

The imperix firmware IP provide timing information, as shown in the next figure. The `_pulse` signals are asserted only for one period of `clk_250_mhz`.

- The **`sampling_pulse`** signal indicates the ADC sampling instant and the start of the **ACQ** phase.
- The **`adc_done_pulse`** signal indicates that the ACQ phases finished and new ADC values are available in the `ADC` registers.
- The **`reading`** signal indicates the start of the **READ** phase when `SBI` registers flagged as real-time are sent to the CPU memory.
- At the end of the **WRITE** phase **`data_valid_pulse`** is asserted notifying that new data is available in the real-time `SBO` registers.



Timing information with no oversampling

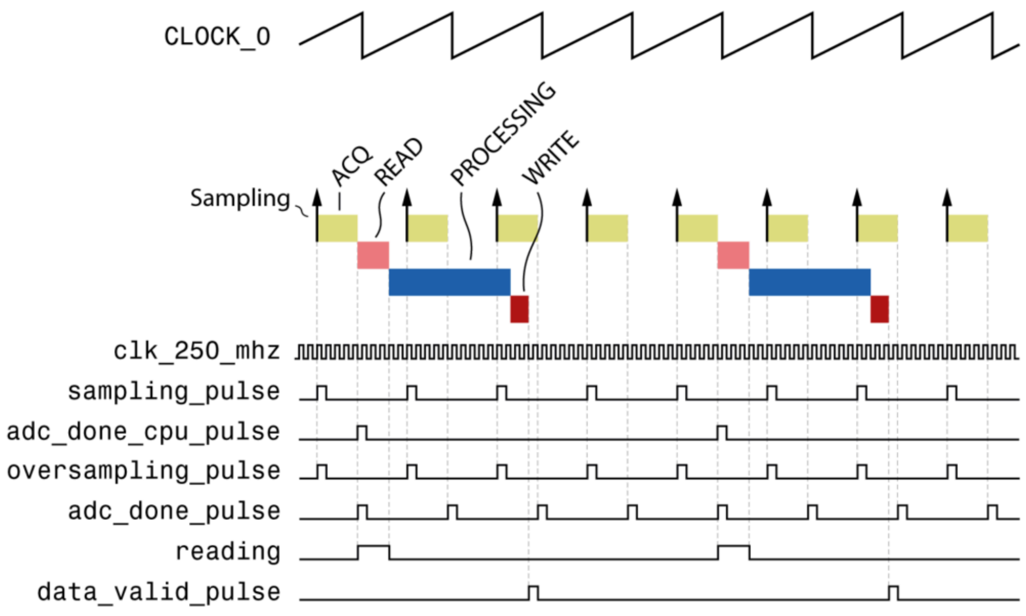
## Executing the FPGA-based algorithm faster than the CPU

The ADC sampling is always linked to CLOCK_0 and can be configured using the Configuration block. The control algorithm is generally executed after each new ADC sample, so for the FPGA-based algorithm to run faster the sampling rate must be increased. It may be done by:

- **Increasing the frequency of CLOCK_0 (recommended)**
- Setting an oversampling ratio (number of sampled per CLOCK_0 period)
- A combination of both

### Increasing the frequency of CLOCK_0

Increasing the frequency of CLOCK_0 is the recommended method to increase the ADC sampling rate. Because the CPU is usually not able to run as fast as the FPGA, using the **CPU postscaler** can be used to reduce the CPU control task rate. The CPU postscaler can be configured in the Configuration block.

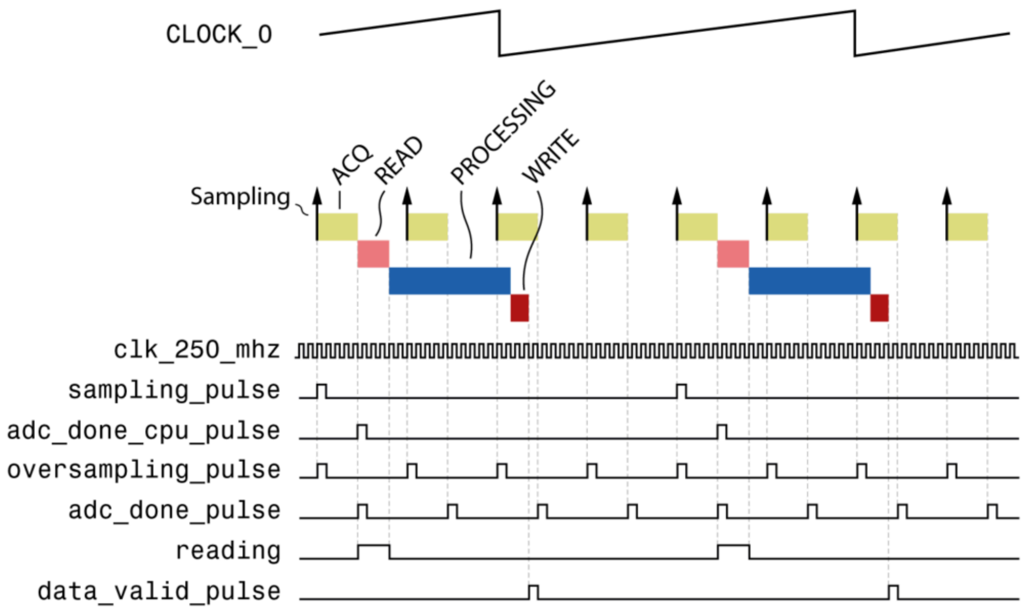


Increased CLOCK_0 frequency, CPU postscaler = 4

**Setting an oversampling ratio**

Alternatively, an oversampling ratio may be set as illustrated below.

Make sure the new synchronous averaging option is disabled in the ADC blocks **when using an oversampling ratio**. ADC channels with synchronous averaging enabled only output a new value once per CLOCK_0 period.



Oversampling = 4

## Exchanging data between the CPU and the FPGA

The 16-bit *Output towards the SandBox* (`SBI`) and *Input from the SandBox* (`SBO`) registers allow data exchange between the user control code in the CPU and the user logic in the FPGA. These registers can be read/written using their respective blocks (SBO and SBI).

The getting started with FPGA development page goes into further details on how to exchange data between the CPU and the FPGA through a step-by-step example.

## Configuration phase

The *configuration phase* occurs only once at the system startup and aims to initiate FPGA-based peripherals and configure the real-time data traffic, possibly across the control network. During this phase, interrupts and PWM operations are not yet active.

With the ACG SDK, `SBO` registers that are only written once are called *configuration registers,* opposing to *real-time registers*. The user selects which registers are set as *configuration registers* from the “Registers” tab of the SBO block and then sets up their values from the “Configuration reg. values” tab. When using C++ SDK, the `SBI`/`SBO` registers can be read/written during the *configuration phase* by using `Sbi_ReadDirectly` or `Sbo_WriteDirectly` in the `UserInit()` function.

As their names indicate, the `Sbi_ReadDirectly` and `Sbo_WriteDirectly` functions infer immediate data transfers, which are rather time-consuming *performance-wise* and, therefore, cannot be used during real-time execution.

## Real-time execution

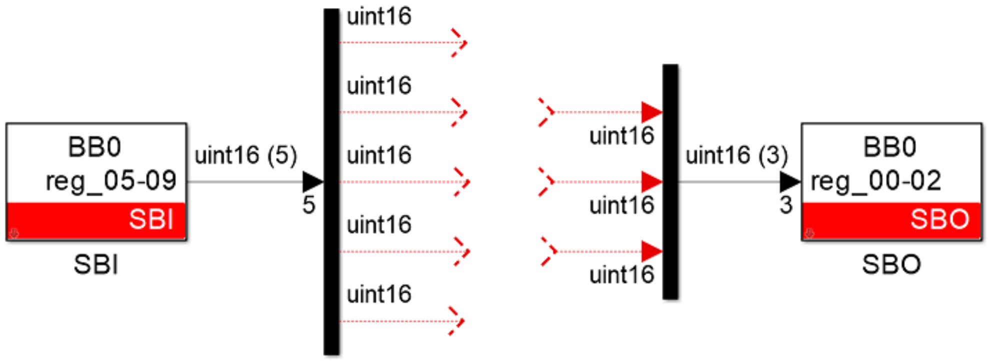
Configuring a `SBI` or `SBO` register as a *real-time register* indicates that the value of this register is to be transferred at each control task execution. Configuring a register as real-time is done by using the C++ function `Sbi_ConfigureAsRealTime` or `Sbo_ConfigureAsRealTime`.

> *In Simulink, real-time registers are clearly distinguished from configuration registers. They take their run-time values from the block inputs.*

Before entering the control interrupt, the data that have previously been configured as real-time are retrieved from the FPGA peripherals and placed into the CPU *read buffer* (**READ** phase). As such, they can readily be used from inside the control interrupt using `Sbi_Read`.

Reciprocally, using the `Sbo_Write` function inside the control interrupt stores data into the *write buffer,* waiting to be sent back to the FPGA once the interrupt is completed (**WRITE** phase).

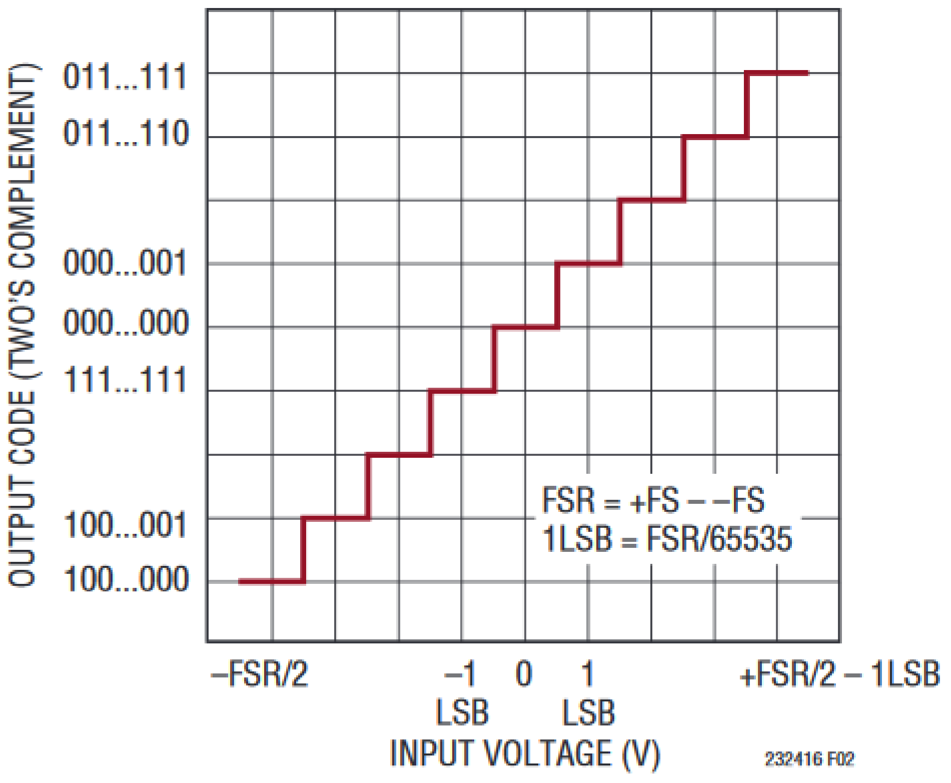
*In Simulink real-time data can be easily manipulated using signal wires.*



Simulink blocks for sandbox interfacing

## Retrieving ADC conversion results

Inside the FPGA, ADC results are available from the `ADC` interface of the firmware IP. The port `sampling_pulse` indicates the start of an ADC conversion and `adc_done_pulse` indicates that the results are available in the `ADC` interface. The conversion 16-bit results are in 2's complement format.

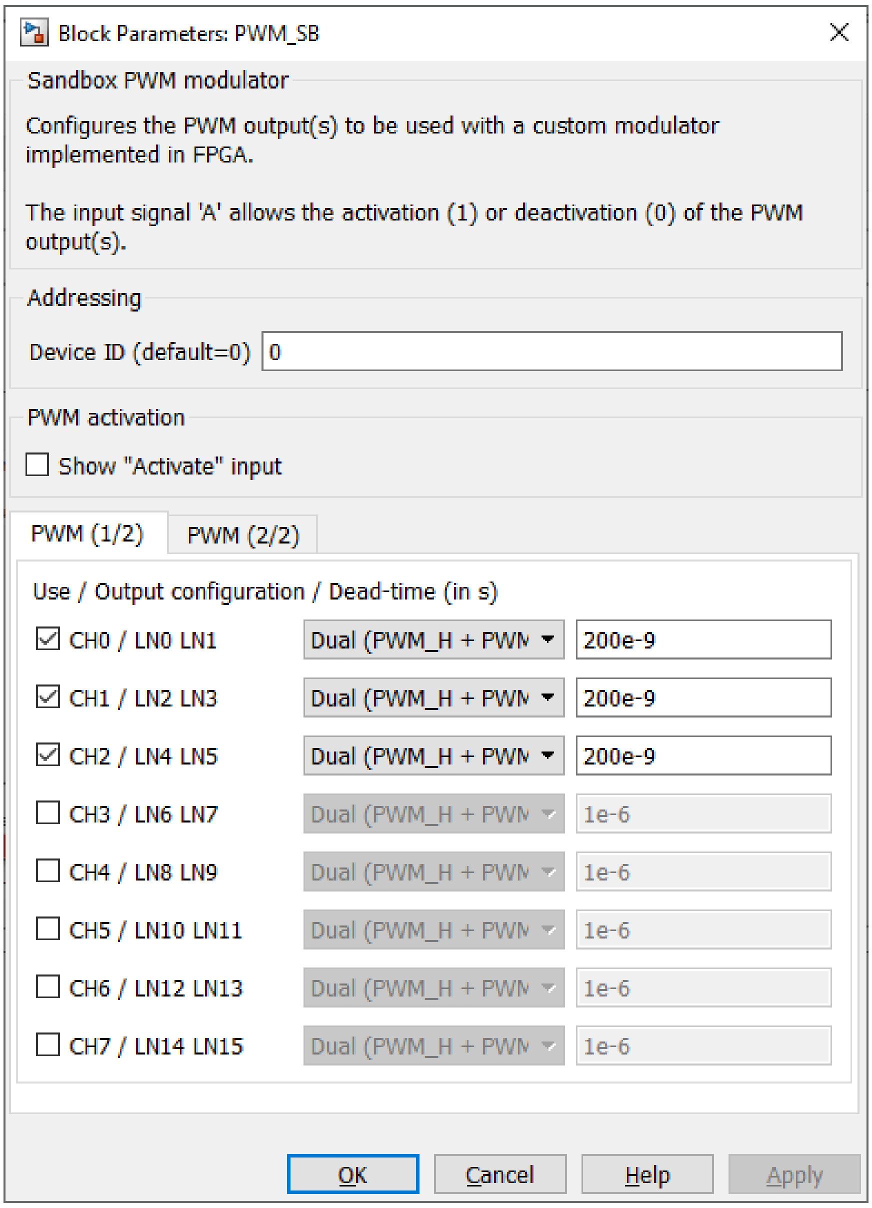


More information regarding the hardware-level implementation is available in:
– The datasheets of B-Box RCP or B-Box Micro for laboratory devices.
– The datasheet of the B-Board PRO or PN201 for embedded systems.

## Driving PWM outputs

On the firmware IP, the `sb_pwm[31:0]` port provides access to the same PWM output chain as that used by other modulators (CB-PWM, PP-PWM, DO-PWM and SS-PWM). This allows the user to generate complementary signals with dead-time, use the standard activate and deactivate functions and rely on the protection mechanism that blocks PWM outputs when a fault is detected. A use example is available on the page Custom PWM modulator implementation in FPGA.

The PWM chain can be configured using the Sandbox-PWM (SB-PWM) block. An example is shown below.



Block Parameters: PWM_SB

Sandbox PWM modulator

Configures the PWM output(s) to be used with a custom modulator implemented in FPGA.

The input signal 'A' allows the activation (1) or deactivation (0) of the PWM output(s).

Addressing

Device ID (default=0): 0

PWM activation

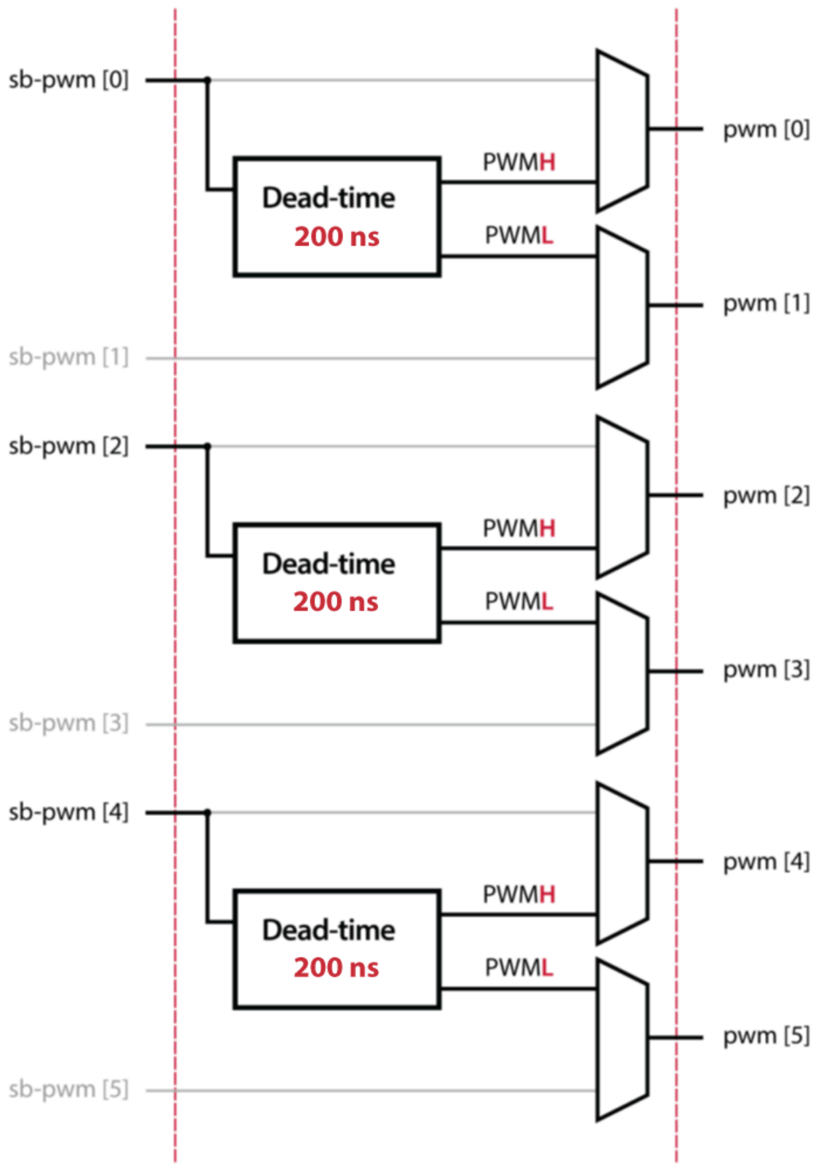
☐ Show "Activate" input

PWM (1/2) | PWM (2/2)

Use / Output configuration / Dead-time (in s)

| | | |
|---|---|---|
| ☑ CH0 / LN0 LN1 | Dual (PWM_H + PWM | 200e-9 |
| ☑ CH1 / LN2 LN3 | Dual (PWM_H + PWM | 200e-9 |
| ☑ CH2 / LN4 LN5 | Dual (PWM_H + PWM | 200e-9 |
| ☐ CH3 / LN6 LN7 | Dual (PWM_H + PWM | 1e-6 |
| ☐ CH4 / LN8 LN9 | Dual (PWM_H + PWM | 1e-6 |
| ☐ CH5 / LN10 LN11 | Dual (PWM_H + PWM | 1e-6 |
| ☐ CH6 / LN12 LN13 | Dual (PWM_H + PWM | 1e-6 |
| ☐ CH7 / LN14 LN15 | Dual (PWM_H + PWM | 1e-6 |

Configuration of the SB-PWM block in Simulink



Mapping between sb_pwm and pwm ports of the imperix IP in Vivado

Each bit of the `sb_pwm[31:0]` IP port corresponds to a PWM lane.

- `sb_pwm[0]` : lane #0
- `sb_pwm[1]` : lane #1
- `sb_pwm[2]` : lane #2
- etc.

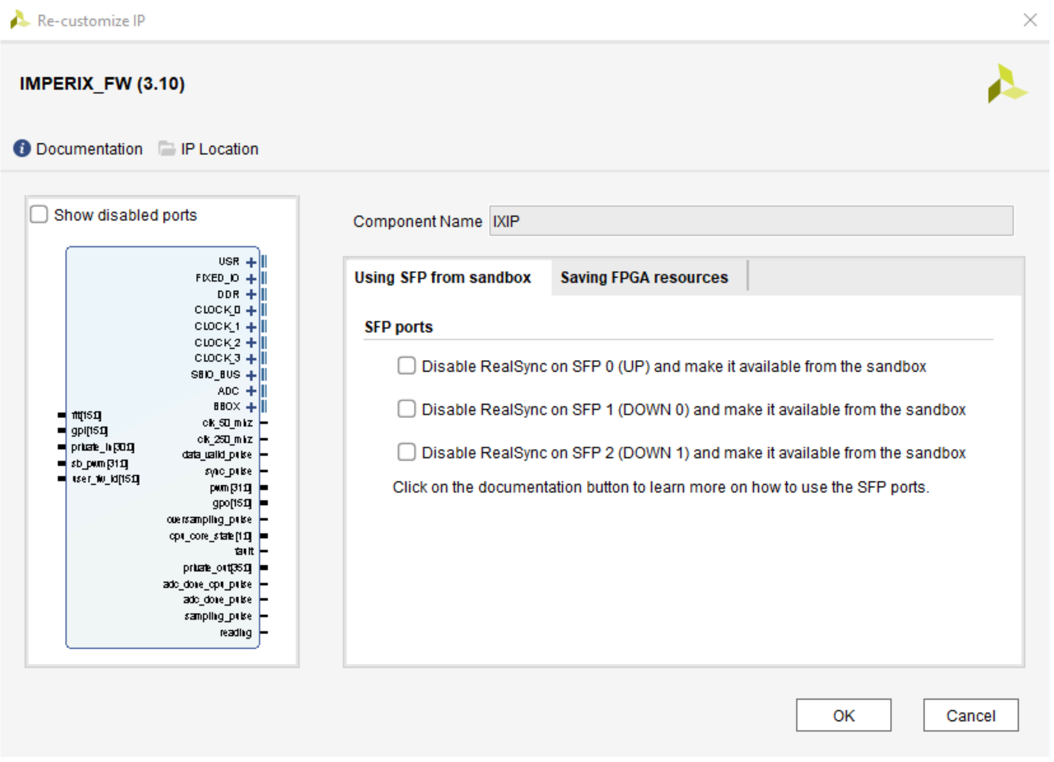
In a channel configuration (pseudo-complementary signals with dead time), the user only needs to generate the HIGH signal, which must be connected to the appropriate `sb_pwm` input (`sb_pwm[0]`, `sb_pwm[2]`, `sb_pwm[4]`, etc.). An example of such a configuration is available in TN120.
Imperix **strongly discourages** the user to directly driving the top-level `pwm` port, as this would bypass the enable/disable mechanism! Instead, the SB-PWM port is meant to provide proper access to PWM outputs, which should be used in all cases.
On B-Box RCP, this is only relatively sensitive as hardware protections exist.

However, on B-Board PRO, this is critical since this mechanism also handles the **fault management** !
More information regarding the hardware-level implementation is available in:
– The datasheets of B-Box RCP or B-Box Micro for laboratory devices.
– The datasheet of the B-Board PRO or PN201 for embedded systems.

## IP configuration

Since version 3.10, the imperix IP is configurable:

- The SFP port can be repurposed by the user, to implement custom communication between devices using the Aurora protocol. An example of SFP port repurposing is available on the page Example of FPGA-based Aurora 8B/10B communication.
- Unused PWM modulators (CB-PWM, SS-PWM or PP-PWM) can be removed, to save up FPGA resources.



**IMPERIX_FW (3.10)**

☐ Show disabled ports

Component Name: IXIP

**SFP ports**

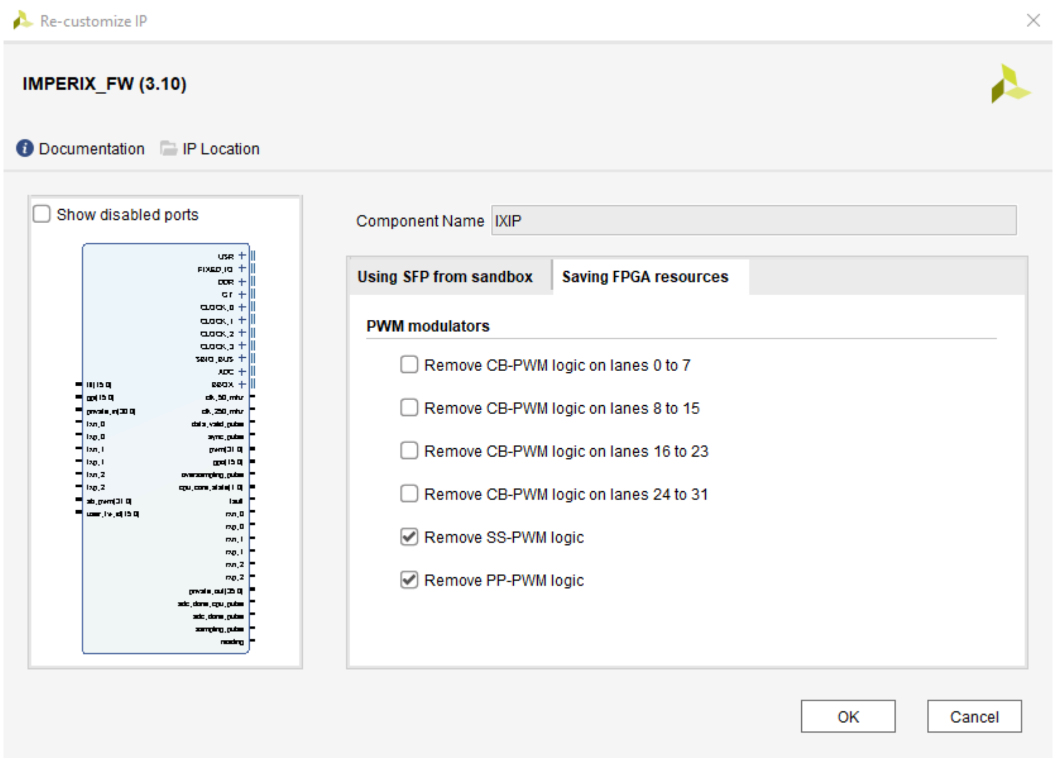
☐ Disable RealSync on SFP 0 (UP) and make it available from the sandbox

☐ Disable RealSync on SFP 1 (DOWN 0) and make it available from the sandbox

☐ Disable RealSync on SFP 2 (DOWN 1) and make it available from the sandbox

Click on the documentation button to learn more on how to use the SFP ports.

Settings to disable RealSync on specific ports


Re-customize IP

**IMPERIX_FW (3.10)**

Documentation   IP Location

[ ] Show disabled ports

Component Name: IXIP

Using SFP from sandbox | Saving FPGA resources

**PWM modulators**

- [ ] Remove CB-PWM logic on lanes 0 to 7
- [ ] Remove CB-PWM logic on lanes 8 to 15
- [ ] Remove CB-PWM logic on lanes 16 to 23
- [ ] Remove CB-PWM logic on lanes 24 to 31
- [✓] Remove SS-PWM logic
- [✓] Remove PP-PWM logic

OK   Cancel

Settings to remove unused PWM modulators to save resources in the FPGA sandbox

## Download

The source files are now available on the imperix IP download page.

Back to FPGA development homepage