Basic Embedded System Design Tutorial

using Zynq-7000 AP SOC embedded processor to design two frequencies PWM modulator system
Chapter 1

INTRODUCTION

1.1 Motivation

"Basic Embedded System Design Tutorial" is a document made for beginners who are entering the embedded system design world using FPGAs. This tutorial explains, step by step, the procedure of designing a simple digital system using C language, Xilinx Vivado Design Suite and Sozius development board.

1.2 Purpose of this Tutorial

Introduction

This tutorial is made to introduce you how to create and test an project and run it on your development board.

The following project is designed for:

- Designing Surface: VIVADO 2018.3
- Programming Language: C
- Device: Sozius Development Board

After completing this tutorial, you will be able to:

- Launch and navigate the Vivado Integrated Design Environment (IDE)
- Create a Zynq-7000 AP SoC processor system project using Vivado IP Integrator tool and Tcl programming interface
- Synthesize and implement the design in the Vivado IDE
- Export a hardware description XML file for later software development
- Create and debug your software application using SDK tool
- Generate the hardware implementation bitstream file and download it to the target development board
- Debug a design in hardware using Vivado Logic Analyzer
- Use custom IPs in your embedded system design
1.3 Objectives of this Tutorial

Objectives of this Tutorial

In this tutorial a **PWM** signal modulated using the sine wave with two **different frequencies** (1 Hz and 3.5 Hz) will be created.

Frequency that will be chosen depends on the position of the two-state on-board switch.

**PWM Signal**

Pulse-width modulation (PWM) uses a rectangular pulse wave whose pulse width is modulated by some other signal (in our case we will use a sine wave) resulting in the variation of the average value of the waveform. Typically, PWM signals are used to either convey information over a communications channel or control the amount of power sent to a load. To learn more about PWM signals, please visit [http://en.wikipedia.org/wiki/Pulse-width_modulation](http://en.wikipedia.org/wiki/Pulse-width_modulation).

![Figure 1.1: Example of the PWM signal](image)

Figure 1.1: Example of the PWM signal

Figure 1.1 illustrates the principle of pulse-width modulation. In this picture an arbitrary signal is used to modulate the PWM signal, but in our case sine wave signal will be used.

1.4 One Possible Solution for the Modulator Design

Considering that we are working with digital systems and signals, our task will be to generate a digital representation of an analog (sine) signal with two frequencies: 1 Hz and 3.5 Hz.

![Figure 1.2: Sine wave for PWM modulation](image)

Figure 1.2 is showing the sine wave that will be used to modulate the PWM signal.
CHAPTER 1. Introduction

Figure 1.2: Sine wave with 256 samples

One period of the sine wave is represented with 256 ($2^8$) samples, where each sample can take one of 4096 ($2^{12}$) possible values. Since the sine wave is a periodic signal, we only need to store samples of one period of the signal.

Note: Pay attention that all of sine signals with the same amplitude, regardless their frequency, look the same during the one period of a signal. The only thing that is different between those sine signals is duration of a signal period. This means that the sample rate of those signals is different.

Considering that the whole system will be clocked with the 50 MHz input signal, which is available on the target development board, to get 1 Hz and 3.5 Hz frequencies (which is much smaller than 50 MHz) we should divide input clock frequency with integer value $N$.

In the Tables 1.1 and 1.2 are shown parameters that are necessary for generating sine signals with 1 Hz and 3.5 Hz frequencies.

<table>
<thead>
<tr>
<th>Division Factor</th>
<th>Steps</th>
<th>Calculation</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T=1 \text{ s}$</td>
<td>$f_1=256$</td>
<td>$T=1/1 \text{ Hz}=1 \text{ s}$</td>
<td>$T$ is the period of the signal</td>
</tr>
<tr>
<td>$f_1=256$</td>
<td></td>
<td>$f_1=256 \times 1 \text{ Hz}=256 \text{ Hz}$ (or read in time: $1 \text{ s}/256$)</td>
<td>$f_1$ is the frequency of reading whole period ($T$) with 256 samples</td>
</tr>
<tr>
<td>$N_1=195312.5$</td>
<td></td>
<td>$N_1=50 \text{ MHz}/256 \text{ Hz}=195312.5$</td>
<td>$N_1$ is the number which divides frequency of the input clock signal (50 MHz) to the required frequency for the digital sine module</td>
</tr>
<tr>
<td>$N_2=48$</td>
<td></td>
<td>$N_2=195313/4096=47.6688$</td>
<td>$N_2$ is the number which divides frequency of the input clock signal (50 MHz) to the required frequency for the PWM’s FSM module</td>
</tr>
<tr>
<td>$N_1=196608$</td>
<td></td>
<td>$N_1=48 \times 4096=196608$</td>
<td>This is new calculation, because $N_1$ must be divisible with 4096</td>
</tr>
</tbody>
</table>

Table 1.1: Sine signal with the frequency of 1 Hz
Table 1.2: Sine signal with the frequency of 3.5 Hz

<table>
<thead>
<tr>
<th>Division Factor Steps</th>
<th>Calculation</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>T=0.286 s</td>
<td>T=1/3.5 Hz=0.286 s</td>
<td>T is the period of the signal</td>
</tr>
<tr>
<td>f2=896 Hz</td>
<td>f2=256x3.5 Hz=896 Hz (or read in time: 0.286 s/256)</td>
<td>f1 is the frequency of reading whole period (T) with 256 samples</td>
</tr>
<tr>
<td>N1=55803.5714</td>
<td>N1=50 MHz/896 Hz=55803.5714</td>
<td>N1 is the number which divides frequency of the input clock signal (50 MHz) to the required frequency for the digital sine module</td>
</tr>
<tr>
<td>N2=14</td>
<td>N2=50804/4096=13.624</td>
<td>N2 is the number which divides frequency of the input clock signal (50 MHz) to the required frequency for the PWM’s FSM module</td>
</tr>
<tr>
<td>N1=57344</td>
<td>N1=14x4096=57344</td>
<td>This is new calculation, because N1 must be divisible with 4096</td>
</tr>
</tbody>
</table>

Now, it is obvious that the sine wave can be generated by reading sample values of one period, that are stored in one table, with appropriate speed. In our case the values will be generated using the sine function from the C numerics library (math.h) and will be stored in an array.

1.4.1 Block diagram

Block diagram on the Figure 1.3 shows the structure of one possible system that can be used to generate required PWM signals.

**Block diagram**

![Block diagram](image)

Figure 1.3: Structure of microprocessor-based embedded system that will be used in tutorial

The embedded system is composed of:

- **Zynq-7000 AP SoC Processor core**
- **AXI Timer core**
• **AXI Interrupt Controller** core
• **AXI GPIO** core to drive the LED and to read the status of the SWITCH

Let us briefly explain each part of this system:

**Zynq-7000 AP Soc Processor** - The Zynq-7000 family is based on the Xilinx All Programmable SoC (AP SoC) architecture. The Zynq-7000 AP SoC is composed of two major functional blocks: **Processing System (PS)** and **Programmable Logic (PL)**, see Figure 1.4. The hart of the Processing System block is dual-core ARM Cortex-A9 MPCore CPU. Beside ARM processor, PS also includes Application Processor Unit (APU), Memory Interface, I/O Peripherals (IOP) and Interconnect.

![Figure 1.4: Zynq-7000 AP SoC block diagram](image)

**Processing System (PS):**

**Application Processor Unit (APU)** - provides an extensive offering of high-performance features and standards-compliant capabilities. APU contains:

- **Dual ARM Cortex-A9 MPCore CPUs with ARM v7:**
  - Run time options allows single processor, asymmetrical (AMP) or symmetrical multiprocessing (SMP) configurations
  - ARM version 7 ISA: standard ARM instruction set and Thumb-2, Jazelle RCT and Jazelle DBX Java acceleration
  - NEON 128 SIMD coprocessor and VFPv3 per MPCore
  - 32 KB instruction and data L1 caches with parity per MPCore
  - 512 KB of shareable L2 cache with parity
  - Private timers and watchdog timers

**Memory Controller** - the memory interfaces includes multiple memory technologies:

- DDR Controller
CHAPTER 1. INTRODUCTION

- DDR Controller Core and Transaction Scheduler
- Quad-SPI Controller
- Static Memory Controller (SMC)

I/O Peripherals - the I/O Peripherals (IOP) are a collection of industry-standard interfaces for external data communication:

- GPIO
- Gigabit Ethernet Controllers (two)
- USB Controller: Each as Host, Device or OTG (two)
- SD/SDIO Controllers (two)
- SPI Controllers (two): Master or Slave
- CAN Controllers (two)
- UART Controllers (two)
- I2C Controllers (two)
- PS MIO I/Os

UART Controller - is a full-duplex asynchronous receiver and transmitter that supports a wide range of programmable baud rates and I/O signal formats. The controller can accommodate automatic parity generation and multi-master detection mode.

The UART operations are controlled by the configuration and mode registers. The state of the FIFOs, modem signals and other controller functions are read using status, interrupt status and modem status registers.

The controller is structured with separate Rx and Tx data paths. Each path includes a 64-byte FIFO. The controller serializes and de-serializes data in the Tx and Rx FIFOs and includes a mode switch to support various loopback configurations for the RxD and TxD signals. Software reads and writes data bytes using Rx and Tx data port registers.

Each UART controller (UART 0 and UART 1) has the following features:

- Programmable baud rate generator
- 64-byte receive and transmit FIFOs
- Programmable protocol
- Parity, framing and overrun error detection
- Line-break generation
- Interrupts generation
- RxD and TxD modes: Normal/echo and diagnostic loopbacks using the mode switch
- Loop UART 0 and UART 1 option
- Modem control signals

The block diagram of the UART module is shown on the Figure 1.5.
CHAPTER 1. INTRODUCTION

Figure 1.5: UART block diagram

Note: The UART Controller will be used in the Sub-chapter 3.3 Creating a C/C++ source files for socius ARM-based processor system to transmit debug and system status information during application execution to the attached PC.

If you want to read and learn more about UART Controller, please refer to Chapter 19 “UART Controller” in the “Zynq-7000 All Programmable SoC – Technical Reference Manual”.

Programmable Logic (PL):

The PL provides a rich architecture of user-configurable capabilities:

- Configurable Logic Blocks (CLB)
- 36 Kb Block RAM
- Digital Signal Processing - DSP48E1 Slice
- Clock Management
- Configurable I/Os
- Low-Power Gigabit Transceivers
- Analog to Digital Converter (XADC)
- Integrated Interface Blocks for PCI Express designs

Note: If you want to read and learn more about the Zynq-7000 AP Soc processor core, please refer to “Zynq-7000 All Programmable SoC - Technical Reference Manual”.

LogiCORE IP AXI Timer/Counter - The LogiCORE IP AXI Timer/Counter is a 32/64-bit timer module that interfaces to the AXI4-Lite interface. The AXI Timer is organized as two identical timer modules. Each timer module has an associated load register that is used to hold either the initial value for the counter for event generation or a capture value, depending on the mode of the timer.

The AXI Timer includes the following features:

- AXI interface based on the AXI4-Lite specification
- Two programmable interval timers with interrupt, event generation, and event capture capabilities
- Configurable counter width
- One Pulse Width Modulation (PWM) output
CHAPTER 1. INTRODUCTION

- Cascaded operation of timers in generate and capture modes
- Freeze input for halting counters during software debug

Figure 1.6: AXI Timer core block diagram

Note: If you want to read and learn more about the AXI Timer/Counter core, please refer to "LogiCORE IP AXI Timer v2.0 Product Guide".

LogiCORE IP AXI Interrupt Controller (INTC) - The LogiCORE IP AXI Interrupt Controller (INTC) core receives multiple interrupt inputs from peripheral devices and merges them to a single interrupt output to the system processor. The registers used for storing interrupt vector addresses, checking, enabling and acknowledging interrupts are accessed through the AXI4-Lite interface.

The AXI Interrupt Controller includes the following features:

- Register access through the AXI4-Lite interface
- Fast Interrupt mode
- Supports up to 32 interrupts. Cascadable to provide additional interrupt inputs
- Single interrupt output
- Priority between interrupt requests is determined by vector position. The least significant bit (LSB, in the case bit 0) has the highest priority
- Interrupt Enable Register for selectively enabling individual interrupt inputs
- Master Enable Register for enabling interrupts request output
- Each input is configurable for edge or level sensitivity. Edge sensitivity can be configured for rising or falling. Level sensitivity can be active-high or active-low
- Output interrupt request pin is configurable for edge or level generation. Edge generation is configurable for rising or falling and level generation is configurable for active-high or active-low
- Configurable Software Interrupt capability
- Support for nested interrupts
CHAPTER 1. INTRODUCTION

Figure 1.7: AXI INTC core block diagram

The LogiCORE IP INTC core concentrates multiple interrupt inputs from peripheral devices to a single interrupt output to the system processor. The registers used for checking, enabling, and acknowledging interrupts are accessed through the AXI4-Lite interface.

*Note:* If you want to read and learn more about the AXI Interrupt Controller core, please refer to "LogiCORE IP AXI Interrupt Controller (INTC) v4.1 Product Guide".

LogiCORE IP AXI General Purpose Input/Output (GPIO) - The LogiCORE IP AXI General Purpose Input/Output (GPIO) core provides a general purpose input/output interface to the AXI interface. This 32-bit soft IP core is designed to interface with the AXI4-Lite interface.

The AXI GPIO includes the following features:

- Supports the AXI4-Lite interface specification
- Supports configurable single or dual GPIO channel(s)
- Supports configurable channel width for GPIO pins from 1 to 32 bits
- Supports dynamic programming of each GPIO bit as input or output
- Supports individual configuration of each channel
- Supports independent reset values for each bit of all registers
- Supports optional interrupt request generation

The AXI GPIO design provides a general purpose input/output interface to an AXI4-Lite interface. The AXI GPIO can be configured as either a single or a dual-channel device. The width of each channel is independently configurable.

The ports are configured dynamically for input or output by enabling or disabling the 3-state buffer. The channels can be configured to generate an interrupt when a transition on any of their inputs occurs.
1.4.2 Design steps

This tutorial will be realized step by step with the idea to explain the whole procedure of designing an digital system, using Vivado tool.

- First, we will create ("modulator") project for Zynq-7000 AP SoC processor system using Vivado IP Integrator tool. The block diagram of this system is shown on the Figure 1.3. Here we will configure the selected microprocessor and peripherals, and specify the interconnections between these components.

- After we create "modulator" project using Vivado IP Integrator tool, we will perform synthesis, implementation and bitstream file generation.

- Then, we will export our hardware platform description to the Software Development Kit (SDK). The exported file has all the necessary information that SDK requires for software development and debug work on the hardware platform that we designed.

- In the SDK, we will create and debug the software application for this project. There will be two different software applications, one without and one with the interrupt controller. Source codes for these two applications will be stored in modulator_socius_no_intc.c and modulator_socius_intc.c source file respectively.

- Now, the design is ready to be implemented. The last step will be to initialize the bitstream with the appropriate ELF file and download it to the target Xilinx development board.
1.5 Embedded Design Process Flow

Vivado Design Suite is designed to help us in all phases of the embedded design process. On the Figure 1.10 is shown the Vivado architecture structure of how the tools operate together to create an embedded system.
Figure 1.10: Typical embedded design process flow
Chapter 2

CREATING THE HARDWARE PLATFORM

In the previous chapter, we have defined the structure of the microprocessor based system that will be used as a part of the solution of PWM signal generation. In this chapter, we will explain how to generate this system using Vivado IP Integrator tool. While entire designs can be created using the IP Integrator, the typical design will consist of HDL, IP and IP integrator block designs.

2.1 Create a New Project

The first step in creating a new design will be to create a new project. We will create a new project using the Vivado IDE New Project wizard. The New Project wizard will create an XPR project file for us. It will be place where Vivado IDE will organize our design files and save the design status whenever the processes are run.

To create a new project, follow these steps:

- Launch the Vivado software:

Select Start -> All Programs -> Xilinx Design Tools -> Vivado 2018.3 -> Vivado 2018.3 and the Vivado Getting Started page will appear, see Figure 2.1.

As you can see from the figure below, the Getting Started page contains a lot of usable links (shortcuts) like Create Project, Open an existing Project, Open Example Project, Open Hardware Manager, Documentation and Tutorials and so on.
Create New Project

- On the **Getting Started** page, choose first offered **Create Project** option, under the **Quick Start** section.

- In the **Create a New Vivado Project** dialog box click **Next** and the wizard will guide you through the creation of a new project.
- In the **Project Name** dialog box specify the name and the location of the new project and click **Next**.

- In the **Project name** field type **modulator** as the name of our project
- In the **Project location** field specify the location where our project data will be stored
- Leave **Create project subdirectory** option enabled, see Figure 2.3

- In the **Project Type** dialog box specify the type of project you want to create. In our case we will choose **RTL Project** option. Select **Do not specify sources at this time** also and click **Next**.
As you can see from the figure above, five different types of the project can be created:

- **RTL Project** - The RTL Project environment enables you to add RTL source files and constraints, configure IP with the Vivado IP catalog, create IP subsystems with the Vivado IP integrator, synthesize and implement the design, and perform design planning and analysis.

- **Post-synthesis Project** - This type of project enables you to import third-party netlists, implement the design, and perform design planning and analysis.

- **I/O Planning Project** - With this type of project you can create an empty project for use with early I/O planning and device exploration prior to having RTL sources.

- **Imported Project** - This type of project enables you to import existing project sources from the ISE Design Suite, Xilinx Synthesis Technology (XST), or Synopsys Synplify.

- **Configure an Example Embedded Evaluation Board Design** - This type of project enables you to target the example Zynq-7000 or MicroBlaze embedded designs to the available Xilinx evaluation boards.

- In the Default Part dialog box choose a default Xilinx part or board for your project and click Next.
CHAPTER 2. CREATING THE HARDWARE PLATFORM

The main component of the Sozius development board is Zynq-7000 AP SoC, so in the Default Part dialog box select Parts option and set the filter parameters as it is shown on the Figure 2.5.

- In the New Project Summary dialog box click Finish if you are satisfied with the summary of your project.

If you are not satisfied, you can go back as much as necessary to correct all the questionable issues, see Figure 2.6.

After we finished with the new project creation, in a few seconds Vivado IDE Viewing Environment will...
When Vivado creates new project, it also creates a directory with the name and at the location that we specified in the GUI. That means that the all project data will be stored in the project name (modulator) directory containing the following:

- **project_name.xpr** file - object that is selected to open a project (Vivado IDE project file)
- **project_name.runs** directory - contains all run data
- **project_name.srcs** directory - contains all imported local HDL source files, netlists, and XDC files
- **project_name.data** directory - stores floorplan and netlist data
- **project_name.sim** directory - contains all simulation data

### Vivado IDE Viewing Environment

![Vivado IDE Viewing Environment](image)

Figure 2.7: Vivado IDE Viewing Environment

#### 2.2 Vivado Integrated Design Environment

The Vivado IDE can be used for a variety of purposes at various stages in the design flow and is very helpful at detecting design problems early in the design flow.

The Vivado IDE allows different file types to be added as design sources, including Verilog, VHDL, EDIF, NGC format cores, SDC, XDC, and TCL constraints files, and simulation test benches. These files can be stored in variety of ways using the tabs at the bottom of the Sources window: **Hierarchy, Library** or **Compile Order**, see Figure 2.8.
By default, after launching, the Vivado IDE opens the Default Layout. Each docked window in the Vivado IDE is called a view, so you can find Sources View, Properties View, Project Summary View and so on, see Figure 2.8.

![Vivado IDE Viewing Environment](image)

**Flow Navigator**

The vertical toolbar present on the left side of the Vivado IDE is the **Flow Navigator**. The Flow Navigator provides control over the major design process tasks, such as project configuration, synthesis, implementation and bitstream creation.

**Sources View**

The **Sources** view displays the list of source files that has been added in the project.

- The **Design Sources** folder helps you keep track of VHDL and Verilog design source files and libraries.
- The **Constraints** folder helps you keep track of the constraints files.
- The **Simulation Sources** folder helps keep track of VHDL and Verilog simulation sources source files and libraries.

Notice that the design hierarchy is displayed as default.

- In the **Libraries** tab, sources are grouped by file type, while the **Compile Order** tab shows the file order used for synthesis.
Project Summary View

The Project Summary view provides a brief overview of the status of different processes executed in the Vivado IDE, see Figure 2.9.

![Project Summary View](image)

**Figure 2.9: Project Summary View**

The Project Settings panel displays the project name, product family, project part, and top module name. Clicking a link in this panel you will open the Project Settings dialog box.

- The Messages panel summarizes the number of errors and warnings encountered during the design process.
- The Synthesis panel summarizes the state of synthesis in the active run. The synthesis panel also shows the target part and the strategy applied in the run.
- The Implementation panel summarizes the state of implementation in the active run. The Implementation panel also shows the target part and the strategy applied in the run.

Tcl Console

Below the Project Summary view, see Figure 2.10, is the Tcl Console which echoes the Tcl commands as operations are performed. It also provides a means to control the design flow using Tcl commands.

2.3 Create ARM-based Hardware Platform

**IP Integrator (IPI) Tool**

To accelerate the creation of highly integrated and complex designs, Vivado Design Suite is delivered with IP Integrator (IPI) which provides a new graphical and Tcl-based IP- and system-centric design development flow.

The Xilinx Vivado Design Suite IP Integrator feature lets you create complex system designs by instantiating and interconnecting IP cores from the Vivado IP Catalog onto a design canvas.
You can create designs interactively through the IP Integrator design canvas GUI, or using a Tcl programming interface.

Rapid development of smarter systems requires levels of automation that go beyond RTL-level design. The Vivado IP Integrator accelerates IP- and system-centric design implementation by providing the following:

- Seamless inclusion of IPI sub-systems into the overall design
- Rapid capture and packing of IPI designs for reuse
- Tcl scripting and graphical design
- Rapid simulation and cross-probing between multiple design views
- Support for processor or processor-less designs
- Integration of algorithmic and RTL-level IP
- Combination of DSP, video, analog, embedded, connectivity and logic
- Matches typical designer flows
- Easy to reuse complex sub-systems
- DRCs on complex interface level connections during design assembly
- Recognition and correction of common design errors
- Automatic IP parameter propagation to interconnected IP
- System-level optimizations

You will typically construct design at the AXI interface level for greater productivity, but you may also manipulate designs at the port level for more precise design control.

Create IP Integrator Design

In this tutorial you will instantiate a few IPs in the IP Integrator tool and then stitch them together to create an IP based system design.

While working on this tutorial, you will be introduced to the IP Integrator GUI, run design rule checks (DRC) on your design, and then integrate the design in a top-level design in the Vivado Design Suite.

Finally, you will run synthesis and implementation process, generate bitstream file and run your design on the Sozius development board.

2.3.1 Create ARM-based Hardware Platform for Sozius Development Board

About Sozius development board

Sozius development platform is a small, portable electronic device that can easily be powered from a USB port, USB charger, Power Over Ethernet or a battery pack.

You can easily develop software and/or digital hardware for it, because it uses an FPGA with an embedded processors.

Sozius delivers already a well designed board and should help you to focus on the specifics of your project and can be easily extended to meet your needs.

The main system with many interfaces and Linux is already preconfigured and ready for use.
Zynq-7000 AP SoC Processor System

The main component of the Sozius development board is **Zynq-7000 AP SoC**. As we already said, the Zynq-7000 AP SoC is composed of two major functional blocks: **Processing System (PS)** and **Programmable Logic (PL)**. Since existing LEDs and switches on the Sozius board are connected to the PS part of the Zynq FPGA, it would require programming PS part of the Zynq FPGA. In this design we will not use PL part of the Zynq FPGA to implement timer and GPIO modules (as it is presented on the Figure 1.3), because we would not be able to connect them to the Sozius board LEDs and switches. Instead, we must use timer and GPIO modules from the PS part of the Zynq FPGA. More specifically, we will use one Triple Timer Counter (TTC) module, TTC0, that is present in the PS part of the Zynq FPGA and four General Purpose IO ports from the GPIO module that is also present in the PS part of the Zynq FPGA, see Figure 2.10.

This sub-chapter will show how to build Zynq-7000 All Programmable (AP) SoC processor "modulator" design using Vivado IDE and Tcl programming interface. In this sub-chapter, you will instantiate a few IPs in the IP Integrator tool and then stitch them together to create an IP based system design. At the end, you will run synthesis and implementation process and generate bitstream file.

The following steps describe how to create ARM-based hardware platform for Sozius development board.

**Create modulator_sozius_arm_rtl.vhd and modulator_components_package.vhd Source Files**

- First, we will create **modulator_sozius_arm_rtl.vhd** and **sozius_components_package.vhd** files using Vivado test editor and save them into the working directory.

  - **modulator_sozius_arm_rtl.vhd** file will hold the top-level module of our design, in which Zynq PS component configured for Sozius development board will be instantiated
  - **sozius_components_package.vhd** file will contain Sozius PS module component declaration.

The content of the both files is presented in the text below.
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
library unisim;
use unisim.vcomponents.all;
library work;
use work.sozius_components_package.all;

entity modulator_sozius_arm is
port(
    -- ps io
    ps_ddr3_addr : inout std_logic_vector(14 downto 0);
    ps_ddr3_ras : inout std_logic;
    ps_ddr3_ba : inout std_logic;
    ps_ddr3_cas_n : inout std_logic;
    ps_ddr3_dq : inout std_logic_vector(31 downto 0);
    ps_ddr3_dqs : inout std_logic;
    ps_ddr3_ck_n : inout std_logic;
    ps_ddr3_ck_p : inout std_logic;
    ps_ddr3_cke : inout std_logic;
    ps_ddr3_cs_n : inout std_logic;
    ps_ddr3_dm : inout std_logic_vector(3 downto 0);
    ps_ddr_vrn : inout std_logic;
    ps_ddr_vrp : inout std_logic;
    ps_clk_i : inout std_logic;
    ps_por_n_i : inout std_logic;
    ps_srst_n_i : inout std_logic;
    ps_phy_mdc_i : inout std_logic;
    ps_phy_mdio_i : inout std_logic;
    ps_phy_rx_clk : inout std_logic;
    ps_phy_rx_ctrl : inout std_logic;
    ps_phy_rxd : inout std_logic_vector(3 downto 0);
    ps_phy_tx_clk : inout std_logic;
    ps_phy_tx_ctrl : inout std_logic;
    ps_phy_txd : inout std_logic_vector(3 downto 0);
    ps_i2c_scl : inout std_logic;
    ps_i2c_sda : inout std_logic;
    ps_led_error : inout std_logic;
    ps_led_front_n : inout std_logic_vector(1 downto 0);
    ps_sw0_a : inout std_logic;
    ps_sw0_b : inout std_logic;
    ps_sw1_a : inout std_logic;
    ps_sw1_b : inout std_logic;
    ps_sw2_a : inout std_logic;
    ps_sw2_b : inout std_logic;
    ps_sw3_a : inout std_logic;
    ps_sw3_b : inout std_logic;
    ps_uart_rx : inout std_logic;
    ps_uart_tx : inout std_logic;
    ps_qspi_cs_n : inout std_logic;
    ps_qspi_data : inout std_logic_vector(3 downto 0);
    ps_qspi_clk : inout std_logic;
    ps_sdio_clk : inout std_logic;
    ps_sdio_cmd : inout std_logic;
    ps_sdio_data : inout std_logic_vector(3 downto 0);
    ps_usb_clk : inout std_logic;
    ps_usb_data : inout std_logic_vector(7 downto 0);
    ps_usb_dir : inout std_logic;
    ps_usb_nxt : inout std_logic;
    ps_usb_stp : inout std_logic;
);
end entity;

architecture structural of modulator_sozius_arm is
-- between architecture and begin is declaration area for types, signals and constants
-- everything declared here will be visible in the whole architecture
-- declaration for fixed signal PL to PS
signal pl_clk0_s : std_logic;
signal pl_reset_n_s : std_logic;
begin

end architecture;
--- instance of processor system PS

soriz_mr_lab_ps_hd_i: component soriz_mr_lab_ps_hd
  port map (
    ddr3_addr => ps_ddr3_addr,
    ddr3_fa => ps_ddr3_fa,
    ddr3_ca => ps_ddr3_ca,
    ddr3_cke => ps_ddr3_cke,
    ddr3_cs => ps_ddr3_cs,
    ddr3_s => ps_ddr3_s,
    ddr3_r => ps_ddr3_r,
    ddr3_i => ps_ddr3_i,
    ddr3_cas_n => ps_ddr3_cas_n,
    ddr3_cas => ps_ddr3_cas,
    ddr3_cas_p => ps_ddr3_cas_p,
    ddr3_cas_n => ps_ddr3_cas_n,
    ddr3_cas_p => ps_ddr3_cas_p,
    ddr3_dq => ps_ddr3_dq,
    ddr3_dq => ps_ddr3_dq,
    ddr3_dq_p => ps_ddr3_dq_p,
    ddr3_dq_n => ps_ddr3_dq_n,
    ddr3_dq_n => ps_ddr3_dq_n,
    ddr3_dq_p => ps_ddr3_dq_p,
    ddr3_dqs_n => ps_ddr3_dqs_n,
    ddr3_dqs_p => ps_ddr3_dqs_p,
    ddr3_odt => ps_ddr3_odt,
    ddr3_ras_n => ps_ddr3_ras_n,
    ddr3_reset_n => ps_ddr3_reset_n,
    ddr3_we_n => ps_ddr3_we_n,
    fixed_io_ddr_y_map => ps_ddr_y_map,
    fixed_io_ddr_y_map => ps_ddr_y_map,
    fixed_io_ddr_y_map => ps_ddr_y_map,
    fixed_io_ddr_y_map => ps_ddr_y_map,
    fixed_io_dio => ps_dio,
    fixed_io_dio => ps_dio,
    fixed_io_dio => ps_dio,
    fixed_io_dio => ps_dio,
    fixed_io_clk => ps_clk,
    fixed_io_clk => ps_clk,
    fixed_io_clk => ps_clk,
    fixed_io_clk => ps_clk,
    fixed_io_port => ps_port,
    fixed_io_port => ps_port,
    fixed_io_port => ps_port,
    fixed_io_port => ps_port,
    pl_uart_1_rxd => '0',
    pl_uart_1_rxd => '0',
    pl_uart_1_rxd => '0',
    pl_uart_1_rxd => '0',
    pl_spi_0_io0_i => '0',
    pl_spi_0_io0_i => '0',
    pl_spi_0_io0_i => '0',
    pl_spi_0_io0_i => '0',
    pl_iic_1_scl_i => '0',
    pl_iic_1_scl_i => '0',
    pl_iic_1_scl_i => '0',
    pl_iic_1_scl_i => '0',
    sdio_0_cdn => '1',
    usbind_0_port_indctl => open,
    usbind_0_port_indctl => open,
    usbind_0_vbus_pwrselect => '1',
    usbind_0_vbus_pwrselect => '1',
    pl_reset_n => pl_reset_n,
    pl_reset_n => pl_reset_n,
  );

  -- assignment of MIO to board names:

  ps_mio_s (53) <= ps_phy_mdio_io;
  ps_mio_s (52) <= ps_phy_mdc_io;
  ps_mio_s (51) <= ps_uart_tx_io;
  ps_mio_s (50) <= ps_uart_rx_io;
  ps_mio_s (49) <= ps_led_error_n_io;
  ps_mio_s (48 downto 47) <= ps_led_front_n_io(1 downto 0);
  ps_mio_s (46) <= ps_sdio_s_ido;
  ps_mio_s (45) <= ps_sdio_s_ido;
  ps_mio_s (44) <= ps_sdio_s_ido;
  ps_mio_s (43) <= ps_sdio_s_ido;
  ps_mio_s (42) <= ps_sdio_s_ido;
  ps_mio_s (41) <= ps_sdio_s_ido;
  ps_mio_s (40) <= ps_sdio_s_ido;
  ps_mio_s (39) <= ps_sdio_s_ido;
  ps_mio_s (38) <= ps_sdio_s_ido;
  ps_mio_s (37) <= ps_sdio_s_ido;
  ps_mio_s (36) <= ps_sdio_s_ido;
  ps_mio_s (35) <= ps_sdio_s_ido;
  ps_mio_s (34) <= ps_sdio_s_ido;
  ps_mio_s (33) <= ps_sdio_s_ido;
  ps_mio_s (32) <= ps_sdio_s_ido;
  ps_mio_s (31) <= ps_sdio_s_ido;
  ps_mio_s (30) <= ps_sdio_s_ido;
  ps_mio_s (29) <= ps_sdio_s_ido;
  ps_mio_s (28) <= ps_sdio_s_ido;
  ps_mio_s (27) <= ps_sdio_s_ido;
  ps_mio_s (26) <= ps_sdio_s_ido;
  ps_mio_s (25) <= ps_sdio_s_ido;
  ps_mio_s (24) <= ps_sdio_s_ido;
  ps_mio_s (23) <= ps_sdio_s_ido;
  ps_mio_s (22) <= ps_sdio_s_ido;
  ps_mio_s (21) <= ps_sdio_s_ido;
  ps_mio_s (20) <= ps_sdio_s_ido;
  ps_mio_s (19) <= ps_sdio_s_ido;
  ps_mio_s (18) <= ps_sdio_s_ido;
  ps_mio_s (17) <= ps_sdio_s_ido;
  ps_mio_s (16) <= ps_sdio_s_ido;
  ps_mio_s (15) <= ps_sdio_s_ido;
  ps_mio_s (14) <= ps_sdio_s_ido;
  ps_mio_s (13) <= ps_sdio_s_ido;
  ps_mio_s (12) <= ps_sdio_s_ido;
  ps_mio_s (11) <= ps_sdio_s_ido;
  ps_mio_s (10) <= ps_sdio_s_ido;
  ps_mio_s (9) <= ps_sdio_s_ido;
  ps_mio_s (8) <= ps_sdio_s_ido;
  ps_mio_s (7) <= ps_sdio_s_ido;
  ps_mio_s (6) <= ps_sdio_s_ido;
  ps_mio_s (5) <= ps_sdio_s_ido;
  ps_mio_s (4) <= ps_sdio_s_ido;
  ps_mio_s (3) <= ps_sdio_s_ido;
  ps_mio_s (2) <= ps_sdio_s_ido;
  ps_mio_s (1) <= ps_sdio_s_ido;
  ps_mio_s (0) <= ps_sdio_s_ido;
CHAPTER 2. CREATING THE HARDWARE PLATFORM

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

package sozius_components_package is
  component sozius_xz_lab_ps_bd is
    port (        
      pl_clk0 : out std_logic;
      pl_reset_n : out std_logic;
      ddr3_cas_n : inout std_logic;
      ddr3_cke : inout std_logic;
      ddr3_ck_n : inout std_logic;
      ddr3_ck_p : inout std_logic;
      ddr3_cs_n : inout std_logic;
      ddr3_reset_n : inout std_logic;
      ddr3_odt : inout std_logic;
      ddr3_ras_n : inout std_logic;
      ddr3_we_n : inout std_logic;
      ddr3_ba : inout std_logic_vector ( 2 downto 0 );
      ddr3_addr : inout std_logic_vector ( 14 downto 0 );
      ddr3_dm : inout std_logic_vector ( 3 downto 0 );
      ddr3_dq : inout std_logic_vector ( 31 downto 0 );
      ddr3_dqs_n : inout std_logic_vector ( 3 downto 0 );
      ddr3_dqs_p : inout std_logic_vector ( 3 downto 0 );
      fixed_io_mio : inout std_logic_vector ( 53 downto 0 );
      fixed_io_ddr_vrn : inout std_logic;
      fixed_io_ddr_vrp : inout std_logic;
      fixed_io_ps_srstb : inout std_logic;
      fixed_io_ps_clk : inout std_logic;
      fixed_io_ps_porb : inout std_logic;
      sdio_0_cdn : in std_logic;
      usbind_0_port_indctl : out std_logic_vector ( 1 downto 0 );
      usbind_0_vbus_pwrselect : out std_logic;
      usbind_0_vbus_pwrfault : in std_logic;
      pl_iic_1_sda_i : in std_logic;
      pl_iic_1_sda_o : out std_logic;
      pl_iic_1_sda_t : out std_logic;
      pl_iic_1_scl_i : in std_logic;
      pl_iic_1_scl_o : out std_logic;
      pl_iic_1_scl_t : out std_logic;
      pl_spi_0_sck_i : in std_logic;
      pl_spi_0_sck_o : out std_logic;
      pl_spi_0_sck_t : out std_logic;
      pl_spi_0_io0_i : in std_logic;
      pl_spi_0_io0_o : out std_logic;
      pl_spi_0_io0_t : out std_logic;
      pl_spi_0_io1_i : in std_logic;
      pl_spi_0_io1_o : out std_logic;
      pl_spi_0_io1_t : out std_logic;
      pl_spi_0_ss_i : in std_logic;
      pl_spi_0_ss_o : out std_logic;
      pl_spi_0_ss1_o : out std_logic;
      pl_spi_0_ss2_o : out std_logic;
      pl_spi_0_ss_t : out std_logic;
      pl_uart_1_txd : out std_logic;
      pl_uart_1_rxd : in std_logic;
    );
  end component;
end package;

sozius_components_package.vhd:

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

package sozius_components_package is
  component sozius_xz_lab_ps_bd is
    port (        
      pl_clk0 : out std_logic;
      pl_reset_n : out std_logic;
      ddr3_cas_n : inout std_logic;
      ddr3_cke : inout std_logic;
      ddr3_ck_n : inout std_logic;
      ddr3_ck_p : inout std_logic;
      ddr3_cs_n : inout std_logic;
      ddr3_reset_n : inout std_logic;
      ddr3_odt : inout std_logic;
      ddr3_ras_n : inout std_logic;
      ddr3_we_n : inout std_logic;
      ddr3_ba : inout std_logic_vector ( 2 downto 0 );
      ddr3_addr : inout std_logic_vector ( 14 downto 0 );
      ddr3_dm : inout std_logic_vector ( 3 downto 0 );
      ddr3_dq : inout std_logic_vector ( 31 downto 0 );
      ddr3_dqs_n : inout std_logic_vector ( 3 downto 0 );
      ddr3_dqs_p : inout std_logic_vector ( 3 downto 0 );
      fixed_io_mio : inout std_logic_vector ( 53 downto 0 );
      fixed_io_ddr_vrn : inout std_logic;
      fixed_io_ddr_vrp : inout std_logic;
      fixed_io_ps_srstb : inout std_logic;
      fixed_io_ps_clk : inout std_logic;
      fixed_io_ps_porb : inout std_logic;
      sdio_0_cdn : in std_logic;
      usbind_0_port_indctl : out std_logic_vector ( 1 downto 0 );
      usbind_0_vbus_pwrselect : out std_logic;
      usbind_0_vbus_pwrfault : in std_logic;
      pl_iic_1_sda_i : in std_logic;
      pl_iic_1_sda_o : out std_logic;
      pl_iic_1_sda_t : out std_logic;
      pl_iic_1_scl_i : in std_logic;
      pl_iic_1_scl_o : out std_logic;
      pl_iic_1_scl_t : out std_logic;
      pl_spi_0_sck_i : in std_logic;
      pl_spi_0_sck_o : out std_logic;
      pl_spi_0_sck_t : out std_logic;
      pl_spi_0_io0_i : in std_logic;
      pl_spi_0_io0_o : out std_logic;
      pl_spi_0_io0_t : out std_logic;
      pl_spi_0_io1_i : in std_logic;
      pl_spi_0_io1_o : out std_logic;
      pl_spi_0_io1_t : out std_logic;
      pl_spi_0_ss_i : in std_logic;
      pl_spi_0_ss_o : out std_logic;
      pl_spi_0_ss1_o : out std_logic;
      pl_spi_0_ss2_o : out std_logic;
      pl_spi_0_ss_t : out std_logic;
      pl_uart_1_txd : out std_logic;
      pl_uart_1_rxd : in std_logic;
    );
  end component;
end package;
```

Add `modulator_sozius_arm_rtl.vhd` and `modulator_components_package.vhd` Source Files

- When we finished with the `modulator_sozius_arm_rtl.vhd` and `sozius_components_package.vhd` files creation, in the Vivado Flow Navigator, click Add Sources command.
- In the **Add or Create Design Sources** dialog box, click the + icon and select **Add Files...** option to include the existing source files into the project, or just click **Add Files** button.

- In the **Add Source Files** dialog box, browse to the project working directory and select the **modulator_sozius_arm_rtl.vhd** and **sozius_components_package.vhd** source files.
Figure 2.13: Add Source Files dialog box

- Click OK and the modulator_sozius_arm_rtl.vhd and sozius_components_package.vhd source files should appear in the Add or Create Design Sources dialog box.

Figure 2.14: Add or Create Design Sources dialog box - with added file

- Click Finish and your source files should appear under the Design Sources in the Sources view in the Project Manager window.
Create `sozius_xz_modulator_vio.xdc` Constraints File

- Now is the time to create constraints file for the Sozius board, `sozius_xz_modulator_vio.xdc`.

Open Vivado text editor, copy your constraints code in it or write directly in it and save the constraints file in your working directory.

The complete `sozius_xz_modulator_vio.xdc` source file you can find in the further text.

Note: If you want to read and learn more about XDC constraints files, please refer to the "Basic FPGA Tutorial", sub-chapter 9.1 "Creating XDC File" where you will find all the necessary information about the types of constraints, how to create them depending on the target board type and use them in your design.

```
# set properties commentstyle for bitstream generation
set_property BITSTREAM.GENERAL.COMPRESS TRUE [current_design]
# set_property BITSTREAM.GENERAL.XADCENHANCEDLINEARITY commentstyle ON [current_design]
# set_property BITSTREAM.GENERAL.XADCPOWERDOWN ENABLE [current_design]

# set commentstyle configuration bank voltages
set_property CFGBVS VCCO [current_design]
set_property CONFIG_VOLTAGE 3.3 [current_design]

# set condition commentstyle for power analyzer
set_operating_conditions -ambient_temp 50
set_operating_conditions -board small
set_operating_conditions -airflow 250
set_operating_conditions -heatsink low
set_operating_conditions -board_layers 12to15

# pins must be implemented!!
set_property PACKAGE_PIN V13 [get_ports pl_phy_reset_n_o]
set_property IOSTANDARD LVCMOS33 [get_ports pl_phy_reset_n_o]
```

Add `sozius_xz_modulator_vio.xdc` Constraints File

- In the Vivado Flow Navigator, click the Add Sources command.

- In the Add Sources dialog box select Add or create constraints option to add the `sozius_xz_modulator_vio.xdc` constraints file into our project and click Next.
- In the Add or Create Constraints dialog box, click + icon and select Add Files... option.

- In the Add Constraint Files dialog box, browse to the project working directory and select the sozius_xz_modulator_vio.xdc constraints file.

- Click OK and the sozius_xz_modulator_vio.xdc constraints file should appear in the Add or Create Constraints dialog box.

- Click Finish and your constraints file should appear under the Constraints in the Sources view.

Figure 2.16: Add or Create Constraints option

- Configure the Zynq PS Part to work on Sozius Development Board

Finally, we must configure the Zynq PS part to work on Sozius development board.

This includes a number of configuration steps, one of them being the proper configuration of the PS GPIO module to connect to the LEDs and switches that are present on the Sozius board.

Also, we must enable the Triple Timer Counter 0 (TTC0) module within Zynq PS, that will be used in the "modulator" design. All these PS configuration steps can be done using the Vivado GUI, by creating a block design.

However, since this task includes a lot of manual settings of the Zynq PS, a better approach would be to do this manual configuration only once and then to create a Tcl script file that can be used in all future configurations of the Zynq PS part.

The Tcl script that should be used to correctly configure Zynq PS to work on Sozius board is sozius_xz_lab_ps_bd.tcl.
CHAPTER 2. CREATING THE HARDWARE PLATFORM

This Tcl script file is too long to be shown in the tutorial, so ask your instructor for details.

**Execute Tcl File in the Vivado IDE**

- Next step is to execute the presented Tcl file in the Vivado IDE. Go to the **Tcl console** window and type the following and press enter:

  ```
  source <path>/sozius_xz_lab_ps_bd.tcl
  ```

  *<path>* stands for the full path to the folder where the *sozius_xz_lab_ps_bd.tcl* Tcl file is stored.

![](Figure2.18_TclConsoleWindow.png)

**Figure 2.18: Tcl Console window**

After Vivado has finished with the Tcl script execution, a created block diagram containing Zynq PS will be visible in the Vivado IDE, as shown on the Figure 2.20.

**Block Diagram after Vivado Tcl Script Execution**

![](Figure2.19_BlockDiagram.png)

**Figure 2.19: Block diagram of Zynq PS configured to run on Sozius board**

**Generate Bitstream File**

To prepare our design to work on the FPGA device, we must convert it to a bitstream file.

This process is composed of two important steps:

- one is to generate a netlist file that will represent your hardware platform and
- second one is to generate a bitstream file that will represent the hardware and software platforms together.
This bitstream file will be downloaded to the FPGA device.

There are two possibilities for netlist and bitstream file generation. One is to generate these files after a hardware platform is specified and the second one is to generate them after a software application development is completed.

If you would like to generate netlist and bitstream file after hardware platform specification:

**Synthesize and Implement "modulator" Design and Generate Bitstream File**

- In the Vivado Flow Navigator, click **Run Synthesis** command, and wait for task to be completed.

- When the synthesis process is completed, click **Run Implementation** command, and wait for task to be completed.

- At the end, when the implementation process is completed, click **Generate Bitstream** command. After this step, bitstream file will be generated.

![Run Synthesis, Run Implementation and Generate Bitstream commands from the Vivado Flow Navigator](image)

If you would like to generate netlist and bitstream files after a software application development is completed, please follow the next chapter where will be explained in detail the necessary steps.
Chapter 3

CREATING THE SOFTWARE PLATFORM USING SDK

Software Platform

In the previous chapter we have designed the hardware component of our embedded system.

To complete the design process we must now create a software component for our embedded system.

This application specific software will be executed on the ARM processor that is a part of our hardware platform.

When using Xilinx development tools, software design process is done using the Software Development Kit (SDK) tool. Our software application will be developed for the hardware platform built in IP Integrator tool.

Figure 3.1 shows, in detail, SDK application development flow.

SDK Application Development Flow

![SDK Application Development Flow Diagram](image)

Figure 3.1: SDK application development flow

The first step in software creation is to export the hardware design into the SDK.

The hardware design will be exported to an XML files that will be used by the SDK to create a software application for our hardware.
To create a software platform for your embedded design, use the following steps:

**Export Hardware**

- Select **File -> Export -> Export Hardware...** option from the main Vivado IDE menu.

- In the **Export Hardware** dialog box, make sure that **Include bitstream** check box is checked, and **Export to** field is set to **Local to Project** and click **OK**.

![Export Hardware dialog box](image)

*Figure 3.2: Export Hardware dialog box*

**Launch SDK**

- To launch SDK after the hardware platform has been exported, select **File -> Launch SDK** from the main Vivado IDE menu.

- In the **Launch SDK** dialog box, make sure that both **Exported location** and **Workspace** are set to **Local to Project** and click **OK**.

![Launch SDK dialog box](image)

*Figure 3.3: Launch SDK dialog box*

**SDK Main Window**
As you can see from the illustration above, SDK has automatically created a new hardware platform specification, `modulator_sozius_arm_hw_platform_0` with the `system.hdf` file inside. The system.hdf file is the hardware platform specification file which contains information about the target device, address map for the selected processor, about the IP blocks present in the design along with the links to the datasheets of all system peripherals.

- The **Source** tab shows the content of the system.hdf file
- The **Overview** tab shows most of the information necessary for writing software

### 3.1 Board Support Package

**Board Support Package**

Each software project must have a corresponding **Board Support Package (BSP)**.

A Board Support Package is a collection of libraries and drivers that form the lowest level of your software application stack.

Before you can create and use software applications in SDK, you must create a board support package. You can create more than one application to run on the BSP.

You can use SDK to create BSP for two different run-time environments:

- **Standalone** - A simple, low-level software layer. It provides access to basic processor features such as cache, interrupts and exceptions as well as the basic features of a hosted environment, such as standard input and output, profiling, abort and exit.
- **FreeRTOS** - A popular real-time operating system kernel for embedded devices, that has been ported to 35 microcontrollers, including ARM and Xilinx MicroBlaze.
In SDK we can have a multiple Board Support Packages. For example, one for a design that runs on the standalone environment, and one that uses Xilinx.

Generating BSP is very easy. The program that is launched is called LibGen (Library Generator). It reads the system.mss file for directions on which drivers and libraries to include. The MSS file is built when a software platform or BSP is created, and contains a software instance for every processor and peripheral IP.

The purpose of running LibGen is to compile the BSP ones. Ones compiled by LibGen, the BSP resides as an object ready to be linked to the compiled software application.

Create Board Support Package

To create a Board Support Package do the following:

- Select File -> New -> Board Support Package option.

- In the New Board Support Package Project dialog box leave all default options as they are set and click Finish.

- In the Board Support Package Settings dialog box leave all default options as they are set in all four tabs (Overview, standalone, drivers and microblaze_0) and click OK.
3.2 Creating an Application Project

Now is the moment when you will create an application project. The actual application will be written in C/C++ programming language. The SDK will create and maintain make file for you.

To create an application project, do the following:

Create Application Project

- Select **File -> New -> Application Project** and the **Application Project** dialog box will appear.
- In the **Project name** field, type a name of the new project. In our case it will be `modulator_sozius_no_intc`.

*Note:* In our design we will create two application projects. One will be without interrupt controller (`modulator_sozius_no_intc`) and the second one will be with interrupt controller (`modulator_sozius_intc`).

- Select the location for the project. If you want to leave default location as it is displayed in the **Location** field, leave the **Use default location** check box selected. Otherwise, type or browse a directory location of your new project.

- For the **OS Platform** leave **standalone** platform selected.

- For the **Target Hardware**, leave `modulator_sozius_arm_hw_platform_0` selected as the Hardware Platform and `ps7_cortexa9_0` as the Processor.

- For the **Target Software**, choose **C** as the Language and choose **Use existing standalone_bsp_0** as the Board Support Package.

- Click **Next**.

- In the **Templates** dialog box, choose one of the available templates to generate a fully-functioning application project. SDK provides a useful sample applications, which are listed in the **Available Templates** box. Beside the Available Template box you can find a description box which gives a brief description of the selected sample application.

To create a blank C project, select **Empty Application** template.
- Click Finish to create `modulator_sozius_no_intc` application project.

After we have created Board Support Package project (`standalone_bsp_0`) and C Project (`modulator_sozius_no_intc`), both of them should appear in the SDK Project Explorer window.

Create `modulator_sozius_intc` Application Project

- Repeat the same procedure to create the other application project, named `modulator_sozius_intc`. 
After this step you should have both application projects in the SDK Project Explorer window.

### 3.3 Creating a C/C++ Source Files for Sozius Board Based Hardware Platform

**Creating a C/C++ Source Files for Sozius Board Based Hardware Platform**

Now it’s time to start writing the software for this project.

With the Vivado tool we have advantage to develop software independently from the hardware, using SDK tool.

To create source files necessary for our embedded system that will be running on the Sozius board, we must create a software component.

This application specific software will be executed on the ARM processor that is already part of our hardware platform.

As we already said in the previous sub-chapter, we will develop two application projects. One will be without interrupt controller (*modulator_sozius_no_intc*), and the second one will be with interrupt controller (*modulator_sozius_intc*). The idea was to illustrate how the same problem can be solved in a number of different ways.

In the ARM-based processor system we will also use UART Controller, that is integral part of the Zynq7 processing system, to transmit debug and system status information during application execution to the attached PC. This will be achieved using *xil_printf* function.

**Modulator Design without Interrupt Controller**

Source files that will be created for *modulator_sozius_no_intc* application project are:

- *modulator_sozius_no_intc.c*
- *modulator.h*
- *init_sim.c*

Create Necessary Source Files for *modulator_sozius_no_intc* Application Project

To create source files necessary for *modulator_sozius_no_intc* application project, please do the following:

- Expand *modulator_sozius_no_intc* application project in Project Explorer and src folder should appear.

In the src folder you should find your source code after creation.

- Right-click on the src folder and select New -> Source File option.

- In the New Source File dialog box:

  - check Source folder field,
  - enter Source file name (in our case it will be *modulator_sozius_no_intc.c*),
  - and use "Default C source template" as Template option.
- Click **Finish** and your `modulator_sozius_no_intc.c` source file should appear in the `src` folder, as we already said.

- Double-click on the `modulator_sozius_no_intc.c` source file in the Project Explorer and it will be immediately opened.

- Copy your source code in it, or write directly in it.

- When you finished with all modifications, click **Save** and SDK will automatically build your application.

- Repeat the same procedure to also create `modulator.h` and `init_sin.c` source files.

*Note:* The complete source files for `modulator_sozius_no_intc.c`, `modulator.h` and `init_sin.c` you can find in the text below.

```c
// modulator_sozius_no_intc.c:

#include "xparameters.h"
#include "xgpiops.h"
#include "xstatus.h"
#include "xttcp.h"
#include "modulator.h"
#include "xil_printf.h"

#define TTC_DEVICE_ID XPAR_XTTCP_0_DEVICE_ID

// Definitions of actual pin locations for LED and push-button on the sozius board
#define ps_sw2_a 10
#define ps_sw2_b 11
#define ps_sw3_a 12

// New type definition that will hold all relevant configuration parameters for the TTC module
typedef struct {
    u32 OutputHz; /* Output frequency */
    u16 Interval; /* Interval value */
    u8 Prescaler; /* Prescaler value */
    u16 Options; /* Option settings */
} TmrCntrSetup;

// Instance of a TmrCntrSetup type, holding TTC0 configuration parameters that will be used in modulator example
static TmrCntrSetup SettingsTable[1] = {
    {10, 65000, 2, 0}, /* Ticker timer counter initial setup, only output freq */
};

int main(void)
{
    /*************** Variable Definitions ***************/
    XGpioPs GpioLeds; // XGPIO instance that will be used to work with LED
    XGpioPs_Config *GPIOConfigPtr;
    XTTcp Ps_Config *Config; // Pointer to a XTTcp_Config type
    XTTcp_TimerInst; // TIMERS instance
    XTTcp_S *Timer; // Pointer to a XTTcp type
    TmrCntrSetup *TimerSetup; // Pointer to a TmrCntrSetup type
```

Figure 3.11: New Source File dialog box
```c
int count_depth = 0; // Counter for sine samples
int sw0, prev_sw0; // Switch used for selecting frequency
int scaling_factor; // Will be used to represent the SCALING_FACTOR_0 or SCALING_FACTOR_1 value
unsigned int end_time; // Will be used to represent the END_TIME_0 or END_TIME_1 value
unsigned int threshold; // Will be used to represent the current value of the sine signal
u16 current_time; // Represents the current timer value

// Sine amplitude values that will be used to generate the PWM signal
static unsigned int sine_ampl[NUM_SAMPLES];

/******************** Initialization **************************/

// LEDs initialization
GPIOConfigPtr = XGpioPs_LookupConfig(XPAR_PS7_GPIO_0_DEVICE_ID);
XGpioPs_CfgInitialize(&GpioLeds, GPIOConfigPtr, GPIOConfigPtr ->BaseAddr);

// Set the direction of the GPIO port connected to the push button to INPUT
XGpioPs_SetDirectionPin(&GpioLeds, ps_sw3_a, 0);

// TIMER initialization
TimerSetup = &SettingsTable[TTC_DEVICE_ID];
Config = XTtcPs_LookupConfig(TTC_DEVICE_ID);
Timer = &TimerInst;
Timer->Config = *Config;
XTtcPs_Stop(Timer);
XTtcPs_CfgInitialize(Timer, Config, Config->BaseAddress);
TimerSetup->Options |= (XTTCPS_OPTION_INTERVAL_MODE | XTTCPS_OPTION_WAVE_DISABLE);
XTtcPs_SetOptions(Timer, TimerSetup->Options);
XTtcPs_SetInterval(Timer, TimerSetup->Interval);
XTtcPs_SetPrescaler(Timer, TimerSetup->Prescaler);

// Sine amplitude values initialization
init_sin_f (sine_ampl);
threshold = (SCALING_FACTOR_0/8) * sine_ampl[count_depth];

/******************** Main Loop *******************************/

xil_printf("Entering main loop.
");
while(1)
{
// Start the timer
XTtcPs_Start(Timer);

// Check the switch position
if (sw0 != prev_sw0)
{
    prev_sw0 = sw0;
    sw0 = XGpioPs_ReadPin (&GpioLeds, ps_sw0_a);
    if ((sw0 & SWITCH_B) == 0) // Masking (we want to check the status of SW0 only)
    {
        end_time = END_TIME_0;
        // We must further divide the end_time with the factor of 8, because we use pre-scaler set to value 8
        end_time = end_time/8;
        // We must further divide the scaling_factor with the factor of 8, because we use pre-scaler set to value 8
        scaling_factor = SCALING_FACTOR_0/8;
    }
    else
    {
        end_time = END_TIME_1;
        end_time = end_time/8;
        scaling_factor = SCALING_FACTOR_1/8;
    }

    // Send current system status information to the terminal using UART
    if (sw0 != prev_sw0)
        // Send current system status information to the terminal using UART
        xil_printf("Sending new state information to the terminal using UART.
");
}
```
if ((sw0 & SWITCH_POS) == 0)
  xil_printf("User selected PWM signal generation with 1 Hz frequency.
");
else
  xil_printf("User selected PWM signal generation with 3.5 Hz frequency.
");
prev_sw0 = sw0;

// Turn on the LED
XGpioPs_WritePin(&GpioLeds, ps_sw2_a, 0x1);

// Pause
while(current_time < threshold);

// Turn off the LED
XGpioPs_WritePin(&GpioLeds, ps_sw2_a, 0x0);

// Pause
while(current_time < end_time);

// Reset the timer
ITtcPs_ResetCounterValue(Timer);

count_depth ++;
if (count_depth == COUNT_DEPTH_END)
  count_depth = 0;

threshold = scaling_factor * sine_ampl[count_depth];
// We must multiply current amplitude value of the sine signal with the scaling_factor
// (48640/4096=11) to "stretch" the range from (0 - 2^width(=4096)) to (0 - 48640)

return 0;

modulator.h:

#ifndef MODULATOR_H_
#define MODULATOR_H_
#include "math.h"

/********************** Constant Definitions ***********************/
#define LED_CHANNEL 1 // GPIO channel (1 or 2) to operate on
#define SWITCH_CHANNEL 2 // GPIO channel (1 or 2) to operate on
#define AXI_TIMER_0 0 // timer counter of the device to operate on
#define SYS_CLK_MHZ 100 // 100 MHz system clock
#define CLOCK_RATE 1000000 * SYS_CLK_MHZ // system clock value, expressed in Hz
// CLOCK_RATE = 100000000 Hz (= 100 MHz)
#define F_LOW 1.0 // F_LOW = 1 Hz
#define F_HIGH 3.5 // F_HIGH = 3.5 Hz
#define DEPTH 8 // the number of samples in one period of the signal (2^8=256)
#define WIDTH 12 // the number of bits used to represent amplitude value (2^12=4096)
#define C1 (1 << DEPTH) // 2^DEPTH
#define C2 (1 << (WIDTH-1)) // 2^(WIDTH-1)
#define C3 (1 << WIDTH) // 2^WIDTH

#define ENO_TIME_0 0 // input clock division factor, when sw0 = 0 (F_LOW = 1 Hz)
#define ENO_TIME_1 1 // input clock division factor, when sw0 = 1 (F_HIGH = 3.5 Hz)

#define SCALING_FACTOR_0 roundf (C / F_LOW) // SCALING_FACTOR_0 = C / F_LOW (=95)
#define SCALING_FACTOR_1 roundf (C / F_HIGH) // SCALING_FACTOR_1 = C / F_HIGH (=27)

#define SWITCH_POS 0x01 // mask to select switch position
#define LED_POS 0x01 // mask to select led position
#define COUNT_DEPTH_END 1 << DEPTH // final threshold value for the depth counter (2^8=256)

void init_sin_f (unsigned int *sine_ampl);
#endif /* MODULATOR_H_ */

init_sin.c:

#include "modulator.h"

void init_sin_f (unsigned int *sine_ampl)
{
    int i;
    float pi = 4.0*atan(1.0); // pi=3.14...

    for (i=0; i<256; i++)
    //sine_ampl[i] = sin(2*pi*i/pow(2,depth)) * (pow(2,width-1)-1) + pow(2.0,width-1)-1;
    sine_ampl[i] = (sin(2*pi*i/C1) * (C2-1) + C2 - 1); // [sin(2*pi*i/N)*(2^(width-1)-1)] + [2^(width-1)1], N = 2^depth
}

Additional Steps

In case of ARM-based design following additional steps must be performed:

- Select modulator_sozius_no_intc application project, right-click on it and choose C/C++ Build Settings option.

- In the Properties for modulator_sozius_no_intc dialog box choose C/C++ Build -> Settings option.

- In the Properties for modulator_sozius_no_intc dialog box choose C/C++ Build -> Settings option.

- In the Settings dialog box, select Tool Settings tab and under the ARM v7 gcc linker select Libraries option.

- In the Libraries (-l) window click on the Add... icon.

- In the Enter Value dialog box, type m and click OK to add math library in the Libraries list.

![Figure 3.12: Adding math library to Libraries list](image)

- In the Properties for modulator_sozius_no_intc dialog box, click OK.
As you can see from the source code above, we have used a lot of different functions.

- For LEDs and switches, we used:
  - `XGpioPs__LookupConfig (u16 DeviceId)`
  - `XGpioPs__CfgInitialize (XGpioPs * InstancePtr, XGpioPs__Config* Config)`
  - `XGpioPs__SetDirectionPin (XGpioPs* InstancePtr, u32 Pin, u32 Direction)`
  - `XGpioPs__SetOutputEnablePin (XGpioPs* InstancePtr, u32 Pin, u32 OpEnable)`
  - `XGpioPs__WritePin (XGpioPs* InstancePtr, u32 Pin, u32 Data)`
- For Triple Timer Counter (TTC), we used:
  - `XTtcPs__LookupConfig (u16 DeviceId)`
  - `XTtcPs__CfgInitialize (XTtcPs* InstancePtr, XTtcPs__Config* ConfigPtr, u32 EffectiveAddr)`
  - `XTtcPs__SetOptions (XTtcPs* InstancePtr, u32 Options)`
  - `XTtcPs__SetInterval (InstancePtr, Value) InstWriteReg((InstancePtr), XTTCPS__INTERVAL__VAL__OFFSET, (Value))`
  - `XTtcPs__SetPrescalar (XTtcPs* InstancePtr, u8 PrescalerValue)`
  - `XTtcPs__Stop (InstancePtr)`
  - `XTtcPs__Start (InstancePtr)`
  - `XTtcPs__GetCounterValue (InstancePtr) (u16)InstReadReg((InstancePtr), XTTCPS__COUNT__VALUE__OFFSET)`
  - `XTtcPs__ResetCounterValue (InstancePtr)`
All of these functions and its definitions and explanations, you can find in the Xilinx directory:

Xilinx / SDK / 2018.3 / data / embeddedsw / XilinxProcessorIPLib / drivers / gpios_v... (or ttcp_v... or some other peripheral) / doc / html / api / xgpios_8h.html (or xttcp_8h.html).

There, you can find a plenty of different functions that you can use in your software design. In this tutorial we have represent just those functions that we have used in our software design. Here are some of them:

- **XGpioPs_LookupConfig** (u16 DeviceId)

  ```c
  #include <xgpios.h>
  ```

  This function looks for the device configuration based on the unique device ID.

  The table XGpioPs_ConfigTable[] contains the configuration information for each device in the system.

  **Parameters:**
  - DeviceId - is the unique device ID of the device being looked up.

  **Returns:** A pointer to the configuration table entry corresponding to the given device ID, or NULL if no match is found.

  **Note:** None.

- **XGpioPs_CfgInitialize** (XGpioPs* InstancePtr, XGpioPs_Config* Config, UINTPTR EffectiveAddr)

  ```c
  #include <xgpios.c>
  ```

  Initialize the XGpioPs instance provided by the caller based on the given configuration data.

  Nothing is done except to initialize the InstancePtr.

  **Parameters:**
  - InstancePtr - is a pointer to an XGpioPs instance. The memory the pointer references must be pre-allocated by the caller. Further calls to manipulate the driver through the XGpioPs API must be made with this pointer.
  - Config - is a reference to a structure containing information about a specific GPIO device. This function initializes an InstancePtr object for a specific device specified by the contents of Config. This function can initialize multiple instance objects with the use of multiple calls giving different Config information on each call.
  - EffectiveAddr - is the device base address in the virtual memory address space. The caller is responsible for keeping the address mapping from EffectiveAddr to the device physical base address unchanged once this function is invoked. Unexpected errors may occur if the address mapping changes after this function is called. If address translation is not used, use Config->BaseAddress for this parameters, passing the physical address instead.

  **Returns:** XST_SUCCESS if the initialization is successful.

  **Note:** None.

  Referenced by XGpioPs_Initialize().

- **XGpioPs_SetDirectionPin** (XGpioPs* InstancePtr, u32 Pin, u32 Direction)

  ```c
  #include <xgpios.c>
  ```
Set the Direction of the specified pin.

**Parameters:**

- InstancePtr - is a pointer to the **XGpioPs** instance. Further calls to manipulate the driver through the **XGpioPs** API must be made with this pointer.
- Pin - is the pin number to which the Data is to be written. Valid values are 0-117 in Zynq and 0-173 in Zynq Ultrascale+ MP.
- Direction - is the direction to be set for the specified pin. Valid values are 0 for Input Direction, 1 for Output Direction.

**Returns:** None.

**Note:** None.

**References:** *XGpioPs::IsReady*.

- **XGpioPs::SetOutputEnablePin** *(XGpioPs* InstancePtr, u32 Pin, u32 OpEnable)*

```c
#include <xgpiops.c>
```

Set the Output Enable of the specified pin.

**Parameters:**

- InstancePtr - is a pointer to the **XGpioPs** instance. Further calls to manipulate the driver through the **XGpioPs** API must be made with this pointer.
- Pin - is the pin number to which the Data is to be written. Valid values are 0-117 in Zynq and 0-173 in Zynq Ultrascale+ MP.
- OpEnable - specifies whether the Output Enable for the specified pin should be enabled. Valid values are 0 for Disabling Output Enable, 1 for Enabling Output Enable.

**Returns:** None.

**Note:** None.

**References:** *XGpioPs::IsReady*.

- **XGpioPs::WritePin** *(XGpioPs* InstancePtr, u32 Pin, u32 Data)*

```c
#include <xgpiops.c>
```

Write data to the specified pin.

**Parameters:**

- InstancePtr - is a pointer to the **XGpioPs** instance. Further calls to manipulate the driver through the **XGpioPs** API must be made with this pointer.
- Pin - is the pin number to which the Data is to be written. Valid values are 0-117 in Zynq and 0-173 in Zynq Ultrascale+ MP.
- Data - is the data to be written to the specified pin (0 or 1).

**Returns:** None.

**Note:** This function does a masked write to the specified pin of the specified GPIO bank. The previous state of other pins is maintained.

**References:** *XGpioPs::IsReady*.
• **XGpioPs_ReadPin** (XGpioPs* InstancePtr, u32 Pin)

```c
#include <xgpio.h>

Read Data from the specified pin.

**Parameters:**
- InstancePtr - is a pointer to the XGpioPs instance. Further calls to manipulate the driver through the XGpioPs API must be made with this pointer.
- Pin - is the pin number for which the data has to be read. Valid values are 0-117 in Zynq and 0-173 in Zynq Ultrascale+ MP. See xgpio.h for the mapping of the pin numbers in the banks.

**Returns:** Current value of the Pin (0 or 1).

**Note:** This function is used for reading the state of the specified GPIO pin.

**References:** XGpioPs::IsReady.

• **XTtcPs_LookupConfig** (u16 DeviceId)

```c
#include <xttcp.h>

Looks up the device configuration based on the unique device ID.

A table contains the configuration info for each device in the system.

**Parameters:**
- DeviceId - contains the unique ID of the device.

**Returns:** A pointer to the configuration found or NULL if the specified device ID was not found. See xttcp.h for the definition of XTtcPs_Config.

**Note:** None.

• **XTtcPs_CfgInitialize** (XTtcPs* InstancePtr, XTtcPs_Config* ConfigPtr, u32 EffectiveAddr)

```c
#include <xttcp.h>

Initializes a specific XTtcPs instance such that the driver is ready to use.

This function initializes a single timer counter in the triple timer counter function block.

The state of the device after initialization is:
- Overflow Mode
- Internal (pclk) selected
- Counter disabled
- All Interrupts disabled
- Output waveforms disabled

**Parameters:**
- InstancePtr - is a pointer to the XTtcPs instance.
- ConfigPtr - is a reference to a structure containing information about a specific TTC device.
- EffectiveAddr - is the device base address in the virtual memory address space. The caller is responsible for keeping the address mapping from EffectiveAddr to the device physical base address unchanged once this function is invoked. Unexpected errors may occur if the address mapping changes after this function is called. If address translation is not used, then use ConfigPtr->BaseAddress for this parameter, passing the physical address instead.

**Returns:**
- XST_SUCCESS if the initialization is successful.
- XST_DEVICE_IS_STARTED if the device is started. It must be stopped to re-initialize.

**Note:** Device has to be stopped first to call this function to initialize it.

**References:** XTtcPs_Cong::BaseAddress, XTtcPs::Config, XTtcPs::Config::DeviceId, XTtcPs::Config::InputClockHz, XTtcPs::IsReady, XTTCPS_CLK_CNTRL_OFFSET, XTTCPS_CNT_CNTRL_OFFSET, XTTCPS_CNT_CNTRL_RESET_VALUE, XTTCPS_IER_OFFSET, XTTCPS_INTERVAL_VAL_OFFSET, XTTCPS_ISR_OFFSET, XTTCPs_IsStarted, XTTCPS_IXR_ALL_MASK, XTTCPS_MATCH_1_OFFSET, XTTCPS_MATCH_2_OFFSET, XTtcPs_ResetCounterValue, XTtcPs_Stop, and XTtcPs_WriteReg.

**XTtcPs_SetOptions** (XTtcPs* InstancePtr, u32 Options)

```c
#include "xttcps.h"
```

This function sets the options for the TTC device.

**Parameters:**
- InstancePtr - is a pointer to the XTtcPs instance.
- Options - contains the specified options to be set. This is a bit mask where a 1 means to turn the option on, and a 0 means to turn the option off. One or more bit values may be contained in the mask. See the bit definitions named XTTCPS_*OPTION in the file xttcps.h.

**Returns:**
- XST_SUCCESS if options are successfully set.
- XST_FAILURE if any of the options are unknown.

**Note:** None.

**References:** XTtcPs_Cong::BaseAddress, XTtcPs::Config, XTtcPs::IsReady, XTTCPS_CLK_CNTRL_OFFSET, XTTCPS_CNT_CNTRL_OFFSET, and XTtcPs_ReadReg.

**XTtcPs_SetInterval** (InstancePtr, Value) InstWriteReg((InstancePtr), XTTCPS_INTERVAL_VAL_OFFSET, (Value))

```c
#include "xttcps.h"
```

This function sets the interval value to be used in interval mode.

**Parameters:**
- InstancePtr - is a pointer to the XTtcPs instance.
- Value - is the 16-bit value to be set in the interval register.

**Returns:** None.

**Note:** C-style signature: void XTtcPs_SetInterval(XTtcPs *InstancePtr, u16 Value).
• `XTtcPs_SetPrescalar (XTtcPs* InstancePtr, u8 PrescalerValue)`

```c
#include <xttcps.h>
```

This function sets the prescaler enable bit and if needed sets the prescaler bits in the control register.

**Parameters:**
- `InstancePtr` - is a pointer to the `XTtcPs` instance.
- `PrescalerValue` - is a number from 0-16 that sets the prescaler to use. If the parameter is 0-15, use a prescaler on the clock of \(2^{\text{PrescalerValue} + 1}\), or 2-65536. If the parameter is `XTTCPS_CLK_CNTRL_PS_DISABLE`, do not use a prescaler.

**Returns:** None.

**Note:** None.

**References:** `XTtcPs_Cong::BaseAddress`, `XTtcPs::Cong`, `XTtcPs::IsReady`, `XTTCPS_CLK_CNTRL_OFFSET`, `XTTCPS_CLK_CNTRL_PS_DISABLE`, `XTTCPS_CLK_CNTRL_PS_EN_MASK`, `XTTCPS_CLK_CNTRL_PS_VAL_MASK`, `XTTCPS_CLK_CNTRL_PS_VAL_SHIFT`, `XTtcPs_ReadReg`, and `XTtcPs_WriteReg`.

• `XTtcPs_Stop (InstancePtr)`

```c
#include <xttcps.h>
```

**Value:** `InstWriteReg((InstancePtr), XTTCPS_CNT_CNTRL_OFFSET, \ 
(InstReadReg((InstancePtr), XTTCPS_CNT_CNTRL_OFFSET) | \ 
XTTCPS_CNT_CNTRL_DIS_MASK))`

This function stops the counter/timer.

This macro may be called at any time to stop the counter. The counter holds the last value until it is reset, restarted or enabled.

**Parameters:**
- `InstancePtr` - is a pointer to the `XTtcPs` instance.

**Returns:** None.

**Note:** C-style signature: `void XTtcPs_Stop(XTtcPs *InstancePtr)`.

**Referenced by:** `XTtcPs_CfgInitialize()`.

• `XTtcPs_Start (InstancePtr)`

```c
#include <xttcps.h>
```

**Value:** `InstWriteReg((InstancePtr), XTTCPS_CNT_CNTRL_OFFSET, \ 
(InstReadReg((InstancePtr), XTTCPS_CNT_CNTRL_OFFSET) & \ 
~XTTCPS_CNT_CNTRL_DIS_MASK))`

This function starts the counter/timer without resetting the counter value.

**Parameters:**
- `InstancePtr` - is a pointer to the `XTtcPs` instance.
Returns: None.

Note: C-style signature: void XTtcPs_Start(XTtcPs *InstancePtr).

- **XTtcPs_GetCounterValue** (InstancePtr) (u16)InstReadReg((InstancePtr), XTTCPS_COUNT_VALUE_OFFSET)

  #include <xttcpss.h>

  This function returns the current 16-bit counter value.

  It may be called at any time.

  Parameters:
  
  - InstancePtr - is a pointer to the XTtcPs instance.

  Returns: zynq: 16 bit counter value. zynq ultrascale+mpsoc: 32 bit counter value.

  Note: C-style signature: zynq: u16 XTtcPs_GetCounterValue(XTtcPs *InstancePtr) zynq ultrascale+mpsoc: u32 XTtcPs_GetCounterValue(XTtcPs *InstancePtr).

- **XTtcPs_ResetCounterValue** (InstancePtr)

  #include <xttcpss.h>

  Value: InstWriteReg((InstancePtr), XTTCPS_CNT_CNTRL_OFFSET, \ 
          (InstReadReg((InstancePtr), XTTCPS_CNT_CNTRL_OFFSET) | \ 
          (u32)XTTCPS_CNT_CNTRL_RST_MASK))

  This macro resets the count register.

  It may be called at any time. The counter is reset to either 0 or 0xFFFF, or the interval value, depending on the increment/decrement mode. The state of the counter, as started or stopped, is not affected by calling reset.

  Parameters:

  - InstancePtr - is a pointer to the XTtcPs instance.

  Returns: None.

  Note: C-style signature: void XTtcPs_ResetCounterValue(XTtcPs *InstancePtr).

  Referenced by XTtcPs_CfgInitialize().

Modulator Design with Interrupt Controller

Source files that will be created for modulator_sozius_intc application project are:

- modulator_sozius_intc.c
- modulator.h
- init_sin.c

modulator.h and init_sin.c source files are the same, only modulator sozius_intc.c source file is different.
CHAPTER 3. CREATING THE SOFTWARE PLATFORM USING SDK

Create Necessary Source Files for modulator_sozius_intc Application Project

To create source files necessary for modulator_sozius_intc application project, please do the following:

- Expand modulator_sozius_intc application project in Project Explorer and src folder should appear. In the src folder you should find your source code after creation.

- Repeat steps from slides 20, 21, 22.

Note: In the step 3 use modulator_sozius_intc.c as the file name for the C source code file.

modulator.h - this file is identical with the modulator.h file used in the design without interrupt controller.

init_sin.c - this file is identical with the init_sin.c file used in the design without interrupt controller.

- Final step requires adding a math library to the ARM v7 gcc linker library settings. To include math library, please repeat steps from the end of the previous "Modulator design without interrupt controller" section, from the slides 32, 33, 34.

modulator_sozius_intc.c:

```c
#include "xparameters.h"
#include "xparameters_ps.h"
#include "xgpiops.h"
#include "xstatus.h"
#include "xttcps.h"
#include "xscugic.h"
#include "xil_exception.h"
#include "modulator.h"
#include "xil_printf.h"

#define TTC_DEVICE_ID XPAR_XTTCPS_0_DEVICE_ID
#define TTC_INTR_ID XPAR_XTTCPS_0_INTR

// Definitions of actual pin locations for LED and push-button on the socius board
#define ps_sw2_a 10
#define ps_sw2_b 11
#define ps_sw3_a 12

// New type definition that will hold all relevant configuration parameters for the TTC module
typedef struct {
  u32 OutputHz; /* Output frequency */
  u16 Interval; /* Interval value */
  u8 Prescaler; /* Prescaler value */
  u16 Options; /* Option settings */
} TmrCntrSetup;

// Instance of a TmrCntrSetup type, holding TTC0 configuration parameters that will be used in modulator example
static TmrCntrSetup SettingsTable[1] = {
  {10, 65000, 2, 0}, /* Ticker timer counter initial setup, only output freq */
};

// global variables necessary for the Global Interrupt Controller initialization
XScuGic INTCInst;
XScuGic_Config *IntcConfig;

static int interrupt_occurred = 0; // variable which will signal when that interrupt has occurred

// Interrupt handler for the timer
void Timer InterruptHandler(void *CallBackRef)
{
  u32 StatusEvent;

  interrupt_occurred = 1;
  // stop timer
  XTtcPs_Stop((XTtcPs *)CallBackRef);
  StatusEvent = XTtcPs_GetInterruptStatus((XTtcPs *)CallBackRef);
  XTtcPs_ClearInterruptStatus((XTtcPs *)CallBackRef, StatusEvent);
}

// Interrupt system initialization function
int InitTxCpsFunction(u16 DeviceId, XTtcPs *TtcPsInt)
{
  int status;
  // Interrupt controller initialization
```
CHAPTER 3. CREATING THE SOFTWARE PLATFORM USING SDK

int IntcConfig = XScuGic_LookupConfig(DeviceId);
status = XScuGic_CfgInitialize(&INTCInst, IntcConfig, IntcConfig->CpuBaseAddress);
if(status != XST_SUCCESS) return XST_FAILURE;

// Connect the interrupt controller interrupt handler to the hardware interrupt
// handling logic in the ARM
Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
(Xil_ExceptionHandler)XScuGic_InterruptHandler,
&INTCInst);

// Enable interrupt controller
Xil_ExceptionEnable();

// Connect timer interrupt to device driver handler
status = XScuGic_Connect(&INTCInst,
TTC_INTR_ID,
(Xil_ExceptionHandler)Timer_InterruptHandler,
(void *)(TtcPsInt);
if(status != XST_SUCCESS) return XST_FAILURE;

// Enable timer interrupt in the interrupt controller
XScuGic_Enable(&INTCInst, TTC_INTR_ID);

// Enable timer interrupt generation in the timer module
XscuGic_EnableInters(TtcPsInt, XTTCPS_IXR_ALL_MASK);

return XST_SUCCESS;

int main(void)
{
/*********************** Variable Definitions *******************/
XGpioPs GpioLeds; // XGPIO instance that will be used to work with LED
XGpioPs_Config *GPIOConfigPtr;
XTtcPs_Config *Config; // Pointer to a XTtcPs_Config type
XTtcPs TimerInst; // TIMER instance
XTtcPs *Timer; // Pointer to a XTtcPs type
TmrCntrSetup *TimerSetup; // Pointer to a TmrCntrSetup type
int count_depth = 0; // Counter for sine samples
int sw0, prev_sw0; // Switch used for selecting frequency
int scaling_factor; // Will be used to represent the SCALING_FACTOR_0 or SCALING_FACTOR_1 value
unsigned int end_time; // Will be used to represent the END_TIME_0 or END_TIME_1 value
unsigned int threshold; // Will be used to represent the current value of the sine signal
unsigned int sine_ampl[COUNT_DEPTH_END]; // Sine amplitude values that will be used to generate the PWM signal
int led_state; // Current state of the LED
int reset_value_0; // Is the value for timer to now from which value will start counting downwards
int reset_value_1; // Is the value for timer to now from which value will start counting downwards
int status; // Interrupt initialization function return status variable

/*********************** Initialization **************************
xil_printf("Initializing peripherals\r\n");
XGpioPs_Config *GPIOConfigPtr = XGpioPs_LookupConfig(XPAR_PS7_GPIO_0_DEVICE_ID);
XGpioPs_CfgInitialize(&GpioLeds, GPIOConfigPtr, GPIOConfigPtr->BaseAddr);

// On the socius board we must properly control both ends of a LED, hence we must use two GPIO ports
XGpioPs_SetDirectionPin(&GpioLeds, ps_sw2_a, 1);
XGpioPs_SetOutputEnablePin(&GpioLeds, ps_sw2_a, 1);
XGpioPs_SetDirectionPin(&GpioLeds, ps_sw2_b, 1);
XGpioPs_SetOutputEnablePin(&GpioLeds, ps_sw2_b, 1);

xil_printf("Initializing SWITCHes\r\n");

// Set the value of one end of LED to always be equal to zero, by changing the other end we will turn it on and off
XGpioPs_WritePin(&GpioLeds, ps_sw2_b, 0x0);

xil_printf("Initializing TIMER\r\n");

// TIMER initialization
TimerSetup = &SettingsTable[TTC_DEVICE_ID];

// Store the current configuration of the timer in a TimerInst object, using a pointer to access it
Timer = &TimerInst;
Timer->Config = &Config;

// Stop the timer

www.so-logic.net 2019/01/28 57
// Initialize the timer with the required configuration parameters
ITCtPs_CfgInitialize(Timer, Config, Config->BaseAddress);
TimerSetup->Options |= (XTTCPS_OPTION_INTERVAL_MODE |
XTTCPS_OPTION_DECREMENT |
XTTCPS_OPTION_WAVE_DISABLE);
ITCtPs_SetOptions(Timer, TimerSetup->Options);
ITCtPs_SetPrescaler(Timer, TimerSetup->Prescaler);

xil_printf("Initializing INTERRUPT CONTROLLER!\n");
// INTERRUPT CONTROLLER initialization
status = IntcInitFunction(XPAR_PS7_SCUGIC_0_DEVICE_ID, Timer);
if(status != XST_SUCCESS) return XST_FAILURE;

xil_printf("Initializing sine_ampl array!\n");
// sine_ampl array initialization
initSin_f (sine_ampl);

// Read the switch position
sw0 = XGpioPs_ReadPin (&GpioLeds, ps_sw3_a);
// Check the switch position
if ((sw0 & SWITCH_POS) == 0) // Masking (we want to check the status of SW0 only)
{
    end_time = END_TIME_0;
    // We must further divide the end_time with the factor of 8, because we use pre-scaler set to value 8
    end_time = end_time/8;
    // We must further divide the scaling_factor with the factor of 8, because we use pre-scaler set to value 8
    scaling_factor = SCALING_FACTOR_0/8;
    threshold = scaling_factor * sine_ampl[count_depth]; // threshold = current amplitude value of the sine signal
    // We must multiply current amplitude value of the sine signal with the scaling_factor
    // (48640/4096=11) to "stretch" the range from (0 - 2^width(=4096)) to (0 - 48640)
    reset_value_0 = end_time - threshold;
    reset_value_1 = threshold;
    // Set the timer countdown interval
    ITCtPs_SetInterval(Timer, reset_value_1);
    // Turn on the LED
    XGpioPs_WritePin(&GpioLeds, ps_sw2_a, 0x1);
    // Start the timer
    ITCtPs_Start(Timer);
    interrupt_occurred = 0; // Interrupt_occurred initialization
    led_state = 1; // Set the initial state of the LED
}
else
{
    end_time = END_TIME_1;
    end_time = end_time/8;
    scaling_factor = SCALING_FACTOR_1/8;
    threshold = scaling_factor * sine_ampl[count_depth]; // threshold = current amplitude value of the sine signal
    // We must multiply current amplitude value of the sine signal with the scaling_factor
    // (48640/4096=11) to "stretch" the range from (0 - 2^width(=4096)) to (0 - 48640)
    reset_value_0 = end_time - threshold;
    reset_value_1 = threshold;
    // Set the timer countdown interval
    ITCtPs_SetInterval(Timer, reset_value_1);
    // Turn on the LED
    XGpioPs_WritePin(&GpioLeds, ps_sw2_a, 0x1);
    // Start the timer
    ITCtPs_Start(Timer);
    interrupt_occurred = 0; // Interrupt_occurred initialization
    led_state = 1; // Set the initial state of the LED
}

/******* Main Loop *******

xil_printf("Entering main loop!\n");
while(1)
{
    // Check the switch position
    sw0 = XGpioPs_ReadPin (&GpioLeds, ps_sw0_a);
    // Check the switch position
    if ((sw0 & SWITCH_POS) == 0) // Masking (we want to check the status of SW0 only)
    {
        end_time = END_TIME_0;
        // We must further divide the end_time with the factor of 8, because we use pre-scaler set to value 8
        end_time = end_time/8;
        // We must further divide the scaling_factor with the factor of 8, because we use pre-scaler set to value 8
        scaling_factor = SCALING_FACTOR_0/8;
    }
    else
    {
        end_time = END_TIME_1;
        end_time = end_time/8;
        scaling_factor = SCALING_FACTOR_1/8;
    }

    // Send the current system status information to the terminal using UART
    if (sw0 != prev_sw0)
    {
        if ((sw0 & SWITCH_POS) == 0)
        {
            xil_printf("User selected PWM signal generation with 1 Hz frequency.\n");
        }
        else
        {
            xil_printf("User selected PWM signal generation with 3.5 Hz frequency.\n");
        }
        prev_sw0 = sw0;
    }
    if (interrupt_occurred == 1)
{interrupt_occurred = 0;
if (led_state == 1)
{
   // Write the starting counter value, reset_value_0 = end_time - threshold
   XTtcPs_SetInterval(Timer, reset_value_0);
   // Turn off the LED
   XGpioPs_WritePin(&GpioLeds, ps_sw2_a, 0x0);
   // Start the timer
   XTtcPs_Start(Timer);
   led_state = 0;
   count_depth ++;
   if (count_depth == COUNT_DEPTH_END)
   {
      threshold = scaling_factor * sine_ampl[count_depth];
      reset_value_0 = end_time - threshold;
      reset_value_1 = threshold;
   }
   else
   {
      // Write the starting counter value, reset_value_1 = threshold
      XTtcPs_SetInterval(Timer, reset_value_1);
      // Turn on the LED
      XGpioPs_WritePin(&GpioLeds, ps_sw2_a, 0x1);
      // Start the timer
      XTtcPs_Start(Timer);
      led_state = 1;
   }
}
return 0;
}

In the modulator_sozius_intc.c source code with the interrupt controller, we have used almost the same functions as in the modulator_sozius_no_intc.c source code without using an interrupt controller. Here are the functions that we have used in our design. Some of them are already explained below the modulator_sozius_no_intc.c source file.

- For LEDs and switches, we used:
  - XGpioPs_LookupConfig (u16 DeviceId)
  - XGpioPs_CfgInitialize (XGpioPs* InstancePtr, XGpioPs_Config* Config, UINTPTR EffectiveAddr)
  - XGpioPs_SetDirectionPin (XGpioPs* InstancePtr, u32 Pin, u32 Direction)
  - XGpioPs_SetOutputEnablePin (XGpioPs* InstancePtr, u32 Pin, u32 OpEnable)
  - XGpioPs_WritePin (XGpioPs* InstancePtr, u32 Pin, u32 Data)
  - XGpioPs_ReadPin (XGpioPs* InstancePtr, u32 Pin)

- For Triple Timer Counter (TTC), we used:
  - XTtcPs_LookupConfig (u16 DeviceId)
  - XTtcPs_CfgInitialize (XTtcPs* InstancePtr, XTtcPs_Config* ConfigPtr, u32 EffectiveAddr)
  - XTtcPs_SetOptions (XTtcPs* InstancePtr, u32 Options)
  - XTtcPs_SetInterval (InstancePtr, Value) InstWriteReg((InstancePtr), XTTCPS_INTERVAL_VAL_OFFSET, (Value))
  - XTtcPs_SetPrescaler (XTtcPs* InstancePtr, u8 PrescalerValue)
  - XTtcPs_Stop (InstancePtr)
  - XTtcPs_Start (InstancePtr)
  - XTtcPs_GetCounterValue (InstancePtr) (u16)InstReadReg((InstancePtr), XTTCPS_COUNT_VALUE_OFFSET)
CHAPTER 3. CREATING THE SOFTWARE PLATFORM USING SDK

- **XTtcPs_ResetCounterValue** (InstancePtr)
- **XTtcPs_GetInterruptStatus** (InstancePtr) InstReadReg((InstancePtr), XTTCPS_ISR_OFFSET)
- **XTtcPs_ClearInterruptStatus** (InstancePtr, InterruptMask)
- **XTtcPs_EnableInterrupts** (InstancePtr, InterruptMask)

For Interrupt Controller, we used:

- **XScuGic_LookupConfig** (u16 DeviceId)
- **XScuGic_CfgInitialize** (XScuGic* InstancePtr, XScuGic_Config* ConfigPtr, u32 EffectiveAddr)
- **XScuGic_Connect** (XScuGic* InstancePtr, u32 Int_Id, Xil_InterruptHandler Handler, void* CallBackRef)
- **XScuGic_Enable** (XScuGic* InstancePtr, u32 Int_Id)

As we already said, all of these functions and it's definitions and explanations, you can find in the Xilinx directory:

Xilinx / SDK / 2018.3 / data / embeddedsw / XilinxProcessorIPLib / drivers / gpios_v... (or ttcp_v... or scugic_v or some other peripheral) / doc / html / api / xgpios_8h.html (or ttcp_8h.html or xscugic_8h.html).

- **XTtcPs_GetInterruptStatus** (InstancePtr) InstReadReg((InstancePtr), XTTCPS_ISR_OFFSET)

  #include <xttcps.h>

  This function reads the interrupt status.

  **Parameters**: InstancePtr - is a pointer to the XTtcPs instance.

  **Returns**: None.

  **Note**: C-style signature: u32 XTtcPs_GetInterruptStatus(XTtcPs *InstancePtr).

- **XTtcPs_ClearInterruptStatus** (InstancePtr, InterruptMask)

  #include <xttcps.h>

  **Value**: InstWriteReg((InstancePtr), XTTCPS_ISR_OFFSET, \ (InterruptMask))

  This function clears the interrupt status.

  **Parameters**:
  - InstancePtr - is a pointer to the XTtcPs instance.
  - InterruptMask - defines which interrupt should be cleared. Constants are defined in xttcps_hw.h as XTTCPS_IXR_. This is a bit mask, all set bits will be cleared, cleared bits will not be cleared.

  **Returns**: None.

  **Note**: C-style signature: void XTtcPs_ClearInterruptStatus(XTtcPs *InstancePtr, u32 InterruptMask).

- **XTtcPs_EnableInterrupts** (InstancePtr, InterruptMask)

  #include <xttcps.h>
Value: `InstWriteReg((InstancePtr), XTTCPS_IER_OFFSET, \
       (InstReadReg((InstancePtr), XTTCPS_IER_OFFSET) | \
       (InterruptMask)))`

This function enables the interrupts.

Parameters:
- `InstancePtr` - is a pointer to the `XTtcPs` instance.
- `InterruptMask` - defines which interrupt should be enabled. Constants are defined in `xttcpshw.h` as `XTTCPS_IXR_*`. This is a bit mask, all set bits will be enabled, cleared bits will not be disabled.

Returns: None.

Note: C-style signature: `void XTtcPs_EnableInterrupts(XTtcPs *InstancePtr, u32 InterruptMask)`.

If you dive deeper into the `modulator_modulos_intc.c` source code, you can notice that additional code has been included before the main. The function `IntcInitFunction (u16 DeviceId, XtmrCtr *TmrInstancePtr)` is necessary and contains additional code to:

- initialize interrupt controller
- connect the interrupt controller interrupt handler to the hardware interrupt handling logic in the ARM processor
- enable the interrupt controller
- connect a timer device driver handler that will be called when an interrupt for the timer occurs. This device driver handler performs the specific interrupt processing for the device.
- Enable timer interrupt in the interrupt controller

As you can see from the code above, in the `IntcInitFunction (u16 DeviceId, XtmrCtr *TmrInstancePtr)` definition we have used some new functions:

- `XScuGic_LookupConfig (u16 DeviceId)`

  ```c
  #include <xscugic.h>
  ```

  Looks up the device configuration based on the unique device ID.

  A table contains the configuration info for each device in the system.

  Parameters:
  - `DeviceId` - is the unique identifier for a device.

  Returns: A pointer to the `XScuGic` configuration structure for the specified device, or NULL if the device was not found.

  Note: None.

  Referenced by `ScuGicExample()`.

- `XScuGic_CfgInitialize (XScuGic* InstancePtr, XScuGic_Config* ConfigPtr, u32 EffectiveAddr)`

  ```c
  #include <xscugic.c>
  ```

  CfgInitialize a specific interrupt controller instance/driver.

  The initialization entails:
- Initialize fields of the XScuGic structure
- Initial vector table with stub function calls
- All interrupt sources are disabled

**Parameters:**
- InstancePtr - is a pointer to the XScuGic instance.
- ConfigPtr - is a pointer to a config table for the particular device this driver is associated with.
- EffectiveAddr - is the device base address in the virtual memory address space. The caller is responsible for keeping the address mapping from EffectiveAddr to the device physical base address unchanged once this function is invoked. Unexpected errors may occur if the address mapping changes after this function is called. If address translation is not used, use Config->BaseAddress for this parameter, passing the physical address instead.

**Returns:** XST_SUCCESS if initialization was successful.

**Note:** None.

Referenced by ScuGicExample().

- **XScuGic_Connect** (XScuGic* InstancePtr, u32 Int_Id, Xil_InterruptHandler Handler, void* CallBackRef)

```
#include <xscugic.c>
```

Makes the connection between the Int_Id of the interrupt source and the associated handler that is to run when the interrupt is recognized.

The argument provided in this call as the Callbackref is used as the argument for the handler when it is called.

**Parameters:**
- InstancePtr - is a pointer to the XScuGic instance.
- Int_Id - contains the ID of the interrupt source and should be in the range of 0 to XSCUGIC_MAX_NUM_INTR_INPUTS - 1.
- Handler - to the handler for that interrupt.
- CallBackRef - is the callback reference, usually the instance pointer of the connecting driver.

**Returns:** XST_SUCCESS if the handler was connected correctly.

**Note:** WARNING: The handler provided as an argument will overwrite any handler that was previously connected.

**References:** XScuGic::Config, XScuGic_Config::HandlerTable, and XScuGic::IsReady.

Referenced by ScuGicExample().

- **XScuGic_Enable** (XScuGic* InstancePtr, u32 Int_Id)

```
#include <xscugic.c>
```

Enables the interrupt source provided as the argument Int_Id.

Any pending interrupt condition for the specified Int_Id will occur after this function is called.

**Parameters:**

---

---
- InstancePtr - is a pointer to the XScuGic instance.

- int Id - contains the ID of the interrupt source and should be in the range of 0 to XSCUGIC_MAX_NUM_INTR_INPUTS - 1

Returns: None.

Note: None.

References: XScuGic::IsReady, XScuGic_DistWriteReg, and XSCUGIC_ENABLE_SET_OFFSET.

3.4 Viewing and Configuring Linker Script file

Viewing and Configuring Linker Script File

A linker is a program that takes one or more object files (.o) generated by a compiler and combines them into a single executable (.elf) file.

![Diagram of Linking and Locating process](image)

Figure 3.14: Linking and Locating process

Linker program combines all files from the application project into the executable .elf file. This process is controlled by the Linker Script file.

Elf file is organized by logical section from each object file. Each section is located in a physical memory space as defined by the linker script. Relocatable symbols are resolved to their physical addresses. Other symbols, such as those for debugging are also added to the .elf file.
When the linker executes, it first combines all of the object sections. Then it resolves addresses and writes LDL files, see Figure 3.15.

**Linker Script**

Linker script controls the linking process. It maps the code and data to a specified memory space, sets the entry point to the executable, reserve space for the heap and stack, define the layout and start address of each section.

Linker script is required if the design contains a discontinuous memory space. It has it’s own language and can be difficult to write. Because of this, Xilinx provides a Linker Script Generator.

Linker Script will be automatically generated when you create an Xilinx C Project within Xilinx SDK tool. In our case it will be in the moment when we have created `modulator_socius_no_intc` C project.

If you want to view or make some modifications to existing linker script file, please do the following:

- In the SDK Project Explorer tab, right-click on the `modulator_socius_no_intc` project, select Generate Linker Script and the Linker Script dialog box will appear, see Figure 3.16.
- If you want to modify the default settings for the Linker Script file, make the required modifications, click **Generate** button and the new linker script file will be created.

### 3.5 Building Application and Generating ELF File

In a microprocessor-based design such as a MicroBlaze design, an ELF file generated in the SDK or in some other software development tool, can be imported and associated with a block design in the Vivado IDE. A bitstream file can be generated that includes the ELF content from the Vivado IDE and run on target hardware.

To build an executable file for this application, SDK performs the following actions, see Figure 3.17.
First, SDK builds the Board Support Package (BSP) using LibGen tool. In our case it is called software platform.

Then, SDK compiles the application software using platform-specific gcc/g++ compiler.

At the end, the object files from the application and the BSP are linked together to form the final executable file (.elf file). This step is performed by a linker which takes as input a set of object files and a linker script that specifies where object files should be placed in memory.

SDK builds BSP ones, after it’s creation. For every source file modification, SDK will automatically generate a new .elf file, compiling all source files that are out of date and linking them with the BSP.

The following sections provide an overview of concepts involved in building applications:

Makefiles

Compilation of source files into object files is controlled using Makefiles. With SDK, there are two possible options for Makefiles:

1. **Managed Make**: For Managed Make projects, SDK automatically creates Makefiles. Makefiles created by SDK typically compile the sources into object files, and finally link the different object files into an executable. In most cases, managed make simply eliminates the job of writing Makefiles. This is the suggested option.

2. **Standard Make**: If you want ultimate control over the compilation process, use standard make projects. In this case, you must manually write a Makefile with steps to compile and link an application. Using the standard Make flow hides a number of dependencies from SDK, so you must follow manual steps for other tasks such as debugging or running the application from within SDK. Therefore, the Standard Make flow is not recommended for general use.
Build Configurations

Software developers typically build different versions of executables, with different settings used to build those executables. For example, an application that is built for debugging uses a certain set of options (such as compiler flags and macro definitions), while the same application is built with a different set of options for eventual release to customers. SDK makes it easier to maintain these different profiles using the concept of build configurations.

Each build configuration could customize:

- Compiler Settings: Debug and Optimization levels
- Macros passed for compilation
- Linker Settings

Build Report

When the program finishes building selected configuration, build report will be visible in the SDK's Console window, including generated code size information.

For example, in case of building `modulator_sozius_no_intc` configuration ARM microprocessor, code size report is shown on the following figure.

![Figure 3.18: Console window with code size information for the modulator_sozius_no_intc build configuration for ARM-based system](image)

As a part of building process, information on the size of your application will normally be displayed at the end of the build log of the Console view, as it is shown on the Figure 3.18. Here is the explanation for each column separately:

- **text** - shows the size of the code and read-only (constant) data in your application (in decimal).
- **data** - shows the size of the initialised data in your application (in decimal). Data counted in the "data" section is not constant, so it will end up in RAM memory. But, because this data is initialised and the initial value is constant, it will be stored in the FLASH memory. The linker allocates the space for initial data values in FLASH which are then copied to RAM in the startup code.
- **bss** - shows the size of uninitialized data in your application (in decimal). bss counts for the uninitialized data in the RAM which will be initialized to zero value in the startup code.
- **dec** - total size, "text" + "data" + "bss" (in decimal).
- **hex** - hexadecimal equivalent of "dec".

Typically,

- the FLASH consumption of your application will be "text" + "data".
- the RAM consumption of your application will be "data" + "bss".

Remember that the RAM consumption provided by this is only that of global data. It will not include any memory consumed by application stack and heap when application is actually executing.
Build Report - Comparisons

By comparing the sizes of the generated executable files for MicroBlaze and ARM processors, they differ.

This is typical if we compile the same source code targeting different processors.

The reason for this is that different tool-chains (compilers, linkers, assemblers) are used with different processors.

Furthermore, different processors have different instruction sets and micro-architectures.

All this inevitably leads to generation of different sizes of executable files when targeting different processors, even if completely identical source code is used.

When planning to migrate an existing software application to a new processor, careful considerations need to be made regarding the expected executable file size, which can either increase or decrease compared to existing solution.

The same holds for the execution time of the software application when targeting different processor platforms.

Programming Languages in Embedded Systems

Additional aspect that can also influence the size and speed of the generated software platform is the choice of the programming language used to specify the software part of an embedded system.

Currently C language dominates embedded software development, but C++ language is increasingly starting to be used also.

The reason for this is that the C++ is one of the dominant programming languages used in desktop applications, servers and networks to which an embedded system will typically interface.

Furthermore, there are many additional pros for using C++ language in embedded system development: better support for multicore programming, use of object oriented programming style, function overloaded, use of templates, etc.

However, when compiled, C++ programs tend to be bigger and slower then equivalent C programs.

In the past this was a major concern, but nowadays, with the availability of matured C++ compilers targeting embedded systems and with careful usage of advanced C++ language features this no longer needs to be or is the case. Having said this, this doesn’t imply that C language will stop being one of the major programming languages used for embedded software development in the foreseeable future, especially in the resource- and time-critical embedded applications.

3.6 Running Application

Running Application

You can run your software application on your hardware platform using SDK tool. The program will run to termination.

You can also stop the program at any time of execution.
CHAPTER 3. CREATING THE SOFTWARE PLATFORM USING SDK

Figure 3.19: Run Workflow diagram

The run workflow is described in the previous diagram, see Figure 3.19.

The workflow is made up of the following components:

- **Executable ELF File**: To debug your application, you must use a compiled Executable and Linkable Format (ELF) file.

- **Run Configuration**: To launch the run session, you must create a run configuration in SDK. This configuration captures options required to start a run session, including the executable name, processor target to run and other information.

- **JTAG Settings**: In most cases, SDK can automatically detect the JTAG settings and doesn’t require special settings.

- **Run Console**: This XMD console view enables you to stop the program execution or terminate the run session.

You can repeat the cycle of modifying the code, building the executable, and running the program in SDK. The program can be run on all supported debug targets.

Before you can run your application, you must generate netlist and bitstream file (if you didn’t generate them after the hardware platform is specified) and download the FPGA’s bitstream file to the board.

To generate netlist and bitstream file, go back to the Vivado IDE main window and follow the same steps as it is explained in the Chapter 2.3.1 Create ARM-based hardware platform for Socinus development board, steps 16, 17 and 18.

### 3.6.1 Downloading ARM-based bitstream file

**Program FPGA**

To download your ARM-based bitstream file to the target board, connect the additional USB cable that will be used to provide UART interface, that will be used during system debug, and do the following steps:
- Select Xilinx -> Program FPGA from the SDK main window.

- The Program FPGA dialog box will appear. The bitstream field should already be populated with the correct bitstream file. Click Program.

![Program FPGA dialog box](image)

Figure 3.20: Program FPGA dialog box

**Program Debugging using Terminal Window for ARM-based Design**

- At the bottom of the SDK, open SDK Terminal tab and click the green "+" button to connect with the serial port.

![SDK Terminal window](image)

Figure 3.21: SDK Terminal window

- In the Connect to serial port dialog box, in the Port field choose COM4 serial port to connect with and leave all other parameters unchanged. Click OK.

![Connect to serial port dialog box](image)

Figure 3.22: Connect to serial port dialog box
After connecting the terminal with the serial port, in the **SDK Terminal** window you should see notification about successfully connection.

![SDK Terminal window](image)

**Figure 3.23:** Terminal notification about successful connection to serial port

- When the ZynqPL is successfully configured with the bitstream file, we can now launch our software application on the Zynq PS:

  In the Project Explorer select your application project, right-click on it and select **Run As -> Launch on Hardware (System Debugger)** option.

- Open the SDK Terminal window and you should see all the messages sent by software application.

As you can see, before changing switch position, one PWM signal generation frequency is selected.

![SDK Terminal window](image)

**Figure 3.24:** Terminal window with messages sent by software application

- Change the switch position on the development board.

In case of using Sozius development board, press and hold the **push button 4** and the terminal will detect that the second PWM signal generation frequency is selected as we predicted in the software application.

When you release the push button 4, the PWM signal generation frequency will return to the value that was present before pressing the button.

![SDK Terminal window](image)

**Figure 3.25:** Terminal window with messages sent by software application after changing the switch position on the development board

On the Figure 3.26 is presented the program FPGA flow in detail.
CHAPTER 3. CREATING THE SOFTWARE PLATFORM USING SDK

Program FPGA Flow

A Block RAM Memory Map (BMM) file is a text file that has syntactic descriptions of how individual block RAMs constitute a contiguous logical data space. When updating the FPGA bitstream with memory initialization data (typically the executable program), the Data2MEM utility uses the BMM file to direct the translation of data into the proper initialization form. Although the BMM file is a text file direct editing is not recommended. This file is generated by the PlatGen (Platform Generator) tool and updated with physical location information by the BitGen (Bitstream Generator) tool.

3.7 Application Debugging

Debugging is an integral part of embedded systems development. The debugging process implies testing, stabilizing, localizing and correcting errors. There are two methods of debugging:

- **Hardware debugging** via logic probe, logic analyzer, on-circuit emulator, or background debugger
- **Software debugging** via a debugging instrument

SDK supports software debugging through:

- **GDB tools**
  
  GDB (GNU Debugger) is a powerful, flexible tool that provides a unified graphical interface for debugging and verifying MicroBlaze processor systems during various development phases. With GDB debugger you can debug programs written in C and C++.

- **Xilinx Microprocessor Debugger (XMD)**
  
  - Runs all the hardware debugging tools and communicates with the hardware
  - Shell for hardware communication
  - Tool command language (Tcl) syntax and command interpreter

- **GNU tools**
CHAPTER 3. CREATING THE SOFTWARE PLATFORM USING SDK

- Communicate with the hardware through XMD

The actual debugger is XMD. GDB is the user interface, or GUI, that talks to XMD through a TCP/IP port via Tcl commands, see Illustration 3.27.

![Figure 3.27: GDB overview](image)

The main purpose of XMD is to attach to the debug hardware interface of the embedded processor, the MicroBlaze Debug Module (MDM). This is done via an internal JTAG chain facilitated by the BSCAN component on the FPGA. The MDM also offers a JTAG uart feature that will show up as a AXI bus uart peripheral for the MicroBlaze. XMD provides many services, including download cable connection and control. One of the main functions of XMD is the debug engine. This engine provides the command interface to the processor debug hardware via a Tcl script and/or simple command line interface. You could directly debug a program from the XMD command line console, but this would be a painful process. The GDB debugger provides an easy-to-use graphical interface that interfaces Tcl with XMD.

### 3.7.1 Debug Overview

With the SDK debugger, you can see what is happening to a program while it executes. You can set breakpoints or watchpoints to stop the processor, step through program execution, view the program variables and stack, and view the contents of the memory in the system. The SDK debugger uses the GNU Debugger (GDB) with Xilinx Microprocessor Debugger (XMD) as the underlying debug engine. It translates each user interface action into a sequence of GDB commands and processes the output from GDB to display the current state of the program being debugged. It communicates to the processor on the hardware and Instruction Set Simulator (ISS) target using XMD.

![Figure 3.28: Debug Workflow diagram](image)
The workflow is made up of the following components:

- **Executable ELF File**: To debug your application, you must use an Executable and Linkable Format (ELF) file compiled for debugging. The debug ELF file contains additional debug information for the debugger to make direct associations between the source code and the binaries generated from that original source.

- **Debug Configuration**: In order to launch the debug session, you must create a debug configuration in SDK. This configuration captures options required to start a debug session, including the executable name, processor target to debug, and other information.

- **JTAG Settings**: When debugging the program on a hardware target, SDK uses XMD for communication to the processor using a JTAG interface on the board. The JTAG settings for the debug session can be specified in the JTAG Settings dialog box. In most cases, the debugger can automatically detect the JTAG settings and do not need to provide special settings.

- **SDK Debug Perspective**: Using the Debug perspective, you can manage the debugging or running of a program in the Workbench. You can control the execution of your program by setting breakpoints, suspending launched programs, stepping through your code, and examining the contents of variables.

You can repeat the cycle of modifying the code, building the executable, and debugging the program in the SDK.

*Note*: If you edit the source code after compiling, the line numbering will be out of step because the debug information is tied directly to the source. Similarly, debugging optimized binaries can also cause unexpected jumps in the execution trace.

### Hardware debug target

SDK supports debugging of a program on processor running on a FPGA. All processor architectures are supported. SDK communicates to the processor on the FPGA over the JTAG interface using the Xilinx JTAG cable. Before you debug the processor on the FPGA, you should configure the FPGA with the appropriate system bitstream.

The debug logic for each processor enables program debugging by controlling the processor execution. The debug logic on hard ARM processor cores is built in and always available for debugging. However, the debug logic on soft MicroBlaze processor cores is configurable and can be enabled or disabled by the hardware designer when building the embedded hardware.

Enabling the debug logic on MicroBlaze processors provides advanced debugging capabilities such as hardware breakpoints, read/write memory watchpoints, safe-mode debugging, and more visibility into MicroBlaze processors. This is the **recommended method** of debugging MicroBlaze software.

If the debug logic is disabled on the hardware, you can debug programs using XMDStub (a ROM monitor). XMDStub is a small debug stub that runs on MicroBlaze processors and can perform basic debug operations such as reading and writing memory and register values and controlling the program execution. It should be initialized to the processor local memory at the reset location, so when the processor resets, the XMDStub is run and ready for debugging. It communicates to XMD over a Universal Asynchronous Receiver-Transmitter (UART), which could be JTAG-based or RS232-based. This method is not supported in SDK and you should use the XMD command-line tool for debugging.

### 3.7.2 Debug Configuration

To debug, run, and profile an application, you must create a configuration that captures the settings for executing the application. The configurations for debugging, running, and profiling an application are similar.

To setup a debug configuration, do the following:

- In the SDK main window, select `modulator_sozius_no_intc` application project and select Run -> **Debug Configurations...** option, see Figure 3.29.
The another way to open Debug Configurations dialog box is to select `modulator_sozius_no_intc` project in the Project Explorer window, right-click on it and choose `Debug As --> Debug Configurations...` option.

- In the `Debug Configurations` dialog box, you can see that the SDK took has automatically created two new debug configurations, `System Debugger using Debug_modulator_sozius_intc.elf on Local` and `System Debugger using Debug_modulator_sozius_no_intc.elf on Local` Debug configurations for us, see Figure 3.30.

As you can see, you can provide an unique name for your configuration and select the application executable to use for execution. Select the appropriate executable for debug configurations, see Figure 3.31. You only need to create the configuration once for the first execution of the application. For subsequent execution of the application, you can select the configuration from the Debug drop-down list in the toolbar.
CHAPTER 3. CREATING THE SOFTWARE PLATFORM USING SDK

- Click **Debug**.

The ELF file will be downloaded to the FPGA into the bootloop placeholder space in the bitstream.

- If the **Confirm Perspective Switch** dialog box appears, click **Yes** to switch to the Debug perspective.

You can also switch to this perspective by clicking on the **Open Perspective** button on the top right bar, see Figure 3.32.

![Figure 3.32: Open Perspective button](image)

When the **Open Perspective** dialog box appears, choose **Debug** option and click **OK**, see Figure 3.33.
3.7.3 Debug Perspective

The Debug perspective lets you manage the debugging or running of a program in the Workbench. You can control the execution of your program by setting breakpoints, suspending launched programs, stepping through your code, and examining the content of variables.

The Debug perspective displays the following information:

- Each program running on the processor (represented as a node in the tree)
- The stack frame for the suspended program that you are debugging

The Debug perspective also drives the C/C++ Editor. As you step through your program, the C/C++ Editor highlights the location of the execution pointer, see Figure 3.34.

- The Debug perspective will open, showing the `modulator_sozius_no_intc.c` source file in the source view, various variables defined in the file in the Variables view, Outline view showing the objects which are in the current scope, thread created and the program suspended in the Debug view. Note that the program operation is suspended at the first executable statement (at the `main()` entry point), see Figure 3.34.
In the process of debugging the most important task is adding breakpoints to halt program execution at the user specified points. Once program execution is suspended, user can use **Step Into**, **Step Over**, **Step Return** and **Resume** commands to control program execution from the encountered breakpoint. These commands, together with variables and memory views, enable user to have total control and overview of the program execution process during debugging.

Following steps will illustrate how these commands can be used in debug process.

- First we will illustrate the usage of **Step Into**, **Step Over** and **Step Return** commands. As already noted, after debugging process is started, program operation will suspend at the start of the `main()` function. We can use **Step Into** button to enter into the Xilinx provided function for the text printing, `xil_printf`, in order to overview the function execution in more details. After you press the **Step Into** button, debugger will reach the `xil_printf` function call within the `main()` function. After pressing the **Step Into** button once more, debugger will automatically jump to the first executable statement of the `xil_printf` function as shown on the Figure 3.35.

- While the debugger is working within a function call, you can use **Step Return** button to execute all remaining statements within a function in order to quickly return to the point where a function has been called. In our case if we press **Step Return** button once, debugger will execute all remaining statements within the `xil_printf` function and return to the `main()` function, because `xil_printf` function has been called from the
main() function, and suspend program execution at the next executable statement, which in our case is yet another function call, this time to the another xil_printf function, as shown on the Figure 3.36.

Figure 3.36: Result of Step Return command execution within xil_printf function

- Although Step Into command can be very useful in the process of program debugging, quite often we are not interested into details of every function execution. If we would like to skip over known working functions, because we have debugged them previously, we can use Step Over button that will execute the complete function in one step, treating C function calls as a single C statement. Please notice that our program that we are currently debugging has suspended execution at the second xil_printf function call as shown on the previous figure. If you are not interested in the details of the execution of this function, you can execute it at once by clicking on the Step Over button. Debugger will now execute all the statements within the xil_printf function and only then suspend the program execution once more, after reaching the first executable statement located after the xil_printf function, in our case this would be another XGpioPs_LookupConfig function call, as shown on the Figure 3.37.

Figure 3.37: Result of the execution of the Step Over command on the second xil_printf function

Breakpoints

Next we will illustrate how to use breakpoints to suspend program execution at the user-selected line of program code. A breakpoint suspends the execution of a program at the location where the breakpoint is set. By default, SDK sets breakpoints at \texttt{main()} and \texttt{exit()} functions. When you start a debug session, the processor stops at the start of the \texttt{main()} function of the program. There are two types of breakpoints used by the debugger:

- \textbf{Software Breakpoint} - To set a software breakpoint, the debugger modifies the program instruction at the breakpoint address. The debugger does not require any hardware resources for setting a software breakpoint, so you can essentially set any number of software breakpoints in your debug session. The debugger requires access to read and write to the breakpoint address location. This is the default breakpoint used by the debugger.

- \textbf{Hardware Breakpoint} - To set a hardware breakpoint, the debugger does not require modification of the program instruction at the breakpoint address. Each processor provides a limited set of hardware breakpoints. In the case of MicroBlaze processors, this is configurable and set by the hardware developer and should be used wisely. You should use hardware breakpoints when the debugger cannot read or write to the program memory, such as when using Flash memory.

- We will place the first breakpoint within \texttt{init_sin_f} function, at the line 36, where for loop is located. \texttt{init_sin_f} function is defined in the \texttt{init_sin.c} source file. First we must select \texttt{init_sin.c} source file by clicking on the \texttt{init_sin.c} tab. Next, point the mouse to the line 36 in the \texttt{init_sin.c} source file and right-click
on the blue stripe located on the left border of the Sources window. A drop-down menu will appear from which **Add Breakpoint...** option should be selected, see Figure 3.38.

![Figure 3.38: Add Breakpoint option](image1)

- When you select **Add Breakpoint...** option a **Properties for C/C++ Line Breakpoint** dialog box will appear allowing you to specify the properties of the new breakpoint as shown on the Figure 3.39. Since we want to add a simple breakpoint at this moment, we don’t have to change anything, so simply click **OK**.

![Figure 3.39: Properties for C/C++ Line Breakpoint dialog box](image2)

- After you have added a new breakpoint, its location in the program code will be made visible by the blue circle marker located on the blue stripe just left of the program code line for which the breakpoint was specified, see Figure 3.40.
Figure 3.40: Breakpoint added

- Please remember that our program execution is currently suspended at line 86 in the modulator_sozins_no_intr.c source file and that there are several executable statements located between this line and line 126 where call to the init_sin_f function is located, which contains the breakpoint. These statements need to be executed before reaching the breakpoint. These executable statements can be executed by pressing the Step Over button appropriate number of times until we reach line 126. However, this would be a very inefficient way of program debugging. Instead we can use Resume button to quickly execute all executable statements between our current position and the breakpoint position.

- After we have pressed Resume button, debugger will execute all necessary statements until it reaches a breakpoint set at line 36 within the init_sin.c source file and then suspend program execution, as shown on the Figure 3.40.

Figure 3.41: Breakpoint reached

- Next we will illustrate how the Memory tab can be used to monitor the content of the array variables. Please notice that program execution is suspended at line 36. We will use the Monitor tab to overlook this initialization process. First thing that must done is to determine the base address at which the sine_ampl array is stored in the memory. To do so, look in the Variables tab for variable with sine_ampl name. Inspect the content of the Value field located in the same row. This is the starting address of the sine_ampl array.

Figure 3.42: Starting address of the sine_ampl array

- To open Memory tab, select Window -> Show View -> Memory option from the main menu.

- In the Memory tab, click on the Add Memory Monitor button. A new dialog box will appear where we should specify the address or expression to monitor. In our case we will specify the starting address of the
sine_ampl array, \(0x0010c024\), see Figure 3.43. After you do so, click OK.

![Monitor Memory dialog box](image)

Figure 3.43: Monitor Memory dialog box

- In the Memory tab, please notice that a new Memory monitor has been added, monitoring the memory content starting from the address \(0x0010c024\) as shown on the Figure 3.44. Currently the content of all memory locations starting from the address \(0x0010c024\) is \(0x00000000\). This is fine, since we still have not initialized the sine_ampl array.

![Content of the sine_ampl array in Memory window before array initialization](image)

Figure 3.44: Content of the sine_ampl array in Memory window before array initialization

- Let us initialize the first member of the sine_ampl array, with index value 0. Please press Step Over button once. After the first step over command, debugger will execute the for statement. Since this is the first time this statement is executed it will set the value of the iterator variable \(i\) to 0, as specified by the for statement. This change of the variable \(i\) value is also indicated in the Variables tab, where line holding the variable \(i\) is coloured yellow and holds the new value for the variable \(i\), see Figure 3.45.

![Change of \(i\) variable value indication in the Variables tab](image)

Figure 3.45: Change of \(i\) variable value indication in the Variables tab

- Press Step Over button twice more. This time debugger will execute the line of code that initialises sine_ampl array member with index value \(i=0\). After debugger finishes executing this line of code it will suspend program execution and update the Memory tab as shown on the Figure 3.46. If you inspect the value stored at the memory location \(0x0010c024\), you can see that it has changed from \(0x00000000\) to \(0x000007FF\) which is the correct initial value for the sine_ampl[0] array member.
CHAPTER 3. CREATING THE SOFTWARE PLATFORM USING SDK

- Finally, we will create a conditional breakpoint in order to stop the sine_ampl initialization process after a specified number of array element have been initialized. Right-click on the blue stripe just left of the line 38 and select Add Breakpoint... option once more. Properties for C/C++ Line Breakpoint dialog box will appear as before. Since now we would like to place a conditional breakpoint we must specify breakpoint condition using the Condition field. In this example we would like to break a program execution when loop iterator i reaches the value 5. This would mean that debugger should stop sine_ampl array initialization process after sine_ampl members 0-4 have been initialized. To specify this condition type i==5 in the Condition field as shown on the Figure 3.47 press OK button to complete the conditional breakpoint setup.

![Figure 3.47: Properties for C/C++ Line Breakpoint dialog box - condition breakpoint setup i==5](image)

- You can verify that a new conditional breakpoint has been placed at the line 38 which is designated by the blue circle located on the blue stripe just left to the line 38. Since this breakpoint is conditional breakpoint, next to the blue circle a question marker is also visible as shown on the Figure 3.48.

![Figure 3.48: Conditional breakpoint added](image)

- Remove the existing breakpoint at line 36 by right-clicking on the blue stripe just left of the line 36 and selecting Toggle Breakpoint option.
- Press Resume button to continue program execution. Debugger will continue initialising sine_ampl array until it reaches the condition specified in the conditional breakpoint located at line 38. This will happen when loop iterator i reaches the value of 5. After this condition is met, debugger will suspend program execution and display the current content of sine_ampl array in the memory tab as show on the Figure 3.49. Please notice that sine_ampl members with index values 0-4 have already been initialised to appropriate values, because memory locations with addresses from 0x0010c024 to 0x0010c034 have values that are different from 0. Since sine_ampl array is an array of unsigned integers, each array member occupies one double word in the memory. This means that sine_ampl members with index values 0-4 should occupied memory block starting from 0x0010c024 to 0x0010c034, which is exactly the memory block that has values different from 0 as show in the Memory tab.

![Figure 3.49: Conditional breakpoint reached, i=5](image-url)
Chapter 4

USING CUSTOM IPS IN EMBEDDED SYSTEM DESIGN

In this chapter you will learn how to integrate a custom IP within the ARM-based embedded system using Xilinx Vivado and SDK tools.

Structure of This Chapter

First we will show how to create a hardware platform that includes a custom IP, in our case it will be PWM Modulator IP core (modulator_axi_ip_v1.0) with AXI-Lite interface.

Next, we will show how to develop a basic driver for PWM Modulator IP core, how to integrate it in the Xilinx SDK tool chain, and how to develop an application that will use this driver to communicate with PWM Modulator IP core.

PWM Modulator IP core (modulator_axi_ip_v1.0) was developed and packaged in the sub-chapter 11.2 "Creating Modulator IP Core with AXI4 Interface" of the Vivado "Basic FPGA Tutorial".

Users not familiar with the details of the PWM Modulator IP core and the process of packaging it to the Vivado compliant IP core, please refer to the mentioned sub-chapter 11.2 for more details.

Block diagram of the hardware platform that we will create is shown on the Figure 4.1.
Comparing the Figure 4.1 with the Figure 1.3 it can be seen that the structure of new hardware platform is simpler. In the new hardware platform only two IP cores are needed: standard Xilinx one-channel GPIO IP core and an instance of PWM Modulator (modulator_axi_ip_v1.0) custom IP core. Since all the functionality needed to generate pwm signal is packaged inside PWM Modulator custom IP core, there is no need for timer and interrupt controller IP cores that are present on Figure 1.3.

About PWM Modulator (modulator_axi_ip_v1.0) Custom IP Core

PWM Modulator (modulator_axi_ip_v1.0) custom IP core is designed to be fully self-contained pwm generator module.

It can generate pwm output signal, modulated by the sine signal, with two different, user-defined frequencies.

It uses an AXI-Lite interface to connect to the AXI4 internal system bus of any AXI enabled microprocessor.

Operation of the PWM Modulator custom IP core is controlled through the set of three internal 32-bit configuration registers, which are accessed through the AXI-Lite interface:

- the first register, sel REGISTER, will be used to replace the sel switch from the board
- the second register, inc_freqhigh REGISTER, will be used for storing inc_freqhigh increment values
- the third register, inc_freqlow REGISTER, will be used for storing inc_freqlow increment values

Table 4.1 shows the internal 32-bit configuration registers address map.
Table 4.1: Internal Registers Address Map of the Modulator IP Core

<table>
<thead>
<tr>
<th>Internal Register Name</th>
<th>S_AXI_AWADDR Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;sel&quot; register</td>
<td>&quot;0000&quot; (0)</td>
</tr>
<tr>
<td>&quot;inc_freqhigh&quot; register</td>
<td>&quot;0100&quot; (4)</td>
</tr>
<tr>
<td>&quot;inc_freqlow&quot; register</td>
<td>&quot;1000&quot; (8)</td>
</tr>
</tbody>
</table>

This information is important for the software developer in order to correctly access, configure and control PWM Modulator IP core.

Next, we will show how to create a hardware platform, based on the ARM processor, using Vivado IP integrator tool.

4.1 Using Custom IP in ARM-based Processor System

In order to create a ARM-based embedded system that uses PWM Modulator custom IP core, following steps need to be performed.

Create New Project

- Create a new modulator_sozius_axi project using Vivado IDE wizard. To create a new project, please repeat steps 1-10 from the Sub-chapter 2.1 Create a New Project.

- In the Vivado Flow Navigator, under the Project Manager, click on the Settings command and in the Settings General window check is the Target language set to VHDL. If it is not, please change it to be VHDL and click OK.

When we have created a new project, we have to add packaged IP to the IP Catalog. The following steps will show you how to add packaged IP to the IP Catalog:

Add Packaged IP into the IP Catalog

- Create a new folder, ip_repository, in the same directory where modulator_sozius_axi project is created. This new folder will be place where we will copy the packaged modulator_axi_ip IP core.

- Copy the packaged modulator_axi_ip IP core to the ip_repository folder and extract it inside the same folder.

- Then, in the Flow Navigator, under the Project Manager, click on the Settings command.

- In the Settings dialog box, under the Project Settings commands from the left pane, expand IP and select Repository option.

Repository Manager lets you add or remove user repositories and establish precedence between repositories.
- In the Repository manager window, click + icon to add the desired repository.

- In the IP Repositories window, choose ip_repository folder with packaged modulator_axi_ip IP core and click Select.

- In the Add Repository dialog box, click OK to add the selected repository (ip_repository with 1 IP) to the project.
In the Flow Navigator, under the Project Manager, click IP Catalog command to verify the presence of the previously added IP in the IP Catalog.

In the Search field type the name of your IP core (modulator_axi_ip) and you should find it under AXI Peripheral IPs.

Now, when we have all the necessary IPs for our design, we will create block design for our project.

The following steps describe how to use the IP Integrator within your project:
- First repeat steps 1 - 15 from the Sub-chapter 2.3.1 "Create ARM-based hardware platform for Sozius development board" to create a block design with Zynq PS configured to run on Sozius development board.

After Vivado has finished with the Tcl script execution, a created block diagram containing Zynq PS will be visible in the Vivado IDE.

![Block diagram of Zynq PS configured to run on Sozius board](image)

Figure 4.6: Block diagram of Zynq PS configured to run on Sozius board

- The next step will be to add the rest of the necessary IPs into the design canvas. These IPs are located in the Vivado IP Catalog and IP Integrator gave you a possibility to add them on three ways:

  - In the design canvas, right-click and choose Add IP... option, see Figure 4.7, or

![Add IP option](image)

Figure 4.7: Add IP option

  - Use the Add IP link in the IP Integrator canvas, see Figure 4.8, or
 CHAPTER 4. USING CUSTOM IPS IN EMBEDDED SYSTEM DESIGN

- on the **Add IP** button in the IP Integrator sidebar menu, see Figure 4.9.

**Figure 4.9: Add IP button**

**Add modulator_axi_ip IP Core into the Design**

- In the design canvas, right-click and choose **Add IP...** option.

- In the **IP Catalog**, search for the **modulator_axi_ip** IP core.

**Figure 4.10: modulator_axi_ip_v1.0 IP core in the IP Catalog**

- When you find it, press enter on the keyboard or simply double-click on the **modulator_axi_ip_v1.0** core in the IP Catalog and the selected core will be automatically instantiated into the IP Integrator design canvas.
CHAPTER 4. USING CUSTOM IPS IN EMBEDDED SYSTEM DESIGN

Figure 4.11: Automatically instantiated modulator_axi_ip_v1.0 core in the IP Integrator design canvas

Re-customize modulator_axi_ip IP Core

- Double-click on the modulator_axi_ip_v1.0 (modulator_axi_ip_0) IP core to re-customize it.

- In the modulator_axi_ip_v1.0(1.0) customization window, set:
  - Lat Depth G to value 12
  - Lat Width G to value 16
  - Nco Width G to value 31

- Click OK.

Figure 4.12: Re-customize IP - modulator_axi_ip_v1.0 (1.0) dialog box
As we already said, in our design we will program PL part of the Zynq FPGA. Since existing LEDs and switches on the Sozius development board are connected to the PS part of the Zynq FPGA, we have to instantiate Integrated Logic Analyzer (ILA) and Virtual Input/Output (VIO) cores into our design. All the detailed information about ILA and VIO cores you can find in the Chapter 10 "Debugging Design" of the "Basic FPGA Tutorial" tutorial.

- The next IPs necessary for our design are Binary Counter (c_counter_binary_0), ILA (ila_0) and VIO vio_0 IPs. Add all three IPs into the "modulator_sozius_axi" block design as it is shown on the Figure 4.13 and make the following IP customizations.

**Add Binary Counter, ILA and VIO IP Core into the Design**

![Diagram](image_url)

Figure 4.13. IP Integrator design canvas with instantiated Counter, ILA and VIO IPs

- Double-click on the Binary Counter (c_counter_binary_0) IP and in the Binary Counter (12.0) Recustomization IP dialog box set the following parameters:
  - in the **Basic** tab:
    - set **Output Width** value to 32, see Figure 4.14 and
Re-customize Binary Counter \((c\_counter\_binary\_0)\) IP Core - Basic Tab

Figure 4.14: Binary Counter (12.0) re-customization IP dialog box - Basic tab

- in the Control tab:
  - enable Clock Enable (CE) and Synchronous Clear (SCLR) options, see Figure 4.15 and click OK.
Re-customize Binary Counter (c_counter_binary_0) IP Core - Control Tab

![Binary Counter (12.0) re-customization IP dialog box - Control tab](image)

**Figure 4.15: Binary Counter (12.0) re-customization IP dialog box - Control tab**

Add Utility Vector Logic (util_vector_logic_0) IP Core into the Design

- Because of the structure of the binary counter that we need, we also had to include one inverter into our IP Integrator design.

- Add **Utility Vector Logic** (util_vector_logic_0) IP into design canvas, double-click on it and re-customize it.

In the **Utility Vector Logic (2.0)** dialog box, make the following changes:

- change the **C_OPERATION** to **not** and

- set the **C_SIZE** to be **1**, see Figure 4.16, and

- click **OK**.
Re-customize Utility Vector Logic \((\text{util\_vector\_logic\_0})\) IP Core

- Double-click on the ILA IP and in the ILA (Integrated Logic Analyzer (6.2)) dialog box, in the General Options, set the following parameters:
  - select Native as Monitor Type
  - set 2 as Number of Probes, and
  - enable Capture Control option in the Trigger And Storage Settings section, as it is shown on the Figure 4.17.
Re-customize ILA (ila_0) IP Core - General Options Tab

Figure 4.17: ILA (Integrated Logic Analyzer (6.2)) Re-customize IP dialog box - General Options

and in the **Probe Ports**(0..7), set the following parameters:

- set **32 bits as Probe Width[1..4096]** value of PROBE0 probe, as it is shown on the Figure 4.18, and
- click **OK**.
Re-customize ILA (ila_0) IP Core - Probe Ports(0..7) Tab

- In case of VIO core, double-click on the VIO IP and in the VIO (Virtual Input/Output (3.0)) dialog box, in the General Options tab, set the Output Probe Count to be 0, see Figure 4.19. In case of VIO core we will need only one input probe, to connect it with the modulator_axi_ip_0 pwm_0 port.
Re-customize VIO (vio_0) IP Core - General Options Tab

- In the PROBE_IN Ports(0..0) tab, leave PROBE_IN0 port to be 1-bit width, because pwm_0 port is also 1-bit wide and click OK, see Figure 4.20.

Re-customize VIO (vio_0) IP Core - PROBE_IN Ports(0..0) Tab

Figure 4.19: VIO (Virtual Input/Output (3.0)) Re-customize IP dialog box - General Options

Figure 4.20: VIO (Virtual Input/Output (3.0)) Re-customize IP dialog box - PROBE_IN Ports(0..0)
Add AXI Protocol Converter (axi_protocol_converter_0) IP Core into the Design

The last IP that we need in our design is AXI Protocol Converter.

- Add the AXI Protocol Converter (axi_protocol_converter_0) IP core into the block design.

- In the AXI Protocol Converter (2.1) dialog box, make the following changes:
  - change SI PROTOCOL mode from Auto to Manual and
  - select AXI3 from the drop-down list of SI PROTOCOLs and
  - click OK, see Figure 4.21.

Re-custimize AXI Protocol Converter (axi_protocol_converter_0) IP Core

![Re-customize IP dialog box](image)

Figure 4.21: AXI Protocol Converter (2.1) Re-customize IP dialog box

At this stage of the design process, the IP Integrator design canvas should look like as it is shown on the Figure 4.22.
IP Integrator Design Canvas With All Necessary IPs

- In the block design double-click on the **ZYNQ7 Processing System** IP block.

- In the **ZYNQ7 Processing System** (5.5) re-customize IP dialog box, select **PS-PL Configuration** option from the left menu. In the **PS-PL Configuration** section, expand **AXI Non Secure Enablement**, then expand **GP Master AXI Interface** and enable **MAXI GP0 interface** option, see Figure 4.23, and click OK.
Re-customize ZYNQ7 Processing System IP Core

![Re-customize ZYNQ7 Processing System IP Core](image)

Figure 4.23: ZYNQ7 Processing System (5.5) Re-customize IP dialog box

We need this extra GPIO port to connect it with AXI Protocol Converter that will be connected with the modulator_axi_ip IP core.

After we re-customized ZYNQ7 Processing System, the new M_AXI_GP0 port will appear in the ZYNQ IP block, see Figure 4.24.

After we added all the necessary IPs into our design and after all the necessary IP customizations, the IP Integrator design canvas should look as it is shown on the Figure 4.24.
IP Integrator Design Canvas With All Necessary and Re-customized IPs

![IP Integrator Design Canvas](image)

Figure 4.24: IP Integrator design canvas with all necessary IPs and re-customizations

- Next step will be to manually connect the IPs:

Place the cursor on top of the desired pin and you can notice that the cursor changes into a pencil indicating that a connection can be made from that pin. Clicking the left mouse button a connection starts. Click and drag the cursor from one pin to another. You must press and hold down the left mouse button while dragging the connection from one pin to another. As you drag the connection wire, a green checkmark appears on the port indicating that a valid connection can be made between these points. The Vivado IP Integrator highlights all possible connections points in the subsystem design as you interactively wire the pins and ports. Release the left mouse button and Vivado IP integrator makes connection between desired ports. Repeat this procedure until all the pins become associated.

**Note**: Connect all the IPs on the same way as it is shown on the Figure 4.25. As you can see we connected all the IPs, except clock ports and reset ports.
Manually Connect the IPs

![Figure 4.25: IP Integrator design canvas with manually connected IPs](image)

Run Connection Automation

- In the IP Integrator window, click the **Run Connection Automation** link and the list of the ports/interfaces that can use the Connection Automation feature will show up.

**Run Connection Automation** link is IP Integrator feature that assist you in putting together a basic microprocessor system, making internal connections between different blocks and making connections to external interfaces.

- In the **Run Connection Automation** dialog box enable **All Automation (6 out of 6 selected)** and click OK.

![Figure 4.26: Run Connection Automation dialog box](image)

After Running Connection Automation

After running the connection automation, the connections will be made and highlighted in the IP Integrator design canvas.
CHAPTER 4. USING CUSTOM IPS IN EMBEDDED SYSTEM DESIGN

Create Address Map for Your Embedded System

- Click on the Address Editor tab, beside Diagram tab, to open the Address Editor window.

- In the Address Editor window, expand sozius_xz_lab_ps, then expand Data, after that expand Unmapped Slaves and right-click on the S00_AXI and choose Assign Address option to assign some memory space for S00_AXI of the modulator_axi_ip_0 IP core.

Validate Design

- From the toolbar menu of the design canvas, run the IP subsystem design rule checks by clicking the Validate Design button.

Alternatively, you can do the same by selecting Tools -> Validate Design from the main menu.
CHAPTER 4. USING CUSTOM IPS IN EMBEDDED SYSTEM DESIGN

Figure 4.30: Validate Design option from the main menu

- In the **Validate Design** dialog box, click **OK**, see Figure 4.31.

![Validate Design dialog box](image)

Figure 4.31: Validate Design dialog box

**Save Block Design**

- At this point, you should save the IP integrator design.

Use the **File -> Save Block Design** command from the main menu to save the design.

**Synthesize and Implement Design, Generate Bitstream File and Program Target Device**

- Synthesize your design using **Run Synthesis** command from the **Flow Navigator / Synthesis** section.

- Implement your design using **Run Implementation** command from the **Flow Navigator / Implementation** section.

- Generate bitstream file using **Generate Bitstream** command from the **Flow Navigator / Program and Debug** section.

To complete the design, we must now create a software component for our embedded system. This application specific software will be executed on the ARM processor that is already part of our hardware platform.

**Create Software Component for Your Embedded System**

- The first step in software creation is to export the hardware design into the SDK tool. To export your hardware platform into the SDK tool and to launch SDK, please repeat steps 1 - 4 from the **Chapter 3 "Creating the Software Platform using SDK"**.

- Create **Board Support Package**. To create BSP, please repeat steps from the **Sub-chapter 3.1 "Board Support Package"**.

- To create **modulator_sozius_axi** application project, repeat steps from the **Sub-chapter 3.2 "Creating an Application Project"**.
- In the `modulator_sozius_axi` application project, create `modulator_sozius_axi.c` and `modulator_sozius_axi.h` source files on the same way as it is explained in the Sub-chapter 3.3 "Creating a C/C++ Source Files for Sozius Board Based Hardware Platform".

The complete `modulator_sozius_axi.c` and `modulator_sozius_axi.h` source files you can find in the text below.

```
#include "xparameters.h"
#include "xgpiops.h"
#include "xstatus.h"
#include "math.h"
#include "modulator_sozius_axi.h"

int main(void)
{
    /*********************** Variable Definitions ***********************/
    XGpioPs GpioSwitches; // XGPIO instance that will be used to work with SWITCHes
    XGpioPs_Config *GPIOConfigPtr;
    int *modulator_axi4_lite_interface; // pointer to modulator_axi_ip memory-mapped internal registers
    int sel = 0; // switch used for selecting frequency
    int current_sel = 0; // current value of the sel bit in the modulator_axi_ip IP core
    int temp;

    /*********************** Initialization ***********************/
    // SWITCHes initialization
    GPIOConfigPtr = XGpioPs_LookupConfig(XPAR_PS7_GPIO_0_DEVICE_ID);
    XGpioPs_CfgInitialize(&GpioSwitches, GPIOConfigPtr, GPIOConfigPtr -> BaseAddr);
    XGpioPs_SetDirectionPin(&GpioSwitches, SWITCH_CHANNEL, 0);

    // modulator_axi_ip IP core initialization
    modulator_axi4_lite_interface = SEL_REGISTER_ADDRESS;
    *modulator_axi4_lite_interface = sel;

    modulator_axi4_lite_interface = INC_FREQHIGH_REGISTER_ADDRESS;
    *modulator_axi4_lite_interface = INC_FREQHIGH;

    modulator_axi4_lite_interface = INC_FREQLOW_REGISTER_ADDRESS;
    *modulator_axi4_lite_interface = INC_FREQLOW;

    // readback
    modulator_axi4_lite_interface = SEL_REGISTER_ADDRESS;
    temp = *modulator_axi4_lite_interface;

    modulator_axi4_lite_interface = INC_FREQHIGH_REGISTER_ADDRESS;
    temp = *modulator_axi4_lite_interface;

    modulator_axi4_lite_interface = INC_FREQLOW_REGISTER_ADDRESS;
    temp = *modulator_axi4_lite_interface;

    /***************** Main Loop *******************/
    while(1)
    {
        // read the switch position
        sel = XGpioPs_ReadPin(&GpioSwitches, SWITCH_CHANNEL);

        sel = sel & SWITCH_POS; // masking (we want to check the status of sel only)

        if (sel != current_sel)
        {
            current_sel = sel;
            modulator_axi4_lite_interface = SEL_REGISTER_ADDRESS;
            *modulator_axi4_lite_interface = sel;
        }

        return 0;
    }
```

www.so-logic.net 2019/01/28 107
modulator_sozius_axi.h

#ifndef MODULATOR_H_
#define MODULATOR_H_
#include "math.h"
#include "xparameters.h"

/********************** Constant Definitions **********************
#define SWITCH_CHANNEL 12
// address map for modulator_axi_ip internal registers
#define SEL_REGISTER_ADDRESS XPAR_MODULATOR_AXI_IP_0_S00_AXI_BASEADDR
// address of inc_freqhigh register
#define INC_FREQHIGH_REGISTER_ADDRESS XPAR_MODULATOR_AXI_IP_0_S00_AXI_BASEADDR + 4
// address of inc_freqlow register
#define INC_FREQLOW_REGISTER_ADDRESS XPAR_MODULATOR_AXI_IP_0_S00_AXI_BASEADDR + 8
#define SYS_CLK_MHZ 50.0 // 50 MHz system clock
#define CLOCK_RATE (1000000.0 * SYS_CLK_MHZ) // system clock value, expressed in Hz
// CLOCK_RATE = 50000000 Hz (= 50 MHz)
#define F_LOW 1.0 // F_LOW = 1 Hz
#define F_HIGH 3.5 // F_HIGH = 3.5 Hz
#define NCO_WIDTH 31 // number of bits used for numerically controlled oscillator
#define C pow(2, NCO_WIDTH) // 2^NCO_WIDTH
#define SWITCH_POS 0x01 // mask to select switch position
#endif /* MODULATOR_H_ */

Download Bitstream File into the Sozius Development Board

- Download your new ARM-based bitstream file to the target board, please repeat steps from the sub-chapter 3.6.1 "Downloading ARM-based Bitstream File".

- Turn back to the Vivado IDE and Flow Navigator, under the Program and Debug, click Open Hardware Manager command.

![Open Hardware Manager](image)

Figure 4.32: Open Hardware Manager command

The another way to open the hardware manager is to select Flow -> Open Hardware Manager option, from the main Vivado menu.

- The next step in opening a hardware target is connecting to the hardware server that is managing the connection to the hardware target. You can do this on three ways:

  - Use the Open target selection in the Hardware Manager view, to open a recent or a new hardware targets, see Figure 4.33.
CHAPTER 4. USING CUSTOM IPS IN EMBEDDED SYSTEM DESIGN

Figure 4.33: Hardware Manager view

- Use the Open Target command, under the Open Hardware Manager in the Program and Device section, to open new or recent hardware targets, see Figure 4.34.

Figure 4.34: Open Target command

- Use Tcl commands to open a connection to a hardware target.

- Click the Open New Target command.

The Open New Hardware Target wizard provides an interactive way for you to connect to a hardware server and target.

Figure 4.35: Open Hardware Target dialog box

- In the Open Hardware Target dialog box, click Next.

- In the Hardware Server Settings dialog box, specify or select a local or remote server, depending on what machine your hardware target is connected to. Leave the default Local server and click Next.
CHAPTER 4. USING CUSTOM IPS IN EMBEDDED SYSTEM DESIGN

Figure 4.36: Hardware Server Settings dialog box

- **Local server**: Use this setting if your hardware target is connected to the same machine on which you are running the Vivado IDE. The Vivado software automatically starts the Vivado hardware server (hw_server) application on the local machine.

- **Remote server**: Use this setting if your hardware target is connected to a different machine on which you are running the Vivado IDE. Specify the host name or IP address of the remote machine and the port number for the hardware server (hw_server) application that is running on that machine.

- In the **Select Hardware Target** dialog box, select the appropriate hardware target from the list of targets that are managed by the hardware server. Note that when you select a target, you will see the various hardware devices that are available on the hardware target. Click **Next**.

Figure 4.37: Select Hardware Target dialog box

- In the **Open Hardware Target Summary** dialog box, click **Finish** to connect to the hardware described in the summary window.
Ones you finish opening a connection to a hardware target, the **Hardware** window is populated with the hardware server, hardware target, and various hardware devices for the open target, see Figure 4.39.

![Hardware window](image)

**Figure 4.39: Hardware window**

**Automatically Detected ILA and VIO Dashboards**

When the debug cores are detected upon refreshing a hardware device, the default dashboard for each debug core is automatically opened.
4.2 Debug a Design using Integrated Vivado Logic Analyzer

**Vivado Logic Analyzer**

*Vivado Logic Analyzer* is an integrated logic analyzer in the Vivado Design Suite.

In this chapter you will learn how to debug your ARM-based system using the Vivado logic analyzer and you will take advantage of it’s functions to debug and discover some potential root causes of your design.

In-system debugging allows you to debug your design in real-time on your target hardware.

IP Integrator provides ways to instrument your design for debugging, which will be explained in this chapter.

After programming the FPGA device with the .bit file that contains the ILA and VIO cores, the Hardware window now shows the ILA and VIO cores that were detected after scanning the device, see Figure 4.40.

Once you have the debug cores in your design, you can use the run time logic analyzer features to debug the design in hardware. The Vivado logic analyzer feature is used to interact with new ILA, VIO, and JTAG-to-AXI Master debug cores that are in your design.

The next step in our design process is to set up the ILA core.

**Automatically Detected ILA and VIO Dashboards**

When the debug cores are detected upon refreshing a hardware device, the default dashboard for each debug core is automatically opened.

The default ILA Dashboard can be seen on the following figure.
Every default dashboard contains windows relevant to the debug core the dashboard is created for. The default dashboard created for the ILA debug core contains five windows, as can be seen on the previous illustration:

- **Settings** window
- **Status** window
- **Trigger Setup** window
- **Capture Setup** window
- **Waveform** window

**Add Probes to the VIO Dashboard**

- Open the VIO dashboard by clicking the `hw_vios` tab and press blue + button in the middle of the VIO dashboard to add the probes.
- In the **Add Probes** window select the only offered `pwm_o` probe and click OK.
- In the VIO Probes window you will see one 1-bit probe, pwm_o.

pwm_o probe is actually connected to the pwm_o output port of the Modulator AXI module. In the VIO Probes window, you can observe the rate of change of the pwm_o signal.

Add Probes to the Trigger Setup Window

- Turn back to the ILA dashboard by clicking the h_ila_1 tab and in the Trigger Setup window press blue + button in the middle to add the probes.

- In the Add Probes window select only pwm_o_1 probe and click OK.
CHAPTER 4. USING CUSTOM IPS IN EMBEDDED SYSTEM DESIGN

The another way to add debug probes to the Trigger Setup window is to drag and drop the probes from the Debug Probes window to the Trigger Setup window.

**Important:** Only probes that are in the **Trigger Setup** or **Capture Setup** window participate in the trigger condition. Any probes that are not in the window are set to "don’t care" values and are not used as part of the trigger condition.

The **Debug Probes** window contains information about the nets that you probed in your design using the ILA and/or VIO cores. This debug probe information is extracted from your design and stored in a data file that typically has an .ltx file extension. Normally, the ILA probe file is automatically created during implementation process. This file is automatically associated with the FPGA hardware device if the probes file is called debug nets.ltx and is found in the same directory as the bitstream file that is associated with the device.

**Change the Compare Values in the Trigger Setup Window**

Now, when the ILA debug probe **pwm_o_1** is in the **Trigger Setup** window, we can create trigger conditions and debug probe compare values.

- In the **Trigger Setup** window, leave == (equal) value in the **Operator** cell, [H] (Hexadecimal) value in the **Radix** cell and set the **Value** parameter to be 0 (logical zero).

As you can see from the illustration above, the **Trigger Setup** window contains three fields that you can configure:

- **Operator**: This is the comparison operator that you can set to the following values:
  - == (equal)
CHAPTER 4. USING CUSTOM IPS IN EMBEDDED SYSTEM DESIGN

- \(! = (not \ equal)\)
- < (less than)
- \(<= (less \ then \ or \ equal)\)
- > (greater than)
- \(>= (greater \ than \ or \ equal)\)

- **Radix**: This is the radix or base of the Value that you can set to the following values:
  - [B] Binary
  - [H] Hexadecimal
  - [O] Octal
  - [A] ASCII
  - [U] Unsigned Decimal
  - [S] Signed Decimal

- **Value**: This is the comparison value that will be compared (using the Operator) with the real-time on the nets(s) in the design that are connected to the probe input of the ILA debug core. Depending on the radix settings, the Value string is as follows:

  - **Binary**
    - 0 : logical zero
    - 1 : logical one
    - X : don't care
    - R : rising or low-to-high transition
    - F : falling or high-to-low transition
    - B : either low-to-high or high-to-low transitions
    - N : no transition (current sample value is the same as the previous value)

  - **Hexadecimal**
    - X : All bits corresponding to the value string character are "don't care" values
    - 0-9 : Values 0 through 9
    - A-F : Values 10 through 15

  - **Octal**
    - X : All bits corresponding to the value string character are "don't care" values
    - 0-7 : Values 0 through 7

  - **ASCII**
    - Any string made up of ASCII characters

  - **Unsigned Decimal**
    - Any non-negative integer value

  - **Signed Decimal**
    - Any integer value

You can use the ILA Dashboard to interact with the ILA core in several ways:

- Use BASIC and ADVANCED trigger modes to trigger on various events in hardware
- Use ALWAYS and BASIC capture modes to control filtering of the data to be captured
- Set the data depth of the ILA capture window
- Set the trigger position to any sample within the capture window
- Monitor the trigger and capture status of the ILA debug core

**Capture mode** - selects what condition is evaluated before each sample is captured:

- ALWAYS: store a data sample during a given clock cycle regardless of any capture conditions
CHAPTER 4. USING CUSTOM IPS IN EMBEDDED SYSTEM DESIGN

- **BASIC**: store a data sample during a given clock cycle only if the capture condition evaluates "true"

**Data Depth** - sets the data depth of the ILA core captured data buffer. You can set the data depth to any power of two from 1 to the maximum data depth.

**Trigger Position** - sets the position of the trigger mark in the captured data buffer. You can set the trigger position to any sample number in the captured data buffer. For instance, in the case of a captured data buffer that is 1024 sample deep:

- sample number 0 corresponds to the first (left-most) sample in the captured data buffer
- sample number 1023 corresponds to the last (right-most) sample in the captured data buffer
- sample numbers 511 and 512 correspond to the two "center" samples in the captured data buffer

**Add Probes to the Capture Setup Window**

- In the **ILA Settings** window, change the **Capture mode** to be **BASIC** in the **Capture Mode Settings** section.
- In the **Capture Setup** window press blue + button in the middle to add the probes.
- In the **Add Probes** window select only **pwm_o_1** probe and click **OK**.

![Figure 4.46: Changing the Compare Values in the Trigger Setup window](image)

**Change the Compare Values in the Capture Setup Window**

- In the **Capture Setup** window, leave **==** (equal) value in the **Operator cell**, [B] (Binary) value in the **Radix cell** and set the **Value** parameter to be F (1-to-0 transition).
CHAPTER 4. USING CUSTOM IPS IN EMBEDDED SYSTEM DESIGN

Figure 4.47: Changing the Compare Values in the Capture Setup window

**Run ILA Core Trigger**

- After we set all the ILA core parameters, we can run or arming the ILA core trigger.

We can run or arm the ILA core trigger in two different modes:

- **Run Trigger mode** - arms the ILA core to detect the trigger event that is defined by the ILA core trigger condition and probe compare values.
  
  To run this mode, click the **Run Trigger** button in the Hardware or Debug Probes window.

- **Run Trigger Immediate mode** - arms the ILA core to trigger immediately regardless of the settings of the ILA core trigger condition and probe compare values. This command is useful for capturing any values present at the probe inputs of the ILA core.
  
  To run this mode, click the **Run Trigger Immediate** button in the Hardware or Debug Probes window.

You can also arm the trigger by selecting and right-clicking on the ILA core (**hw_ila_1**) in the Hardware window and selecting **Run Trigger** or **Run Trigger Immediate** option from the popup menu, see Figure 4.47.

**Captured Waveform - sel = 0**

Once the ILA core captured data has been uploaded to the Vivado IDE, it is displayed in the **Waveform Viewer**.

After triggering the ILA core, in the waveform viewer change the **e_counter_binary_0.Q[31:0]** Waveform Style from **Digital** to **Analog**, and your captured waveform should look like as the waveform on the following figure.
Captured Waveform - $sel = 1$

- To change the switch position from 0 to 1, press and hold the push button 4 on the Sozius development board and while holding the push button 4 run trigger for the ILA core. Do not release the button until the trigger is complete.

- Now when you have changed the switch position, run the trigger for the ILA core one more. After triggering the ILA core your captured waveform should look like as the waveform on the following figure.

By comparing the waveforms shown on Figures 4.48 and 4.49 we can observe that they differ in the amplitude value. This is expected since the waveforms actually represent the width of the PWM pulse generated by the modulator module. Since the frequencies of two generated PWM signals differ (one has a frequency of 1 Hz and the other of 3.5 Hz) and the PWM pulse width measurement module always uses the same frequency for measuring the duration of the PWM pulse, when the PWM frequency increases the duration of the PWM pulse will decrease, therefore decreasing the amplitude of the output signal of the PWM pulse width measurement module.

The ILA core can capture data samples when the core status is Pre-Trigger, Waiting for Trigger or Port-Trigger. As we already said, Capture mode selects what condition is evaluated before each sample is captured. Capture mode stores a data sample during a given clock cycle only if the capture condition evaluates "true". We used $pwm_0$ signal to do the signal capturing.

Capture condition is a Boolean combination of events that is detected by match unit comparators that are attached to the trigger ports of the core. Only when this combination is detected, data will be stored in the ILA’s buffer.

To be able to capture at least one period of the sine signal and to store it in the ILA buffer, we have to use capture condition feature. After triggering the ILA core, in the waveform viewer change the Waveform Style...
from Digital to Analog and your captured waveform should look like as the waveform on the Figure 4.48 or Figure 4.49.

4.3 Developing a Device Driver for the Custom IP Core

Device Driver

In all previous software applications communication with the `modulator_axi_ip` IP core was done by directly accessing its internal registers.

This requires intimate knowledge of `modulator_axi_ip` IP core's internal organization, as well as exact address values for every internal register.

This approach significantly complicates software application development by a typical software developer, who is not aware and even doesn't want to be aware of all these details about the hardware device he is using.

A better approach includes an additional software layer that provides a software interface to the hardware device, enabling user application to access hardware functions without needing to know precise details of the hardware being used.

This additional software layer is known as a **device driver**.

Device Driver Working Principles

A device driver typically communicates with the hardware device through the system bus or some other communication subsystem to which the hardware device is connected.

When a user application program invokes a routine in the driver, the driver issues low level commands to the hardware device.

Once the hardware device sends data back to the device driver, the driver may invoke routines in the original calling program.

Device drivers are hardware dependent and operating system specific.

Device drivers simplify programming by acting as translator between a hardware device and the applications or operating system that use it.

Programmers can write the higher level application code independently of what ever specific hardware the end-user is using.

4.3.1 Device driver for PWM Modulator IP core

Device Driver for PWM Modulator IP Core

In this sub-chapter it will be shown how to develop a device driver for the `modulator_axi_ip` IP core and how to integrate it and use it within the Xilinx SDK tool.

Writing your drivers with a hierarchical structure is considered to be good design practice, and will save you a lot of time when you need to debug the design.

Once you have verified that you have some measurable results by writing to one register, you can then move on to testing some of the others.

It is advisable to create a function that can be re-used in your code, because this will form the basis of your driver hierarchy.
"xil_io" Driver Function

The great way to start writing your own driver is to use the supplied Xilinx IO driver functions.

The xil_io driver provides some useful IO functions which can read and write data values to registers in the hardware IP core, and these are perfect when the user wishes to design a driver for custom IP.

xil_io is easy to use and can be included in any project using a simple #include statement in the C code.

The most commonly used functions calls are Xil_Out32() and Xil_In32(), but similar functions exist for 16 bit and 8 bit data transfers.

For example, following two functions can be used to write and read a value from the selected internal register of the IP core. modulator_axi_ip_Set_Register function writes a user specified value (supplied through the value input argument) to the selected custom IP internal register (selected by specifying base address value, via baseaddr input argument and offset value, via offset input argument) using Xil_Out32 function. Similarly, modulator_axi_ip_Get_Register function reads the current value from the selected custom IP internal register and returns it to the caller, using Xil_In32 function.

```c
void modulator_axi_ip_Set_Register(int baseaddr, int offset, int value)
{
    Xil_Out32(baseaddr + offset, value);
}

int modulator_axi_ip_Get_Register(int baseaddr, int offset)
{
    int temp = 0;
    temp = Xil_In32(baseaddr + offset);
    return (temp);
}
```

In these functions, we have implemented the use of offsets from the base address, rather than using hard-coded addresses each time. Another good coding practice would be to use #define statements to specify names for commonly used hex values, enabling the user to quickly identify which register is being addressed without having to constantly cross-reference everything to the address map of the IP.

Additional Driver Functions

Presented two functions enable us to communicate with the custom IP core’s internal registers to exercise the various features of the custom IP core.

In the case of the modulator_axi_ip IP core, following additional driver functions can be defined:

- **modulator_axi_ip_Set_PWM_Frequency** function - used to set the desired frequency of the PWM signal that will be generated
- **modulator_axi_ip_Get_PWM_Frequency** function - used to read the current frequency settings used in the PWM signal generation
- **modulator_axi_ip_Select_PWM_Frequency** function - used to select one of two possible frequencies for the PWM signal generation
- **modulator_axi_ip_Get_Selected_PWM_Frequency** function - used to read the current selection of the PWM signal frequency

Source code of the modulator_axi_ip_Set_PWM_Frequency driver function is shown below.

```c
void modulator_axi_ip_Set_PWM_Frequency (Modulator_AXI * InstancePtr, int freq_sel, float freq_val)
{
    // c = 2^NCO_WIDTH
    float c = 1.0;
    for (int i = 1; i <= InstancePtr->NCO_Width; i++)
    {
        c = c * 2;
    }
    // inc_factor = C * F_LOW / CLOCK_RATE
    int inc_factor = (int)((c * freq_val/(float)InstancePtr->ClockRate) + 0.5);
    modulator_axi_ip_Set_Register(InstancePtr->BaseAddress, 4+4*freq_sel, inc_factor);
}
```
When called, this function sets the selected PWM frequency value to the user-specified value in the selected instance of the `modulator_axi_ip` IP core. Instance of the `modulator_axi_ip` IP core for which one of the PWM frequency values should be changed is specified through the `Modulator_AXI *InstancePtr` input argument. Which one of the two PWM frequency values should be changed is specified through the `int freq_sel` input argument. Function calculates the necessary value of the increment factor (variable `inc_factor`) based on the desired PWM frequency value and specified characteristics of the selected instance of the `modulator_axi_ip` IP core (`InstancePtr->NCO_Width` and `InstancePtr->ClockRate`). Calculated increment factor is then written to the appropriate internal register of the selected instance of the `modulator_axi_ip` IP core, using `modulator_axi_ip_Set_Register` function.

Source code of the `modulator_axi_ip_Get_PWM_Frequency` driver function is shown below:

```c
float modulator_axi_ip_Get_PWM_Frequency (Modulator_AXI * InstancePtr, int freq_sel)
{
    // c = 2^NCO_WIDTH
    float c = 1.0;
    for (int i = 1; i <= InstancePtr->NCO_Width; i++)
        c *= 2;
    float inc_factor = (float) modulator_axi_ip_Get_Register(InstancePtr->BaseAddress, 4+4*freq_sel);

    // freq = inc_factor * CLOCK_RATE / C
    float temp = (inc_factor * (float)InstancePtr->ClockRate / c);
    return temp;
}
```

When called, this function returns the current value of selected PWM frequency from the selected instance of the `modulator_axi_ip` IP core. Instance of the `modulator_axi_ip` IP core from which one of the PWM frequency values will be read is specified through the `Modulator_AXI *InstancePtr` input argument. Which one of the two PWM frequency values should be read is specified through the `int freq_sel` input argument. Function first reads the current value of the increment factor from the selected instance of the `modulator_axi_ip` IP core, using `modulator_axi_ip_Get_Register` function. This increment factor value is then converted to frequency value and returned to the calling function.

Source code of the `modulator_axi_ip_Select_PWM_Frequency` driver function is shown below:

```c
void modulator_axi_ip_Select_PWM_Frequency (Modulator_AXI * InstancePtr, int freq_sel)
{
    modulator_axi_ip_Set_Register(InstancePtr->BaseAddress, 0, freq_sel);
}
```

When called, this function sets the value of the PWM frequency selector in the selected instance of the `modulator_axi_ip` IP core to the user-specified value. Instance of the `modulator_axi_ip` IP core for which the PWM frequency selector value will be set is specified through the `Modulator_AXI InstancePtr` input argument. Which one of the two PWM frequency values should be used is specified through the `int freq_sel` input argument. Function simply writes the `freq_sel` input argument’s value to the appropriate internal register of the selected instance of the `modulator_axi_ip` IP core by calling `modulator_axi_ip_Set_Register` function.

Source code of the `modulator_axi_ip_Get_Selected_PWM_Frequency` driver function is shown below:

```c
int modulator_axi_ip_Get_Selected_PWM_Frequency (Modulator_AXI * InstancePtr)
{
    return modulator_axi_ip_Get_Register(InstancePtr->BaseAddress, 0);
}
```

When called, this function simply returns the current PWM frequency selector value from the selected instance of the `modulator_axi_ip` IP core. Instance of the `modulator_axi_ip` IP core from which one of the PWM frequency values will be read is specified through the `Modulator_AXI InstancePtr` input argument. Current PWM frequency selector value is read from the selected instance of the `modulator_axi_ip` IP core, using `modulator_axi_ip_Get_Register` function.
**modulator_axi_ip_Initilize**" Driver Function

To facilitate the possibility of having more than one instance of the `modulator_axi_ip` IP core in the system, driver functions presented above were written using a parametrized approach, via `Modulator_AXI *InstancePtr` input argument.

By modifying this input argument value, calling function can communicate with different instances of the `modulator_axi_ip` IP core that may be present in the system.

However, this approach requires that the user initialize every instance of the `modulator_axi_ip` IP core that is present in the system, and keep track of this via a separate `Modulator_AXI` object.

For this purpose `modulator_axi_ip_Initilize` driver function can be used.

```c
int modulator_axi_ip_Initialize(Modulator_AXI *InstancePtr, u16 DeviceId)
{
    modulator_axi_ip_Config *ConfigPtr = NULL;
    int Index;
    // Assert arguments
    Xil_AssertNonvoid(InstancePtr != NULL);
    /* Lookup configuration data in the device configuration table.
    * Use this configuration info down below when initializing this
    * driver. */
    for (Index = 0; Index < XPAR_MODULATOR_AXI_NUM_INSTANCES; Index++)
    {
        if (modulator_axi_ip_ConfigTable[Index].DeviceId == DeviceId)
        {
            ConfigPtr = &modulator_axi_ip_ConfigTable[Index];
            break;
        }
    }
    if (ConfigPtr == (modulator_axi_ip_Config *) NULL)
    {
        InstancePtr->IsReady = 0;
        return (XST_DEVICE_NOT_FOUND);
    }
    // Initialize the driver
    InstancePtr->BaseAddress = ConfigPtr->BaseAddress;
    InstancePtr->LUT_Width = ConfigPtr->LUT_Width;
    InstancePtr->LUT_Depth = ConfigPtr->LUT_Depth;
    InstancePtr->NCO_Width = ConfigPtr->NCO_Width;
    // Indicate the instance is now ready to use, initialized without error
    InstancePtr->IsReady = XIL_COMPONENT_IS_READY;
    return (XST_SUCCESS);
}
```

`modulator_axi_ip_Initilize` function initializes the selected `Modulator_AXI` object (specified by the `Modulator_AXI *InstancePtr` input argument) with the configuration information related to the `modulator_axi_ip` IP core whose device ID value is specified via the `u16 DeviceId` input argument. First, the function searches for the configuration information of the selected `modulator_axi_ip` IP core by searching through a `modulator_axi_ip_ConfigTable` array. `modulator_axi_ip_ConfigTable` array holds the configuration information of every `modulator_axi_ip` IP core instance that is present in the system. Please notice that this configuration information array is created automatically by the Xilinx SDK tool, during the building of the Board Support Package, using the instructions in the specific user-defined TCL file, that will be discussed later in this chapter. If a device ID match is found, configuration of the selected `modulator_axi_ip` IP core is copied to the ConfigPtr object. Using this configuration information, supplied `Modulator_AXI` is initialized and returned to the calling function, along with the status information about the completed initialization process.

Finally, to complete our device driver for the `modulator_axi_ip` IP core, one more driver function is needed, `modulator_axi_ip_Set_Input_Clock_Frequency`. This driver function is called for each instance of the `modulator_axi_ip` IP core during the initialization process, to correctly set the system clock frequency value that is used in every instance of the `modulator_axi_ip` IP core. This information is needed to correctly calculate increment factors in the `modulator_axi_ip_Set_PWM_Frequency` and `modulator_axi_ip_Get_PWM_Frequency` driver functions.
void modulator_axi_ip_Set_Input_Clock_Frequency (Modulator_AXI *InstancePtr, int clock_rate)
{
    InstancePtr->ClockRate = clock_rate;
}

When called, `modulator_axi_ip_Set_Input_Clock_Frequency` driver function simply updates the `ClockRate` field of the specified `Modulator_AXI` object with the user-specified value (specified via the `int clock_rate` input argument).

Create `modulator_axi_ip.c` and `modulator_axi_ip.h` Source Files

The driver functions shown above should illustrate that drivers can quickly be built to exercise the various features of the custom IP.

It should also be very obvious that we are starting to generate significant amounts of code, and we already have many functions in our source file which are beginning to be untidy.

To continue the development of the driver functions, we can now move to code into a separate `modulator_axi_ip.c` source file, and the custom prototypes into a `modulator_axi_ip.h` header file.

To maintain visibility of the functions across the different files, be sure to add the line `#include "modulator_axi_ip.h"` at the top of the main application source file, and also to the `modulator_axi_ip.c` source file.

With this file structure in place, it is possible to quickly and easily continue the development of the custom IP drivers, while keeping the source code in the main test application tidy and manageable. For very complex driver functions, additional source files can be added to the file set to facilitate hierarchy in the source code.

Below, the complete device driver source code for `modulator_axi_ip` IP core, organized into two separate files, `modulator_axi_ip.c` and `modulator_axi_ip.h`, is presented.

`modulator_axi_ip.c`:

```c
#include "xil_io.h"
#include "xstatus.h"
#include "xparameters.h"
#include "modulator_axi_ip.h"
extern modulator_axi_ip_Config modulator_axi_ip_ConfigTable[];

/****************************************************************************/

/**
* Initialize the Modulator_AXI instance provided by the caller based on the
* given DeviceID.
*
* Nothing is done except to initialize the InstancePtr.
*
* @param InstancePtr is a pointer to an Modulator_AXI instance. The memory the
* pointer references must be pre-allocated by the caller. Further
* calls to manipulate the instance/driver through the Modulator_AXI API
* must be made with this pointer.
*
* @param DeviceId is the unique id of the device controlled by this Modulator_AXI
* instance. Passing in a device id associates the generic Modulator_AXI
* instance to a specific device, as chosen by the caller or
* application developer.
*
* @return
* - XST_SUCCESS if the initialization was successful.
* - XST_DEVICE_NOT_FOUND if the device configuration data was not
*     found for a device with the supplied device ID.
*
* @note None.
*
****************************************************************************/

int modulator_axi_ip_Initialize(Modulator_AXI *InstancePtr, u16 DeviceId)
{
    modulator_axi_ip_Config *ConfigPtr = NULL;
    int Index;

    // Assert arguments
    Xil_AssertNonvoid(InstancePtr != NULL);

    /* Lookup configuration data in the device configuration table.
    * Use this configuration info down below when initializing this
    * driver. */
    for (Index = 0; Index < XPAR_MODULATOR_AXI_IP_NUM_INSTANCES; Index++)
        ...
```c
if (modulator_axi_ip_ConfigTable[Index].DeviceId == DeviceId)
{
    ConfigPtr = &modulator_axi_ip_ConfigTable[Index];
    break;
}

if (ConfigPtr == (modulator_axi_ip_Config *) NULL)
{
    InstancePtr->IsReady = 0;
    return (XST_DEVICE_NOT_FOUND);
}

// Initialize the driver
InstancePtr->BaseAddress = ConfigPtr->BaseAddress;
InstancePtr->LUT_Width = ConfigPtr->LUT_Size;
InstancePtr->LUT_Depth = ConfigPtr->LUT_Depth;
InstancePtr->NCO_Width = ConfigPtr->NCO_Width;

// Indicate the instance is now ready to use, initialized without error
InstancePtr->IsReady = XIL_COMPONENT_IS_READY;
return (XST_SUCCESS);
}

void modulator_axi_ip_Set_Input_Clock_Frequency (Modulator_AXI *InstancePtr, int clock_rate)
{
    InstancePtr->ClockRate = clock_rate;
}

void modulator_axi_ip_Set_Register(int baseaddr, int offset, int value)
{
    Xil_Out32(baseaddr + offset, value);
}

int modulator_axi_ip_Get_Register(int baseaddr, int offset)
{
    int temp = 0;
    temp = Xil_In32(baseaddr + offset);
    return (temp);
}

void modulator_axi_ip_Set_PWM_Frequency (Modulator_AXI * InstancePtr, int freq_sel, float freq_val)
{
    // c = 2^NCO_WIDTH
    float c = 1.0;
    for (int i = 1; i <= InstancePtr->NCO_Width; i++)
        c = c * 2;

    // inc_factor = C * F_LOW / CLOCK_RATE
    int inc_factor = (int)((c * freq_val / (float)InstancePtr->ClockRate) * 0.5);
    modulator_axi_ip_Set_Register(InstancePtr->BaseAddress, 4+4*freq_sel, inc_factor);
}

float modulator_axi_ip_Get_PWM_Frequency (Modulator_AXI * InstancePtr, int freq_sel)
{
    // c = 2^NCO_WIDTH
    float c = 1.0;
    for (int i = 1; i <= InstancePtr->NCO_Width; i++)
        c = c * 2;

    float inc_factor = (float) modulator_axi_ip_Get_Register(InstancePtr->BaseAddress, 4+4*freq_sel);

    // freq = inc_factor * CLOCK_RATE / C
    float temp = (inc_factor * (float)InstancePtr->ClockRate / c);
    return temp;
}

void modulator_axi_ip_Select_PWM_Frequency (Modulator_AXI * InstancePtr, int freq_sel)
{
    modulator_axi_ip_Set_Register(InstancePtr->BaseAddress, 0, freq_sel);
}

int modulator_axi_ip_Get_Selected_PWM_Frequency (Modulator_AXI * InstancePtr)
{
    return modulator_axi_ip_Get_Register(InstancePtr->BaseAddress, 0);
}
```
CHAPTER 4. USING CUSTOM IPS IN EMBEDDED SYSTEM DESIGN

**modulator_axi_ip.h:**

```c
#ifndef MODULATOR_AXI_IP_H /* prevent circular inclusions */
#define MODULATOR_AXI_IP_H /* by using protection macros */

#include "xil_types.h"

/**************************************************************************
** This typedef contains configuration information for the device.         
**
** typedef struct {
**  u16 DeviceId; /* Unique ID of device */
**  u32 BaseAddress; /* Device base address */
**  u32 LUT_Depth; /* Number of samples in one period of the signal */
**  u32 LUT_Width; /* Number of bits used to represent amplitude value */
**  u32 NCO_Width; /* Number of bits used for numerically controlled oscillator */
**} modulator_axi_ip_Config; 
/**
** The Modulator_AXI driver instance data. The user is required to allocate a 
** variable of this type for every Modulator_AXI device in the system. A pointer 
** to a variable of this type is then passed to the driver API functions. 
**
** typedef struct {
**  u32 BaseAddress; /* Device base address */
**  u32 IsReady; /* Device is initialized and ready */
**  u32 ClockRate; /* Input clock frequency, specified in Hz */
**  u32 LUT_Depth; /* Number of samples in one period of the signal */
**  u32 LUT_Width; /* Number of bits used to represent amplitude value */
**  u32 NCO_Width; /* Number of bits used for numerically controlled oscillator */
**} Modulator_AXI; 

int modulator_axi_ip_Initialize(Modulator_AXI *InstancePtr, u16 DeviceId);
void modulator_axi_ip_Set_Input_Clock_Frequency (Modulator_AXI *InstancePtr, int clock_rate);
void modulator_axi_ip_Set_Register(int baseaddr, int offset, int value);
int modulator_axi_ip_Get_Register(int baseaddr, int offset);
void modulator_axi_ip_Set_PWM_Frequency (Modulator_AXI * InstancePtr, int freq_sel, float freq_val);
float modulator_axi_ip_Get_PWM_Frequency (Modulator_AXI * InstancePtr, int freq_sel);
void modulator_axi_ip_Select_PWM_Frequency (Modulator_AXI * InstancePtr, int freq_sel);
int modulator_axi_ip_Get_Selected_PWM_Frequency (Modulator_AXI * InstancePtr);
#endif /* end of protection macro */
```

4.4 Creating Xilinx driver file and folder structure

Folder Structure

When the driver source files have been written, the next step is to put the driver files into a directory tree structure that the Xilinx SDK will understand.

This is a very important step because the Xilinx tools will expect to find files and folders with reserved names.

An example of the required folder structure is show on the following figure.

![Figure 4.51: Required folder structure](image)

**Required Folder Structure**

- Create the folder structure in the same way as it is shown on the previous figure, wherever you like on the disk.
The top level folder can be called anything that the user chooses, and be placed anywhere that they choose, but must contain a sub-folder called "drivers".

Below that, there may be one or more folders which represent each driver for the custom IP.

We have just one driver in our example which is called "modulator_axi_v1_00_a".

**Driver Sub-folders**

Under the structure described above, three further folders are expected:

- The "src" folder contains the .c and .h source files for the driver, in addition to a Makefile which can be written to describe the build process required to compile the sources in the correct order / dependency.
- The "example" folder contains examples of software application code, showing how your driver must be used in a final application.
- The "data" folder contains the following control files which are specific to the operation of the Xilinx SDK tools, and which detail how various device driver source files should be used:
  - the MDD file - Microprocessor Driver Definition file
  - the TCL file - used by the Xilinx BSP creation tools to automatically generate some parameters which can be used later by the software developer

**Save the modulator_axi_ip.c and modulator_axi_ip.h in the "src" Folder**

Save the source files, modulator_axi_ip.c and modulator_axi_ip.h, created in the previous sub-chapter in the "src" folder, because the "src" folder should contain the .c and .h source files for the driver.

**The MDD File**

The first control file in the "data" folder is the Microprocessor Driver Definition (MDD) file.

The file name is required to have a suffix of "v2_1_0.mdd", and in the case of our example has the full name of "modulator_axi_v2_1_0.mdd".

The suffix relates to the version of the syntax that is used within the MDD file, in this case v2.10.

The content of the MDD file is relatively simple, and for most MDD files the text is identical with the exception of one or two lines.

In our case, the content of the "modulator_axi_v2_1_0.mdd" MDD file is the following:

```
modulator_axi_v2_1_0.mdd:

OPTION psf_version = 2.1;
BEGIN DRIVER modulator_axi
  OPTION supported_peripherals = (modulator_axi_ip);
  OPTION driver_state = ACTIVE;
  OPTION copyfiles = all;
  OPTION VERSION = 1.0;
  OPTION NAME = modulator_axi;
END DRIVER
```

The "OPTION supported_peripherals" line should be updated to list the peripherals that are served by the custom driver. In the example shown here, there is just one peripheral which will be supported by the driver, but multiple peripherals can be listed, separated by a space. The names of supported peripherals must match the names of the IPs that have been created using the IP Packager. In our case driver supports only one IP, modulator_axi_ip.
The "BEGIN driver" line should be edited to create a name for the driver that is being developed. There are no specific rules for the naming convention of this parameter, but it is wise to create a name that will be clearly identifiable to the end user. In our case, a name for the driver will be `modulator_axi`. Identical name should be used in the "OPTION NAME" line of the MDD file.

The "OPTION driver_state" line allows the user to maintain a recognised lifespan of their custom driver, and list the driver as either ACTIVE, DEPRECATED, or OBSOLETE as and when the developer chooses to supersede or retire it from the service. The effect of changing this option away from the "active" state, generates either warnings or errors in the driver compile process when it is invoked by the end user. Driver that we are developing is in the active state, so this option is set to ACTIVE.

The "OPTION copyfiles" line tells the Xilinx SDK which source files in the custom driver's "src" directory should be copied when the SDK generates the BSP. In most cases this will be left set to "all".

The "VERSION" line allows the user to specify a version number for their driver. This should match the directory name used previously.

- Create `modulator_axi_v2_1_0.mdd` control file, following instructions from the text above, and save it into the "data" folder.

The TCL File

The second control file in the "data" folder is the TCL file, which is used by the Xilinx BSP creation tools to automatically generate some parameters which can be used later by the software developers.

The content of the TCL file is shown in the following text.

```
proc generate {drv_handle} {
::his::utils::define_include_file $drv_handle "xparameters.h" "modulator_axi_ip" "NUM_INSTANCES" "DEVICE_ID" "C_S00_AXI_BASEADDR" "C_S00_AXI_SRAMADDR" "lut_depth_g" "lut_width_g" "nco_width_g" "FCLK_CLK0"
::his::utils::define_config_file $drv_handle "modulator_axi_driver_g.c" "modulator_axi_ip" "DEVICE_ID" "C_S00_AXI_BASEADDR" "lut_depth_g" "lut_width_g" "nco_width_g"
::his::utils::define_canonical_xpars $drv_handle "xparameters.h" "modulator_axi_ip" "NUM_INSTANCES" "DEVICE_ID" "C_S00_AXI_BASEADDR" "lut_depth_g" "lut_width_g" "nco_width_g"
}
```

- Create `modulator_axi_v2_1_0.tcl` control file, following instructions from the text above, and save it into the "data" folder.

The TCL file shown above, that is part of a device driver we are developing, is actually only 4 lines in length, including the final curly bracket on its own line, but lines 2, 3 & 4 are extremely long and therefore difficult to reproduce clearly in this document.

The lines 2 & 4 should be edited by the developer (beginning with "define_include_file" and "define_canonical_xpars"), to list a number of parameters that will be generated automatically by the BSP generation tools before being placed into a file called "xparameters.h" within the automatically generated board support package.

The editable section of these lines begins with the parameter after "xparameters.h". In this example, the first two editable parameters are "modulator_axi_ip" and "NUM_INSTANCES". These two parameters are used to create a #define statement in the "xparameters.h" file called "XPAR_MODULATOR_AXI_IP_NUM_INSTANCES" and assign a numerical value to it. The Xilinx BSP generation tools have the ability to automatically count the number of instances of each type of peripheral that are added into the user's embedded processor design. This information can be of great use when the same driver is used to control multiple instances of the same peripheral, and provides the ability for a loop to be created in software that will automatically update when the number of instances of a given peripheral is increased and decreased in the design. In our example there is only one instance of the modulator_axi_ip peripheral, so we need not worry about such a feature, but the line "#define XPAR_MODULATOR_AXI_IP_NUM_INSTANCES 1" will be automatically added to the "xparameters.h" file when the BSP is generated.
The list of parameters in the TCL file on the rest of lines 2 and 4 represent additional `#define` statements that will be generated in the "xparameters.h" file.

Each of the `#define` statements will have a prefix related to the instance name of the IP, and a suffix copied from the text string in quotes on lines 2 and 4 of the TCL file.

Each of the text strings matches the name of a Generic in the top level VHDL entity for the custom IP, and the value of each `#define` is populated using either the Generic’s default value in the VHDL code, or the value that the user has set in the Vivado block diagram tool to override that default.

In our case, eight `#define` statements will be created in the "xparameters.h" file during BSP generation, and they will be populated according to the user’s chosen settings and the number of instances of the custom IP that were added to the user’s design.

In our case, a section of "xparameters.h" file related to the `modulator_axi_ip` peripheral, that will be automatically generated, is shown below:

```c
/* Definitions for driver MODULATOR_AXI */
#define XPAR_MODULATOR_AXI_IP_NUM_INSTANCES 1
/* Definitions for peripheral MODULATOR_AXI_IP_0 */
#define XPAR_MODULATOR_AXI_IP_0_DEVICE_ID 0
#define XPAR_MODULATOR_AXI_IP_0_S00_AXI_BASEADDR 0x43C00000
#define XPAR_MODULATOR_AXI_IP_0_S00_AXI_HIGHADDR 0x43C0FFFF
#define XPAR_MODULATOR_AXI_IP_0_LUT_DEPTH_G 12
#define XPAR_MODULATOR_AXI_IP_0_LUT_WIDTH_G 31
#define XPAR_MODULATOR_AXI_IP_0_FCLK_CLK0 0
```

Tcl command in line 3 is used to automatically generate `modulator_axi_ip` configuration table array object, which was discussed in the `modulator_axi_ip_Initialize` function. This time, the editable section of this line begins with the parameter after $drv_handle. First parameter specifies the filename of the C source code file ("modulator_axi_driver_g.c", in our case) that will be automatically generated by the Xilinx SDK tool during the BSP generation process. This C file will hold a definition of the configuration table related to the IP, whose name is specified by the second parameter ("modulator_axi_ip"). Remaining parameters in the line define the fields that configuration table array member should contain (in our case each member should consist from four fields, "DEVICE_ID" "FC_S00_AXI_BASEADDR" "lut_depth_g" "lut_width_g" "nco_width_g").

As before, each of the text strings must match the name of a Generic in the top level VHDL entity for the custom IP, and the values of these fields will be populated automatically using either the Generic’s default value in the VHDL code, or the value that the user has set in the Vivado block diagram tool to override that default.

The ability to generate a software BSP which will automatically update itself based upon changes made to the hardware design is an incredibly powerful feature, and can save the software engineering team a lot of manual development effort and time. The end user’s software application can then make use of these parameters, allowing the hardware and software teams to work completely independently, yet vastly reduce the possibility for software errors and bugs to occur due to any changes made in the hardware design that were not manually communicated to the software engineering team.

**Configuring the Xilinx SDK**

With the driver control files written and placed in the correct folder structure, the final step is to configure the Xilinx SDK tools so that they have visibility of the new driver in the list of available driver repositories.

**Configuring the Xilinx SDK**

- In the SDK, open the Preferences dialog by choosing Window -> Preferences from the menu bar.
- In the Preferences dialog box, select the Xilinx SDK -> Repositories pane.
In the **Preferences** dialog box, click the **New...** button next to the **Local Repositories** list to add a new repository.

In the **Browse For Folder** dialog box, point to the folder that you created for your custom drivers and click **OK**.

The folder you choose here should be the level of the file system above the "drivers" folder. In our case it is "pwm_modulator_driver" folder.

Click the **Rescan Repositories** button, followed by **OK** button.

This setting will provide visibility of your custom driver to the SDK tool, and will enable your driver to be selected in the BSP settings.

There is an addition list shown on this screen called **Global Repositories**, see Figure 5.51. This performs an identical function to that of adding the drivers to the "Local Repositories" list, with the exception that the drivers will be visible to any SDK workspace that is created or opened in the future. This is a powerful feature if your goal is to create a single repository of custom drivers for multiple custom IPs, and still have them routinely available and visible to all of your SDK workspaces.
CHAPTER 4. USING CUSTOM IPS IN EMBEDDED SYSTEM DESIGN

Selecting a Custom Driver in the BSP

The task of creating and configuring your custom IP and custom driver is now complete, and the Vivado and SDK tools will now have visibility of them. The final stage is to select your custom driver and allocate it to be automatically compiled as part of the BSP for your user application.

Selecting a Custom Driver in the BSP

- Right-click on the Board Support Package in the Project Explorer that you created just before (standalone_bsp_0), and choose the Board Support Package Settings menu item from the bottom of the list.

- In the Board Support Package Settings dialog box, choose the drivers pane from the choices shown on the left of the window.

- In the Drivers configuration table, identify the instance of your custom IP (modulator_axi). It will now be possible to select your custom driver from the drop down menu in the Driver column of the table.

- Click OK and the BSP will automatically be re-generated.

Figure 4.54: Board Support Package Settings dialog box with selected modulator_axi custom driver

Note: If you had created multiple folders representing different versions of the same driver, you will also be able to choose the version of the driver in the Driver Version column.

If you now examine the "include" and "libsrc" folders in the standalone_bsp_0 BSP’s source tree, you will find that your header file (modulator_axi_ip.h) and C source file (modulator_axi_ip.c) have been automatically copied and compiled into the BSP. Furthermore, one additional C source file is automatically generated, modulator_axi_driver_g.c. Remember that this is the source file we have specified in the modulator_axi_v2_1_0.tcl TCL file. This source file contains the configuration table array object, holding all relevant configuration information for every modulator_axi_ip peripheral that is present in the embedded

www.so-logic.net 2019/01/28 131
system. In our case, we have only one instance of the `modulator_axi_ip` peripheral, so the generated configuration table array object has only one entry, as can be seen by inspecting the generated `modulator_axi_driver.c` source file. There is no longer any requirement for custom driver functions to be added to the list of sources for each application, because they are now included in the BSP alongside the drivers provided by Xilinx for supplied IP.

Creating New Application Project that will use Developed Device Driver

Creating New Application Project that will use Developed Device Driver

The last step in our design process will be to create a new application project that will use developed `modulator_axi` device driver.

One possible solution is shown in the `modulator_axi_using_driver.c` and `modulator_axi_using_driver.h` source files.

Note: For example, these two files can be stored in the "examples" folder from the device driver folder structure.

```
modulator_axi_using_driver.c:

#include "xparameters.h"
#include "xgpiops.h"
#include "xstatus.h"
#include "modulator_axi_using_driver.h"
#include "modulator_axi_ip.h"

int main(void)
{
  /************** Variable Definitions **************/
  XGpioPs GpioSwitches;  // XGPIO instance that will be used to work with SWITCHes
  XGpioPs_Config *GPIOConfigPtr;
  Modulator_AXI pwm_modulator;  // Modulator AXI instance
  int sel = 0;  // switch used for selecting frequency
  int current_sel = 0;  // current value of the sel bit in the modulator_axi_ip IP core
  float temp;
  /************** Initialization **************/
  // SWITCHes initialization
  GPIOConfigPtr = XGpioPs_LookupConfig(XPAR_PS7_GPIO_0_DEVICE_ID);
  XGpioPs_CfgInitialize(&GpioSwitches, GPIOConfigPtr, GPIOConfigPtr ->BaseAddr);
  XGpioPs_SetDirectionPin(&GpioSwitches, SWITCH_CHANNEL, 0);
  // modulator_axi_ip IP core initialization
  modulator_axi_ip_Initialize(&pwm_modulator, XPAR_MODULATOR_AXI_IP_0_DEVICE_ID);
  // Set the input clock frequency for the Modulator_AXI instance to appropriate value
  modulator_axi_ip_Set_Input_Clock_Frequency (&pwm_modulator, CLOCK_RATE);
  // Set the values for two possible frequencies for the PWM signal
  modulator_axi_ip_Set_PWM_Frequency (&pwm_modulator, 0, F_HIGH);
  temp = modulator_axi_ip_Get_PWM_Frequency (&pwm_modulator, 0);
  modulator_axi_ip_Set_PWM_Frequency (&pwm_modulator, 1, F_LOW);
  temp = modulator_axi_ip_Get_PWM_Frequency (&pwm_modulator, 1);
  // Select the initial frequency that will be used to generate PWM signal
  modulator_axi_ip_Select_PWM_Frequency (&pwm_modulator, sel);
  current_sel = modulator_axi_ip_Get_Selected_PWM_Frequency (&pwm_modulator);

  /************* Main Loop *************
  while(1)
  {
    // read the switch position
    sel = XGpioPs_ReadPin(&GpioSwitches, SWITCH_CHANNEL);
    sel = sel & SWITCH_POS;  // masking (we want to check the status of sel only)
    if (sel != current_sel)
    {
      // Write the new switch position to the pwm_modulator core
      modulator_axi_ip_Select_PWM_Frequency (&pwm_modulator, sel);
      current_sel = modulator_axi_ip_Get_Selected_PWM_Frequency (&pwm_modulator);
    }
  }
```

```
modulator_axi_using_driver.h:

```c
#ifndef MODULATOR_H_
#define MODULATOR_H_
#include "math.h"
#include "xparameters.h"

/********************** Constant Definitions **********************
#define SWITCH_CHANNEL 12 // GPIO channel to operate on
#define SYS_CLK_MHZ 50 // 50 MHz system clock
#define CLOCK_RATE 1000000 * SYS_CLK_MHZ // system clock value, expressed in Hz
    // CLOCK_RATE = 50000000 Hz (= 50 MHz)
#define F_LOW 1.0 // F_LOW = 1 Hz
#define F_HIGH 3.5 // F_HIGH = 3.5 Hz
#define SWITCH_POS 0x01 // mask to select switch position
#endif /* MODULATOR_H_ */
```

Creating a New Application Project

- In the SDK, select File -> New -> Application Project option from the main SDK menu.

- In the Application Project dialog box:
  - type a name of the new application project in the Project name field. In our case the name will be modulator_axi_using_driver
  - in the Target Software section use Use existing standalone BSP 0 Board Support Package instead of creating new
  - leave all other parameters unchanged
  - click Next and choose Empty Application as a default template
  - click Finish and the new modulator_axi_using_driver application project should appear in the Project Explorer window

Run Application Project with Developed Device Driver

- Expand modulator_axi_using_driver application project in the Project Explorer window and in the src folder add previously created modulator_axi_using_driver.c and modulator_axi_using_driver.h files.

- Run your application on the Sozias development board, on the same way as it is explained in the Sub-chapter 3.6.1 "Downloading ARM-based bitstream file" and you should get the same results like in the Sub-chapter 4.2 "Debug a Design using Integrated Vivado Logic Analyzer".
Chapter 5

CONCLUSION

In this tutorial a range of different implementations of the PWM Modulator system has been presented:

1. A software solution that uses an external timer, \texttt{PWM\_SW\_no\_inte}
2. A software solution that uses an external timer and interrupt controller, \texttt{PWM\_SW\_inte}
3. A hardware/software co-design solution, that uses a custom PWM Modulator IP core, \texttt{PWM\_HW/SW}

Furthermore, in the "Basic FPGA Tutorial" a pure hardware solution for the PWM Modulator system has also been presented, \texttt{PWM\_HW}.

All these solutions represent valid and functionally correct implementations of the PWM Modulator system, but they differ in a way how this functionality is actually implemented and therefore have different performance results and resource requirements.

The performance of the PWM Modulator system can be represented by three attributes:

1. Maximum frequency of generated PWM signal
2. Maximum accuracy of duty cycle value of generated PWM signal
3. Minimum jitter value of generated PWM signal

Maximum frequency of generated PWM signal is probably the most important performance attribute of any PWM Modulator system. Typical PWM signal frequency values range from 0.05 Hz, in case of an electric stove, 120 Hz, in case of a lamp dimmer, from few KHz to tens of KHz for a motor drive, and well into the tens or hundreds of kHz in audio amplifiers and computer power supplies.

Four proposed PWM Modulator systems have different maximum PWM frequency values. \texttt{PWM\_HW} and \texttt{PWM\_HW/SW} solutions can generate PWM signals with the highest frequency values, because the PWM signal generation is done completely in hardware. In the proposed hardware solution, maximum PWM frequency is limited by two factors:

- maximum operating frequency of the Frequency Trigger module (see Chapter 2 "Frequency Trigger" from the "Basic FPGA Tutorial"), \( F_{\text{max}} \), and
- the number of bits used to represent amplitude value of the PWM modulating signal ("Width G" field from the \texttt{modulator\_axi\_ip} customization window, shown on the Figure 5.12, or width field in the \texttt{design\_setting\_g} generic from the modulator top-level entity, see Chapter 8 "Modulator" from the "Basic FPGA Tutorial"), \( \text{width} \).

The maximum PWM frequency that can be generated by the \texttt{PWM\_HW} and \texttt{PWM\_HW/SW} solutions can be calculated by the following formula:

\[
PWM_{F_{\text{max}}} = \frac{F_{\text{max}}}{2^{\text{width}}}
\]
For example, maximum PWM signal frequency that can be generated with the configuration of the PWM Modulator IP core that was used in this tutorial is equal to:

$$PWM_{F_{max}} = \frac{100MHz}{2^{12}} = 24414Hz$$

This value is sufficient for the majority of PWM applications, described earlier.

Estimating the maximum PWM signal frequencies for the software solutions (PWM_{SW\_no\_inte} and PWM_{SW\_inte}) is not so straightforward. In these implementations maximum PWM frequency is also constrained by the amount of time required to complete one iteration of the while loop located in the main procedures from the modulator\_no\_inte\_mb.c, modulator\_inte\_mb.c, modulator\_no\_inte\_axi.c and modulator\_inte\_axi.c source codes. The estimations of exact maximum PWM signal frequency values would require a measurement of this while loop time. Furthermore, this time would also depend on the type of the processor that executes the PWM Modulator software and the C/C++ compiler settings that were used to build the executable version.

Next, we will comment on the pros and cons of each of the proposed implementation of the PWM Modulator system.

PWM\_SW\_no\_inte implementation

Pros:

- Easiest implementation with the shortest design time.
- Easy to modify the used PWM generator algorithm, for instance easy to change the waveform of the modulating signal, all we need to do is to recompile the modified PWM Modulator source code.

Cons:

- Maximum frequency of generated PWM signal is the lowest from all proposed implementations, because the duration of one iteration of the while loop from the modulator\_no\_inte\_mb.c and modulator\_no\_inte\_axi.c source codes is the longest. This duration becomes even longer if the processor needs to carry out some additional tasks other than PWM generation, because all these tasks would have to be included in this while loop.
- Jitter of generated PWM signal is significant, and potentially can be very high, but it can be relatively accurately estimated (since we know the amount and ordering of additional commands that are located in the while loop) and compensated.

PWM\_SW\_inte implementation

Pros:

- More difficult to implement than the PWM\_SW\_no\_inte implementation, because it requires the configuration of the interrupt controller and writing the interrupt routine.
- In case a processor has to perform some other actions apart from the PWM signal generation, this is the only acceptable solution (from "pure" software solutions).
- Easy to modify the used PWM generator algorithm, for instance easy to change the waveform of the modulating signal, all we need to do is to recompile the modified PWM Modulator source code.

Cons:

- Maximum frequency of generated PWM signal should be a little bit higher than the maximum PWM signal frequency generated by the PWM\_SW\_no\_inte implementation, because the duration of one iteration of the while loop in this case is shorter.
- Jitter of generated PWM signal is significant, and potentially can be very high, but this time it cannot be predicted in advance, because we cannot estimate the actual point in time when an PWM timer interrupt will be generated. This is even more unpredictable if there are other interrupt sources with equal or higher interrupt priority present in the system.
PWM_HW/SW implementation

Pros:
- Offers the generation of PWM signal with the highest possible frequency, identical to the PWM_HW implementation, because in this implementation, as well as PWM_HW implementation, PWM signal generation is done completely in hardware.
- Jitter of generated PWM signal is minimal.
- Great flexibility (for example, easy adjusting of frequencies of generated PWM signals, easy integration into more complex systems like multiple PWM generators, etc.), especially when compared with the PWM_HW implementation.

Cons:
- If PWM Modulator IP core is not readily available, then the development time is significant, however if this IP is available then the development time is comparable with the "pure" software solutions.

PWM_HW implementation

Pros:
- Offers the generation of PWM signal with the highest possible frequency, identical to the PWM_HW implementation, because in this implementation, as well as PWM_HW implementation, PWM signal generation is done completely in hardware.
- Jitter of generated PWM signal is minimal.
- Minimum amount of required hardware resources, minimum power consumption.

Cons:
- Limited flexibility, because every modification of the design requires a modification of hardware, leading to the necessity to reimplement the complete system (redo the hardware synthesis, place and route the design), which in large systems can take several hours.
- Long development time, because we need to design and verify a complete hardware system, which is much more time consuming that software design.
Chapter 6

EXERCISES

This section holds a set of exercises that can be used to verify the knowledge acquired by reading this tutorial.

Exercise 1

Modify the init_sin.c source code in order to be able to generate PWM signal modulated by the following signals:

1. Triangle wave signal
2. Sawtooth wave signal

Exercise 2

Make the necessary modifications to the ARM-based embedded system and modulator_no_intc.c application to enable the user to specify which type of modulating signal (sine, triangle or sawtooth) should be used to generate PWM signal on-the-fly, during the operation of the system.

Exercise 3

Using the ARM-based embedded system and modulator_no_intc.c application as a starting point, make the necessary modifications to both the hardware and software parts of the ARM-based embedded system in order to be able to generate two completely independent PWM generators.

Exercise 4

Using the ARM-based embedded system and modulator_intc.c application as a starting point, make the necessary modifications to both the hardware and software parts of the ARM-based embedded system in order to be able to generate two completely independent PWM generators.

Exercise 5

Using the ARM-based embedded system based on the PWM modulator custom IP core and modulator_axi.c application as a starting point, make the necessary modifications to both the hardware and software parts of the ARM-based embedded system in order to be able to generate two completely independent PWM generators.

Exercise 6

Using the ARM-based embedded system based on the PWM modulator custom IP core, modulator_axi driver and modulator_axi_using_driver.c application as a starting point, make the necessary modifications to both the hardware and software parts of the ARM-based embedded system in order to be able to generate two completely independent PWM generators.