SDSoC Environment Platform Development Guide

UG1146 (v2019.1) June 5, 2019

See all versions of this document

Revision History

The following table shows the revision history for this document.

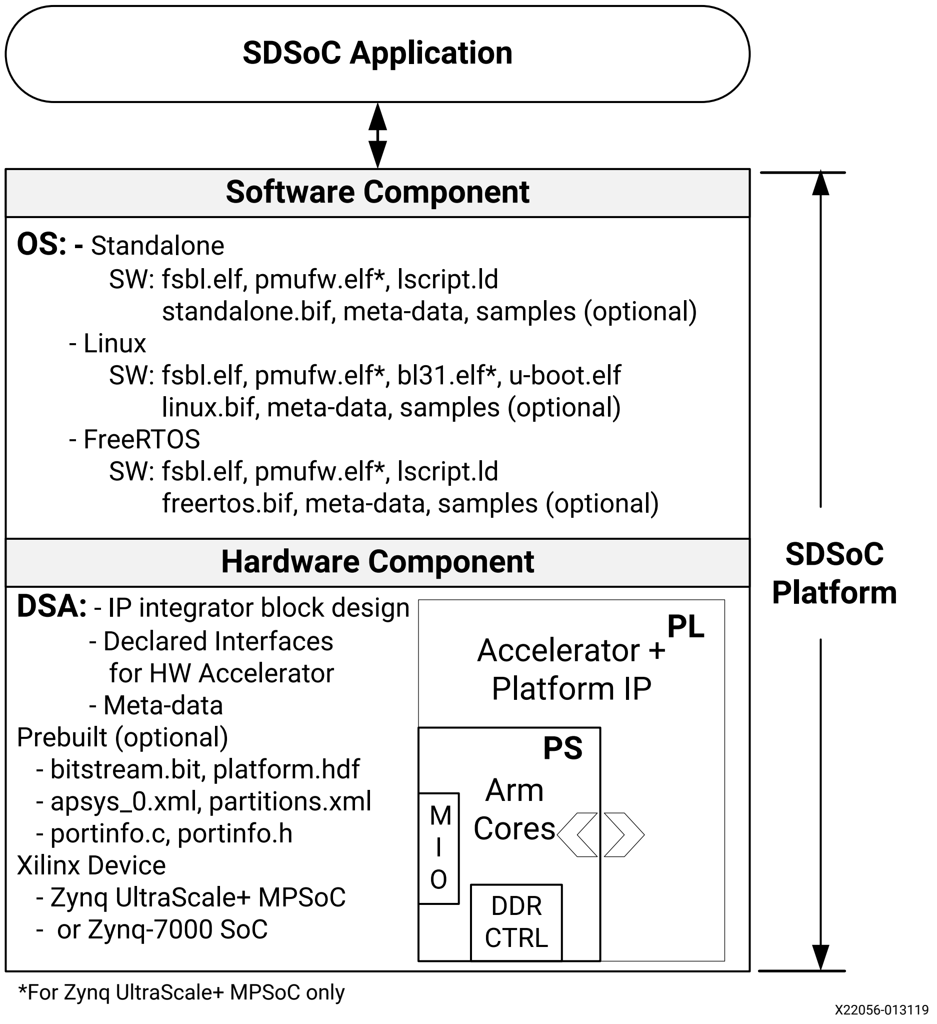
Section	Revision Summary
06/05/2019 Version 2019.1	
Throughout the document	Updated for 2019.1 support.
01/24/2019 Version 2018.3	
Throughout the document	General updates.
Chapter 2: Creating SDSoC Platforms	Updated process and reordered content.
12/05/2018 Version 2018.3	
Throughout the document	Reordered chapters/appendices for example flows.
Throughout the document	Updated for 2018.3 support.
Creating an SDSoC Platform Project	Provided example SDSoC platform for Standalone and Linux targets using files from the SDx zcu102 base platform.
Chapter 3: Creating the Platform Hardware Component	Provided example hardware component (DSA) for a custom platform that runs on a zcu102 board.
Chapter 4: Creating the Platform Software Component	Provided Standalone and Linux software components for the custom platform.
References	Added reference to *PetaLinux Tools Documentation: Reference Guide* (UG1144)
07/02/2018 Version 2018.2	
Using an SDx Workspace	Updated link to release notes.
06/06/2018 Version 2018.2	
Chapter 5: Sample Applications	Replaced `template.xml` with `description.json`.
	Minor editorial updates throughout.
04/04/2018 Version 2018.1	
Prebuilt Hardware	Updated text to reflect the process of populating the **Prebuilt Data** for a platform project with files from an application project.

Table of Contents

Introduction

The software-defined system-on-chip SDSoC™ environment provides the tools necessary to implement heterogeneous embedded systems for Zynq® UltraScale+™ MPSoC and Zynq®-7000 SoC devices. This document describes how to create an SDSoC platform using the Eclipse-based SDx™ integrated development environment (IDE) with a hardware design generated by the Vivado® Design Suite.

Figure 1: **Platform Components**


SDSoC Application
Software Component
OS: - Standalone
SW: fsbl.elf, pmufw.elf*, lscript.ld
standalone.bif, meta-data, samples (optional)
- Linux
SW: fsbl.elf, pmufw.elf*, bl31.elf*, u-boot.elf
linux.bif, meta-data, samples (optional)
- FreeRTOS
SW: fsbl.elf, pmufw.elf*, lscript.ld
freertos.bif, meta-data, samples (optional)
Hardware Component
DSA: - IP integrator block design
- Declared Interfaces for HW Accelerator
- Meta-data
Prebuilt (optional)
- bitstream.bit, platform.hdf
- apsys_0.xml, partitions.xml
- portinfo.c, portinfo.h
Xilinx Device
- Zynq UltraScale+ MPSoC
- or Zynq-7000 SoC
PL
Accelerator + Platform IP
PS
Arm Cores
MIO
DDR CTRL
SDSoC Platform
*For Zynq UltraScale+ MPSoC only
X22056-013119

The concept of a platform is integral to the SDSoC environment, as it defines the hardware and software components as well as the meta-data on which SDSoC applications are built. The above figure illustrates the platform and its components. An SDSoC platform defines a base hardware/ software architecture and an application context including the processing system, external memory interfaces, custom input/output, a software runtime with an operating system (possibly "bare-metal"), boot-loaders, drivers for platform peripherals and a root file system. Every project created with the SDx IDE targets a specific platform and is customized with application-specific hardware accelerators and data motion networks.

A platform developer designs the platform's hardware component by first implementing a hardware design using the Vivado Design Suite and its IP integrator design canvas. The IP integrator block design is created with platform interfaces that are enabled for use by the `sds++/sdscc` (referred to as `sds++`) system compiler as attachment points for generated hardware accelerators and data movers. This hardware design component along with its meta-data are encapsulated into a Device Support Archive (DSA). The DSA contains the Vivado IP integrator design as well as the required processor and memory system configuration, all board interfaces and I/O connections. Platform properties are also defined in the DSA for platform identification and interface configuration.

The platform developer must also provide any boot loaders and target operating systems used to bring-up the platform. A platform optionally includes software libraries for linking with applications. If a platform supports a Linux target operating system, the Petalinux tools can be used to configure and build the Linux kernel, produce a U-Boot bootloader and a root file system. The Xilinx SDK provides code templates for developing software components for a platform and can be customized. The SDx IDE can also be used to generate software components but for customization it is recommended that developers use the SDK or PetaLinux tools directly.

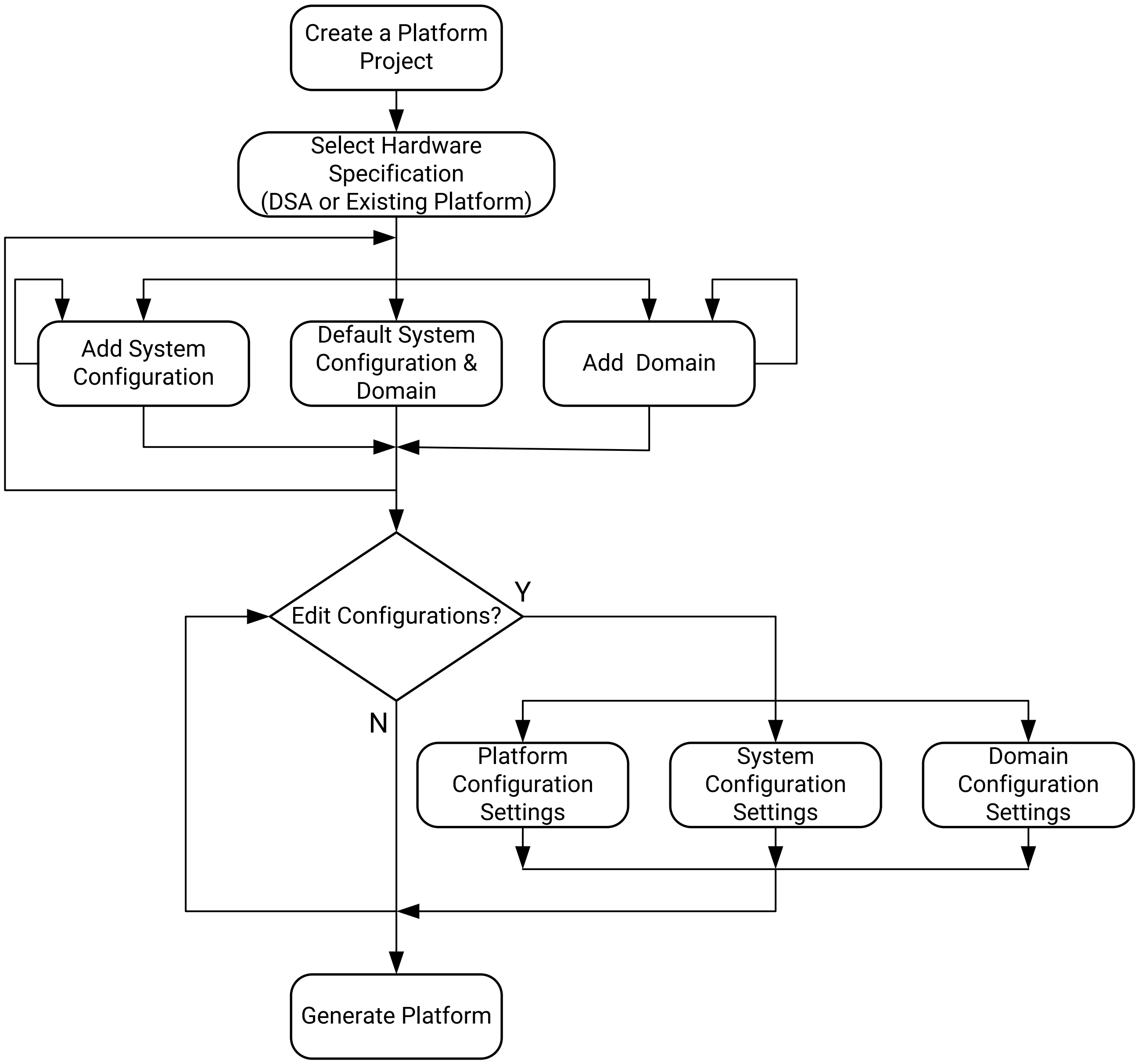
Creating a platform is accomplished by generating or gathering together the hardware and software components for use in an SDx Platform project. The SDx tools can use an existing platform as a basis for a new platform or use the hardware defined by a user-provided input DSA. In a similar manner, the SDx environment can use a set of existing software objects created by the developer prior to invoking the SDx tools, or generate the software objects as part of the SDSoC platform creation process.

The example platform that is described and built in this document uses the hardware and software components that are provided as part of the ZCU102 platform included with the SDx tools. After going through the steps of using the SDx Platform project flow to assemble a platform, this document describes how to build the hardware and software components that make up a custom platform.

Creating SDSoC Platforms

The SDx™ IDE is used to assemble the hardware and software components into an SDSoC™ platform. The design flow for generating a platform is shown below and is utilized in the platform creation example that follows.

Figure 2: **Platform Generation Design Flow**

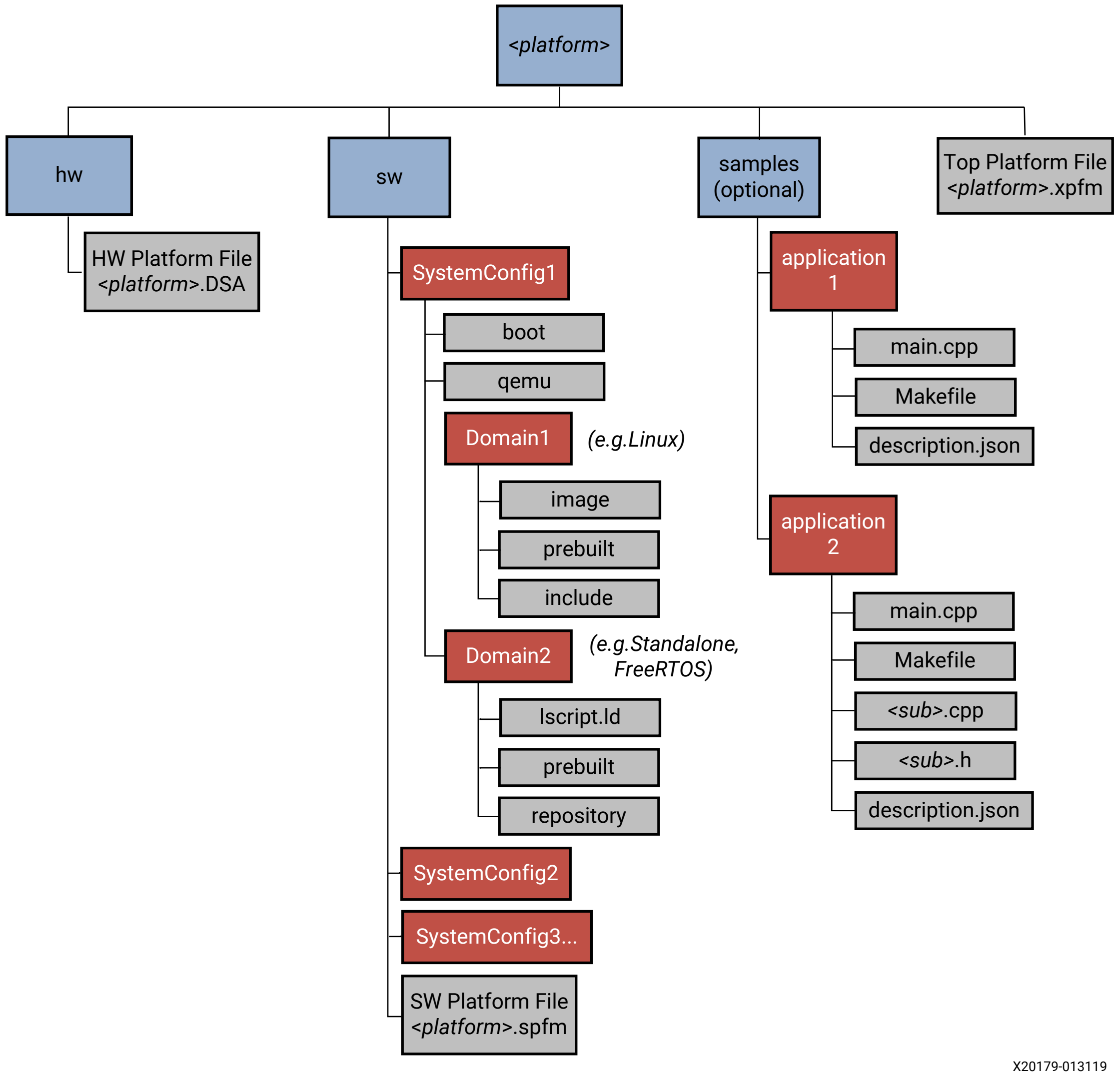

Create a Platform Project
Select Hardware Specification (DSA or Existing Platform)
Add System Configuration
Default System Configuration & Domain
Add Domain
Edit Configurations?
Y
N
Platform Configuration Settings
System Configuration Settings
Domain Configuration Settings
Generate Platform
X22269-013119

In creating an SDx platform project, developers select settings for the platform, the system configurations, and the domains. Platforms can contain multiple system configurations and domains as iterated and defined through the SDx IDE. Platform settings for prebuilt data allow inclusion of a hardware bitstream or its automatic generation to save on hardware build time when creating software solutions that do not require or create hardware accelerators. System configuration settings are available to either generate the needed platform software components or reference prebuilt software components. The type of software component Exectuable and Linkable Format (ELF) files, Boot Image Format (BIF) files, and Flattened Image Tree (FIT) image files are listed below. Domain settings provide choices for processor selection, operating system, and the runtime environment.

- Zynq® UltraScale+™ MPSoC
  - FSBL (ELF)
  - PMU (ELF)
  - BL31 (ELF)
  - U-BOOT (ELF) - for Linux target
  - LINUX image.ub (FIT) - for Linux target
  - BOOT file components (BIF)
- Zynq®-7000 SoC
  - FSBL (ELF)
  - U-BOOT (ELF) - for Linux target
  - LINUX image.ub (FIT) - for Linux target
  - BOOT file components (BIF)

Once generated, the file system layout of an SDSoC platform is structured as shown in the following figure.

Figure 3: **Directory Structure for a Typical SDSoC Platform**



In general, only the platform provider can ensure that a platform is "correct" for use within the SDSoC environment. However, the folder `<SDx_Install_Dir>/samples/platforms/Conformance` contains basic tests you should run, with instructions describing how to run them. These tests should build cleanly, and should be tested on the hardware platform.

***Note*:** <SDx_Install_Dir> signifies the location where the SDx tools are installed, including the released version: <SDx_Install_Dir> = <installation_path>/SDx/<*version*>.

A platform should provide tests for every custom interface so that users have examples of how to access these interfaces from application C/C++ code.

A platform may optionally include sample applications. By creating a `samples` sub-folder containing source files and a `description.json` file for each application, users can use the SDx IDE New Project wizard to select and build any of the provided sample applications. For additional information on application template creation, see Chapter 5: Sample Applications.

To create a platform using the SDx IDE, you can launch the application and select the Platform Project type.

IMPORTANT! *Before creating the platform project, you must have the DSA hardware definition as described in Chapter 3: Creating the Platform Hardware Component, and the software files as described in Chapter 4: Creating the Platform Software Component, available for use in defining the SDSoC platform.*

Using an SDx Workspace

IMPORTANT! *Linux host is strongly recommended for SDSoC™ platform development, and required for creating a platform supporting a target Linux OS.*

1. Launch the SDx™ IDE directly from the desktop icon or from the command line by one of the following methods:
    - Using either of the following commands from the command prompt:

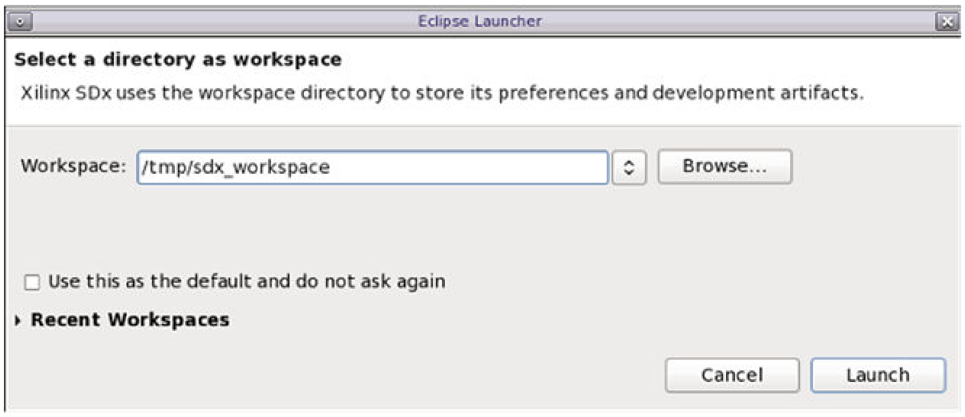
      ```
      sdx
      ```

      or

      ```
      sdx -workspace <workspace_name>
      ```

    - Double-clicking the **SDx** icon to start the program.
    - Launching from the **Start** menu in the Windows operating system.
2. The SDx IDE opens and prompts you to select a workspace, as shown in the following figure.

Figure 4: **Specify the SDx Workspace**



Eclipse Launcher

Select a directory as workspace

Xilinx SDx uses the workspace directory to store its preferences and development artifacts.

Workspace: /tmp/sdx_workspace

IMPORTANT! *When opening a new shell to enter an SDx command, ensure that you first source the* `settings64` *and* `setup` *scripts to set up the tool environment. On Windows, run the* `settings64.bat` *file from the command shell. See the SDSoC Environments Release Notes, Installation, and Licensing Guide (UG1294) for more information.*

The SDx workspace is the folder that stores your projects, source files, and results while working in the tool. You can define separate workspaces for each project or have workspaces for different types of projects. The following instructions show you how to define a workspace for an SDSoC project.

1. Click the **Browse** button to navigate to, and specify, the workspace, or type the appropriate path in the **Workspace** field.
2. Select the **Use this as the default and do not ask again** check box to set the specified workspace as your default choice and eliminate this dialog box in subsequent uses of SDx.
3. Click **Launch**.

TIP: *You can change the current workspace from within the SDx IDE by selecting* ***File→Switch Workspace****.*

You have now created an SDx workspace and can populate the workspace with projects. Platform and application projects are created to describe the SDx tool flow for creating an SDSoC platform.

The SDx IDE can populate the workspace with three types of user selected project types:

- Application Project
- Platform Project
- Library Project

The following sections describe how to use the Platform and Application project types while constructing the example SDSoC platforms.

Creating an SDSoC Platform Project

In this chapter, two platform projects are created using files from the SDx install tree. The first platform shows what files are needed and used in creating a platform for standalone use. The second platform illustrates the same flow but for the case of a platform that runs Linux on the target hardware. In general, a Linux and standalone system configuration can coexist in a single platform and they are not required to be in separate platform projects. Files sourced from the ZCU102 platform provided with the SDx tools are used to generate a new platform using the Platform Project flow. Subsequent chapters show how to create the hardware component of a platform with the Vivado® Design Suite and how to create the software components using the SDx IDE for standalone and Linux applications.

- Platform Assembly with Prebuilt Files

  Chapter 2: Creating SDSoC Platforms
- Building the Hardware

  Chapter 3: Creating the Platform Hardware Component
- Building the Software

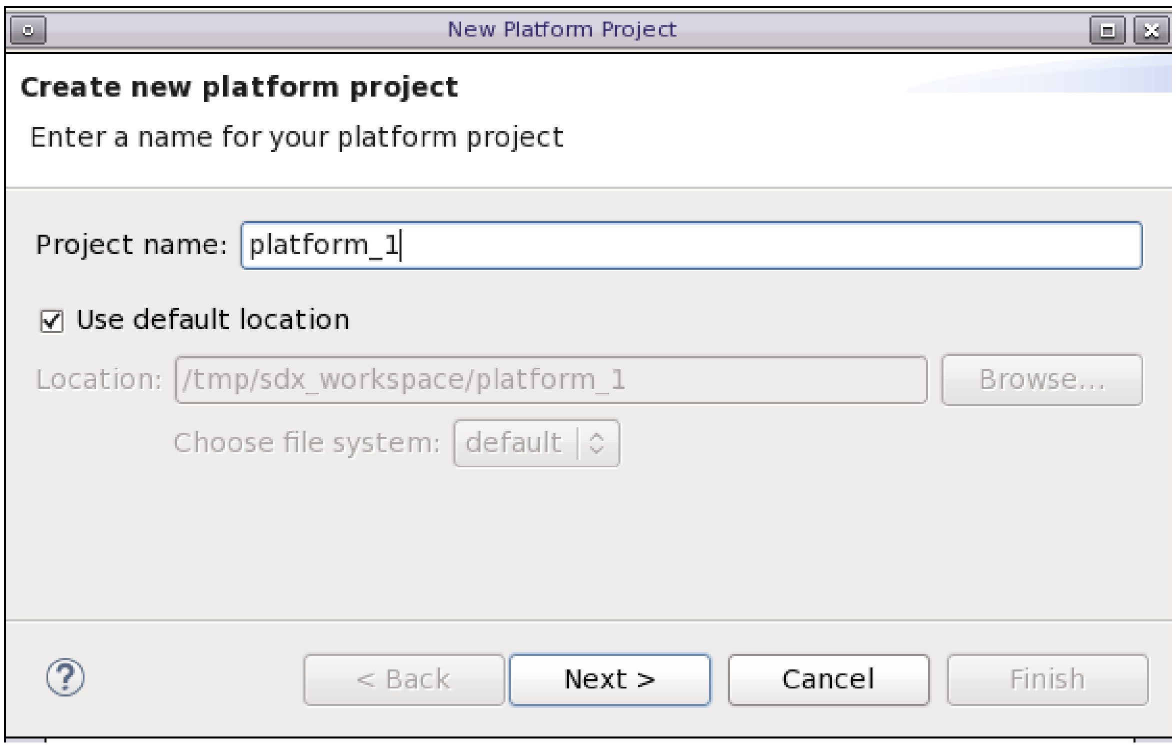
  Chapter 4: Creating the Platform Software Component

TIP: *There are sample platform files provided in the SDx tools software installation area at* `<SDx_Install_Dir>/platforms`.

Defining a Standalone Domain with Prebuilt Hardware and Software

After launching the SDx IDE, you can define a new SDSoC platform by creating a Platform Project using either the Welcome screen or the SDx menubar by selecting **File → New → SDx Platform Project**. The **New Platform Project** dialog opens and prompts you for a project name. Name the project `platform_1` for this ZCU102 based example.

Figure 5: **New Platform Project**


New Platform Project

Create new platform project

Enter a name for your platform project

Project name: platform_1

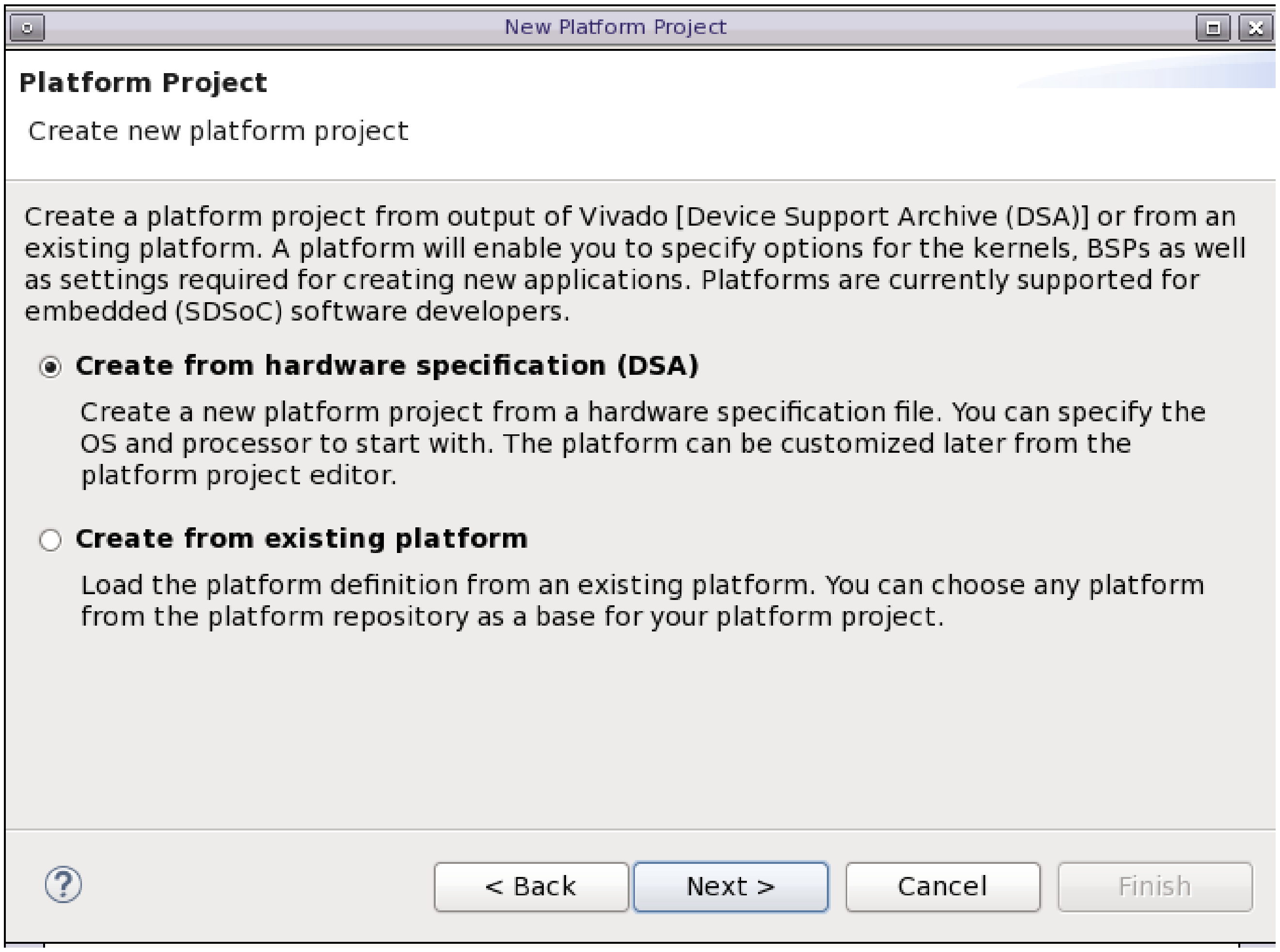
☑ Use default location

Location: /tmp/sdx_workspace/platform_1

Choose file system: default

Click **Next** to advance to the next dialog and select the source of the hardware specification for the platform. You can choose to use a DSA or an existing platform as the source for the hardware component of the platform. Select DSA for this example.

Figure 6: **Specify New Platform Source - DSA or Existing Platform**



New Platform Project

Platform Project

Create new platform project

Create a platform project from output of Vivado [Device Support Archive (DSA)] or from an existing platform. A platform will enable you to specify options for the kernels, BSPs as well as settings required for creating new applications. Platforms are currently supported for embedded (SDSoC) software developers.

[✓ **Create from hardware specification (DSA)**]

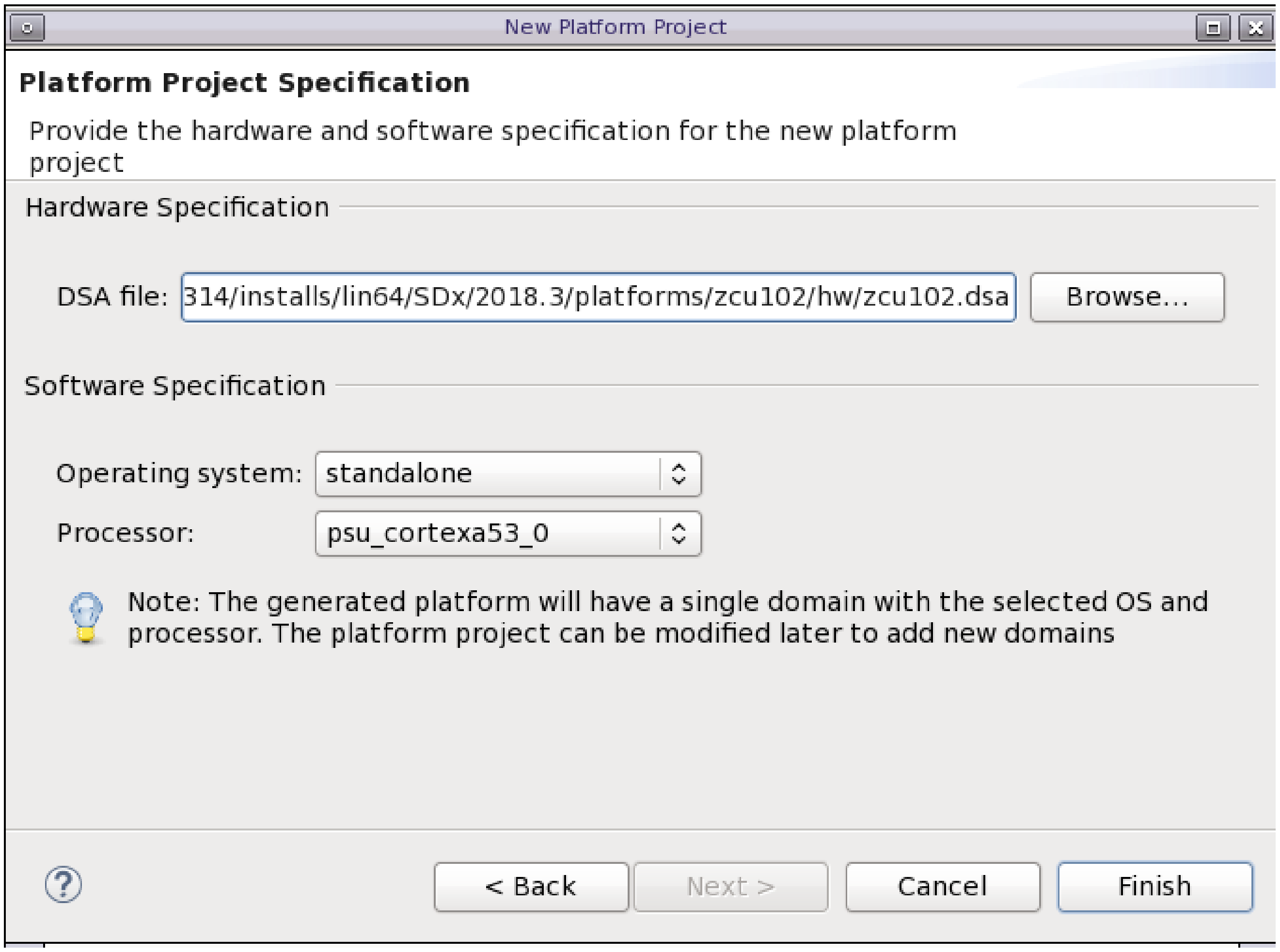
Create a new platform project from a hardware specification file. You can specify the OS and processor to start with. The platform can be customized later from the platform project editor.

Create from existing platform

Load the platform definition from an existing platform. You can choose any platform from the platform repository as a base for your platform project.

Click **Next** to specify the DSA as the **Hardware Specification**. Use the Browse button to locate the DSA for your project, or simply type the path to the DSA file in the field. The SDx tool includes platforms and DSA files that you can use as the foundation for creating your own SDSoC platforms, or you can create the DSA for a new platform using the Vivado Design Suite. Refer to Chapter 3: Creating the Platform Hardware Component for more information on creating the DSA. For this example, use the ZCU102 DSA located at `<SDx_Install_Dir>/platforms/zcu102/hw/zcu102.dsa`.

Figure 7: **Platform Hardware Specification**



Platform Project Specification

Provide the hardware and software specification for the new platform project

Hardware Specification

DSA file: 314/installs/lin64/SDx/2018.3/platforms/zcu102/hw/zcu102.dsa

Software Specification

Operating system: standalone

Processor: psu_cortexa53_0

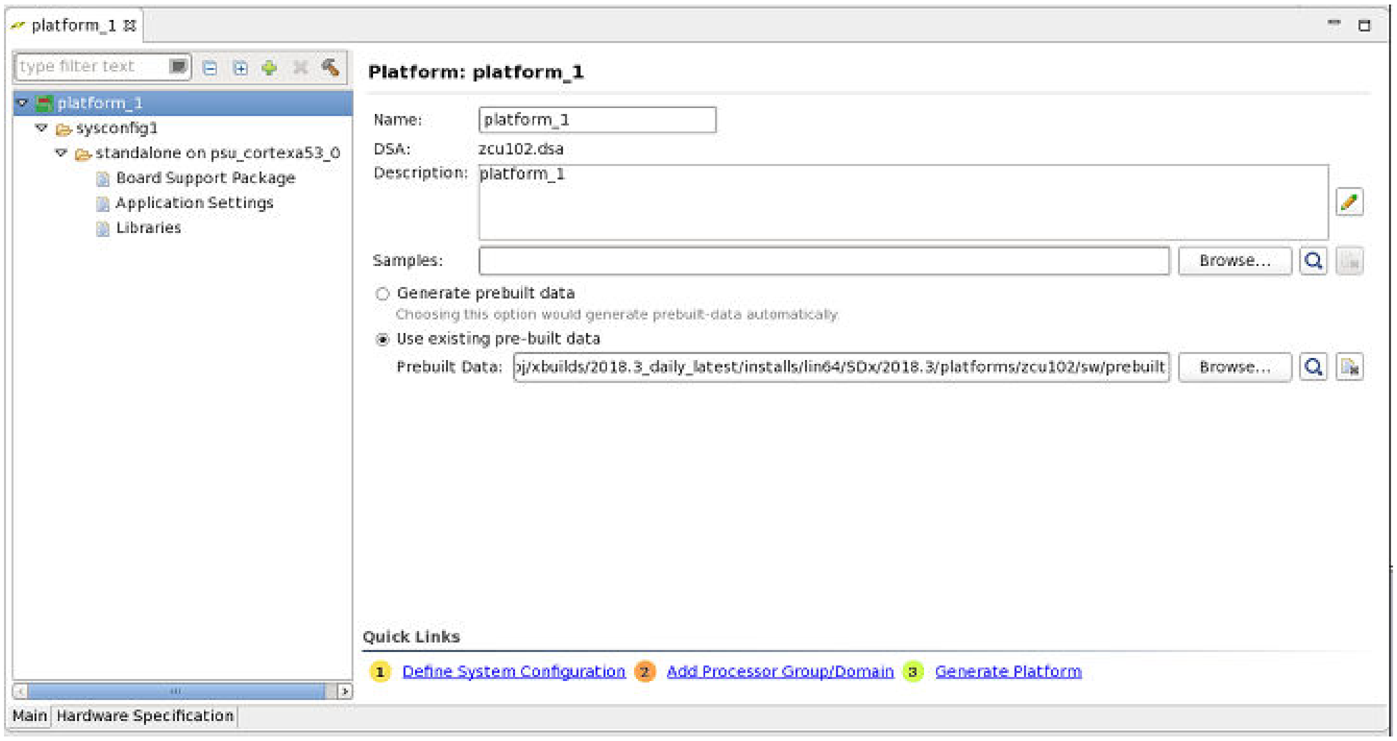
Note: The generated platform will have a single domain with the selected OS and processor. The platform project can be modified later to add new domains

Leave the Operating system and Processor settings as **standalone** and **psu_cortexa53_0**.

Click **Finish** to create the Platform project.

The platform configuration settings opens in the Editor Area of the SDx IDE. For this example, use the prebuilt hardware components and software components for the ZCU102 platform provided with the SDx tools at `<SDx_Install_Dir>/platforms/zcu102`. Select **Use existing pre-built data** on the platform configuration settings view and browse to or set the **Prebuilt Data** path to `<SDx_Install_Dir>/platforms/zcu102/sw/prebuilt`.

Figure 8: **Platform Configuration Settings**



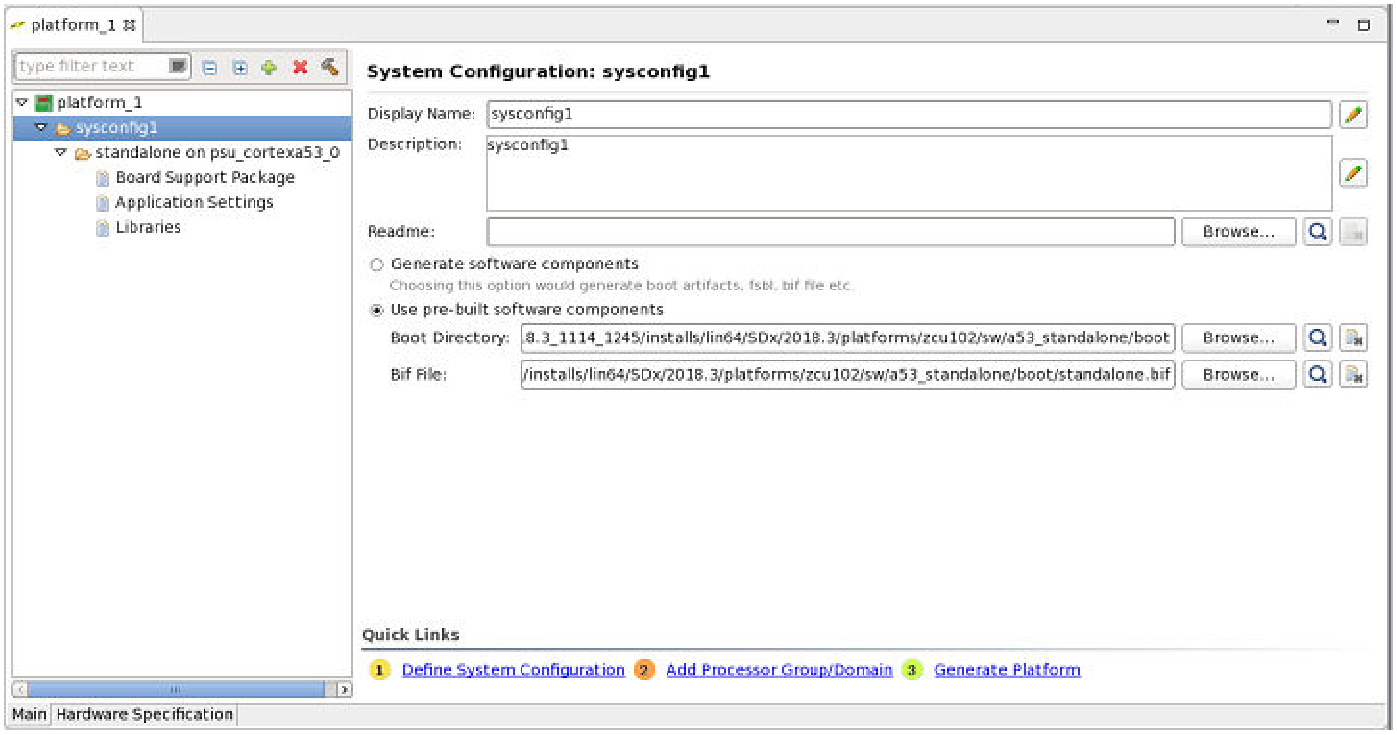
Platform: platform_1

Name: platform_1
DSA: zcu102.dsa
Description: platform_1
Samples:
( ) Generate prebuilt data
Choosing this option would generate prebuilt-data automatically
(●) Use existing pre-built data
Prebuilt Data: pj/xbuilds/2018.3_daily_latest/installs/lin64/SDx/2018.3/platforms/zcu102/sw/prebuilt

Providing a platform with prebuilt data containing software files with port interface specifications and a bitstream allows developers to quickly compile and run software applications that do not invoke hardware accelerated functions.

Click on **sysconfig1** for the System Configuration settings view. Select **Use pre-built software components** and browse to or set the **Boot Directory** path to `<SDx_Install_Dir>/platforms/zcu102/sw/a53_standalone/boot`. Browse to or set the **BIF File** to `<SDx_Install_Dir>/platforms/zcu102/sw/a53_standalone/boot/standalone.bif`.

Figure 9: **System Configuration Settings**



System Configuration: sysconfig1

Display Name: sysconfig1

Description: sysconfig1

Readme:

○ Generate software components

Choosing this option would generate boot artifacts, fsbl, bif file etc.

● Use pre-built software components

Boot Directory: 8.3_1114_1245/installs/lin64/SDx/2018.3/platforms/zcu102/sw/a53_standalone/boot

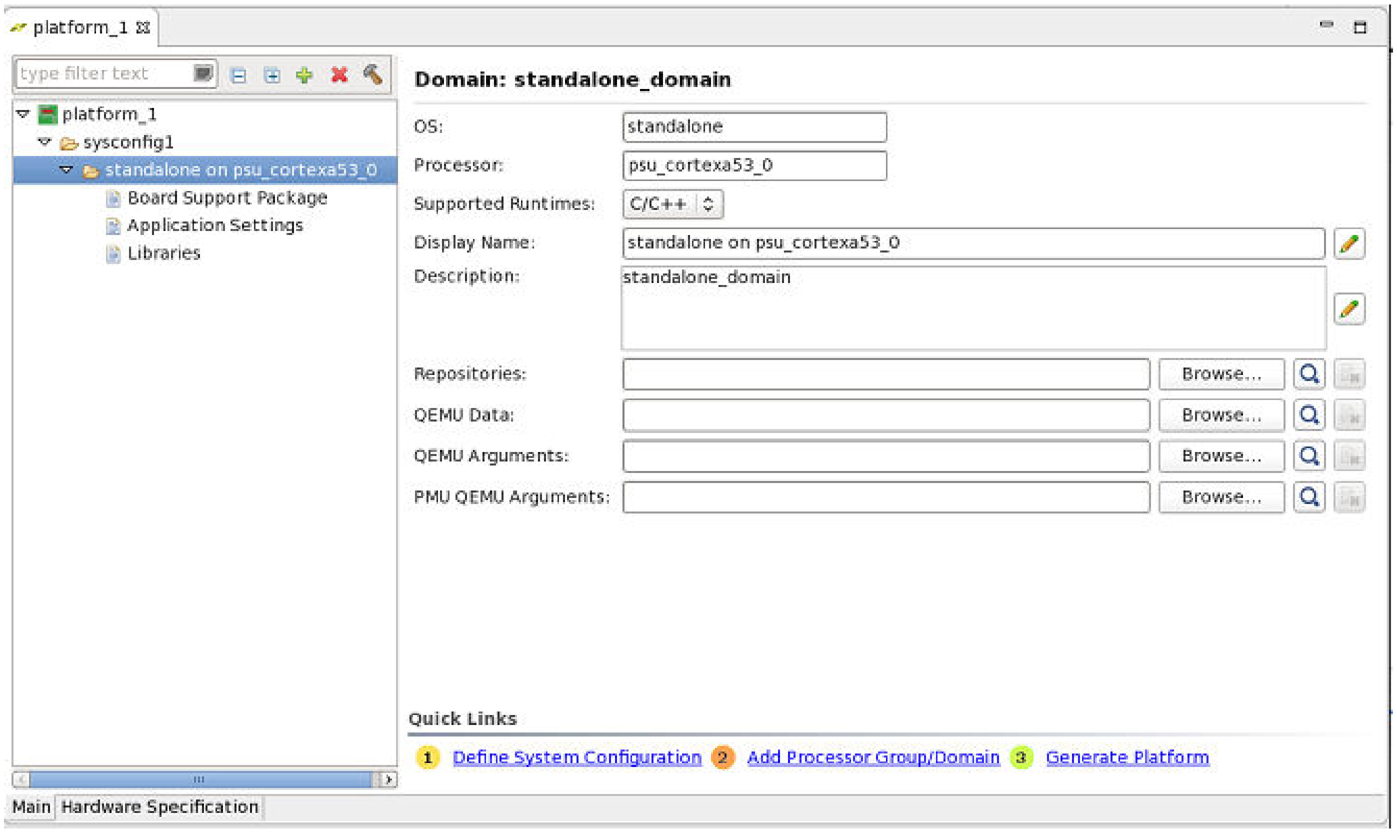
Bif File: /installs/lin64/SDx/2018.3/platforms/zcu102/sw/a53_standalone/boot/standalone.bif

Quick Links

1 Define System Configuration 2 Add Processor Group/Domain 3 Generate Platform

Click on **standalone on psu_cortexa53_0** for the Domain settings view. Do not make any changes to the domain settings for this example, but note that this view is used to change settings for the board support package, the application linker script, and included libraries.

Figure 10: **Domain Configuration Settings**



Domain: standalone_domain

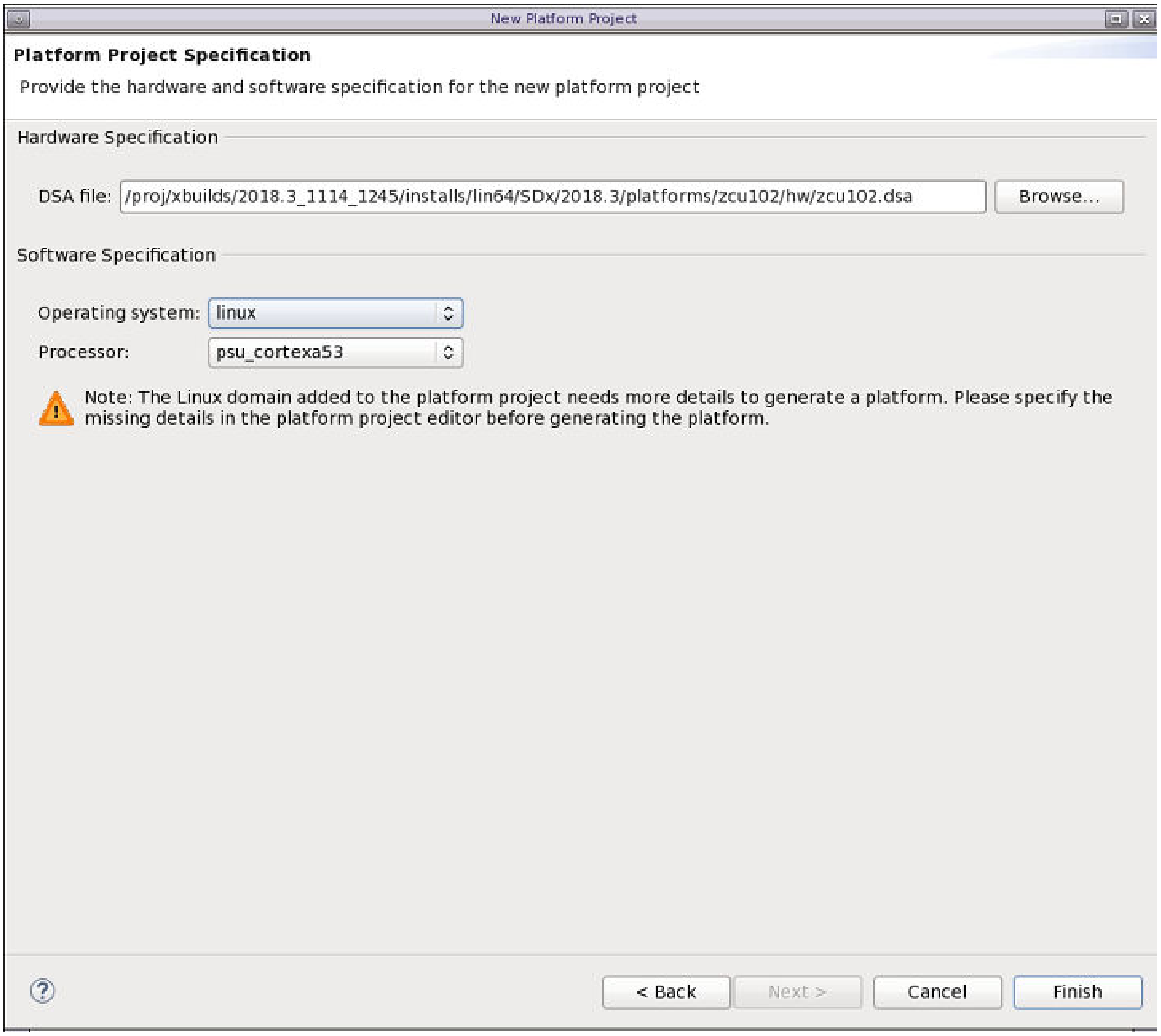
OS:	standalone
Processor:	psu_cortexa53_0
Supported Runtimes:	C/C++
Display Name:	standalone on psu_cortexa53_0
Description:	standalone_domain
Repositories:	
QEMU Data:	
QEMU Arguments:	
PMU QEMU Arguments:	

At this point, set up references to all the files necessary to create a standalone platform for the ZCU102. Click on **Quick Links - #3 Generate Platform** to complete the platform generation process. After the platform is generated, build SDx applications targeting the platform.

Defining a Linux Domain with Prebuilt Hardware and Software

As an example of creating an SDSoC platform for Linux applications, use the source files from the ZCU102 platform again and create a second platform, `platform_2` in the same workspace as `platform_1`. Begin by creating a new platform project using the menu bar **File → New → SDx Platform Project** and use the same ZCU102 DSA as before at `<SDx_Install_Dir>/platforms/zcu102/hw/zcu102.dsa`. Select **linux** for the **Operating system** and **psu_cortexa53** for the **Processor**.

Figure 11: **Platform Hardware Specification**



New Platform Project

Platform Project Specification

Provide the hardware and software specification for the new platform project

Hardware Specification

DSA file: /proj/xbuilds/2018.3_1114_1245/installs/lin64/SDx/2018.3/platforms/zcu102/hw/zcu102.dsa

Software Specification

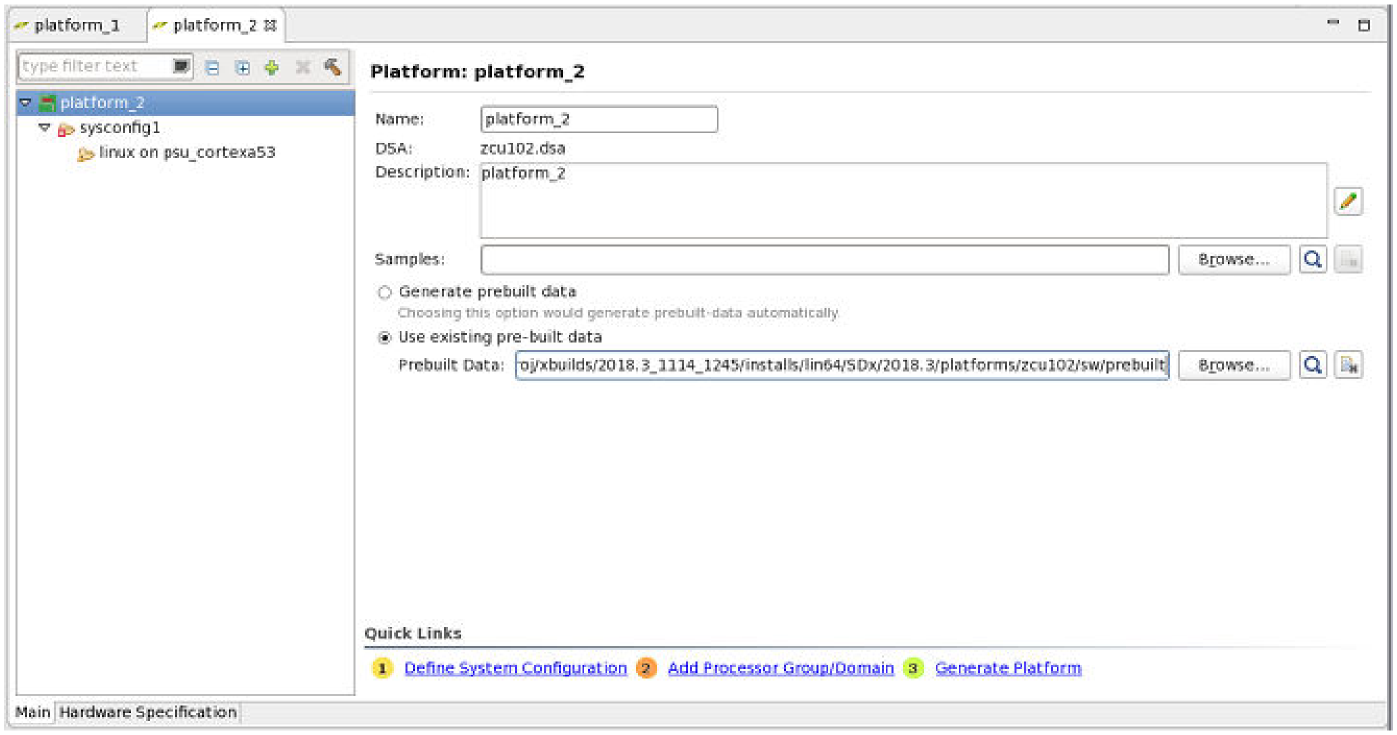
Operating system: linux

Processor: psu_cortexa53

Note: The Linux domain added to the platform project needs more details to generate a platform. Please specify the missing details in the platform project editor before generating the platform.

Select **Use existing pre-built data** on the platform configuration settings view and browse or set the **Prebuilt Data** path to `<SDx_Install_Dir>/platforms/zcu102/sw/prebuilt`.

Figure 12: **Platform Configuration Settings**

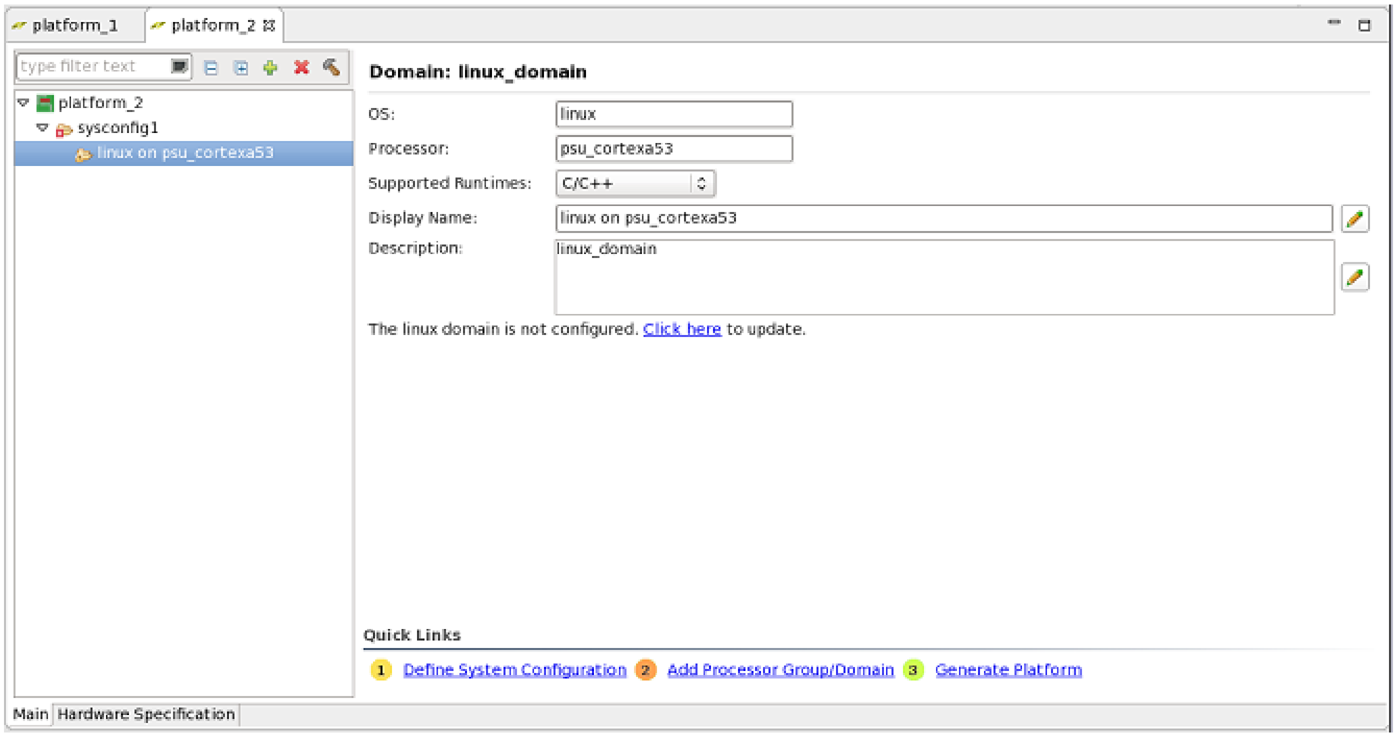


Platform: platform_2

Name: platform_2
DSA: zcu102.dsa
Description: platform_2
Samples:
Generate prebuilt data
Choosing this option would generate prebuilt-data automatically
[✓ Use existing pre-built data]
Prebuilt Data: oj/xbuilds/2018.3_1114_1245/installs/lin64/SDx/2018.3/platforms/zcu102/sw/prebuilt

Select **linux on psu_cortexa53** to view the Linux domain settings and click on the **Click here** link within the dialog's **The linux domain is not configured. Click here to update** prompt.

Figure 13: **Linux Domain Settings**



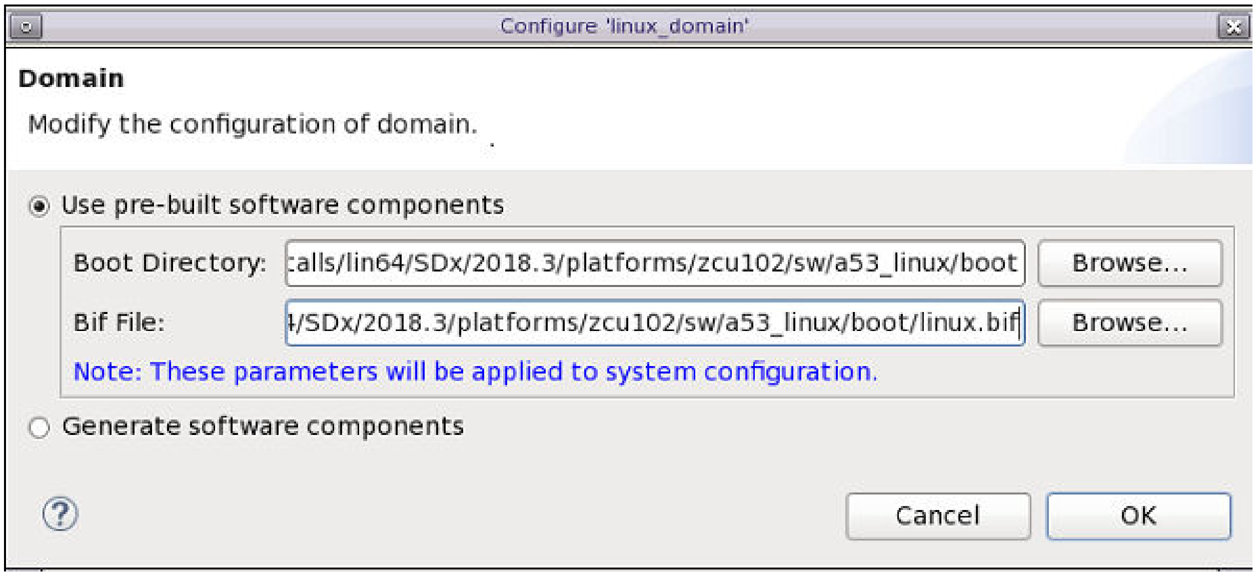
Domain: linux_domain

OS: linux
Processor: psu_cortexa53
Supported Runtimes: C/C++
Display Name: linux on psu_cortexa53
Description: linux_domain

The linux domain is not configured. Click here to update.

In the domain configuration dialog, select **Use pre-built software components** and browse to or set the **Boot Directory** path to `<SDx_Install_Dir>/platforms/zcu102/sw/a53_linux/boot`. Browse to or set the **BIF File** to `<SDx_Install_Dir>/platforms/zcu102/sw/a53_linux/boot/linux.bif`.

Figure 14: **Linux Boot and BIF Files**



Domain

Modify the configuration of domain.

(●) Use pre-built software components

Boot Directory: :alls/lin64/SDx/2018.3/platforms/zcu102/sw/a53_linux/boot

Bif File: 4/SDx/2018.3/platforms/zcu102/sw/a53_linux/boot/linux.bif

Note: These parameters will be applied to system configuration.

( ) Generate software components

The linux_domain configuration settings view now shows additional fields that can be populated to reference the following items:

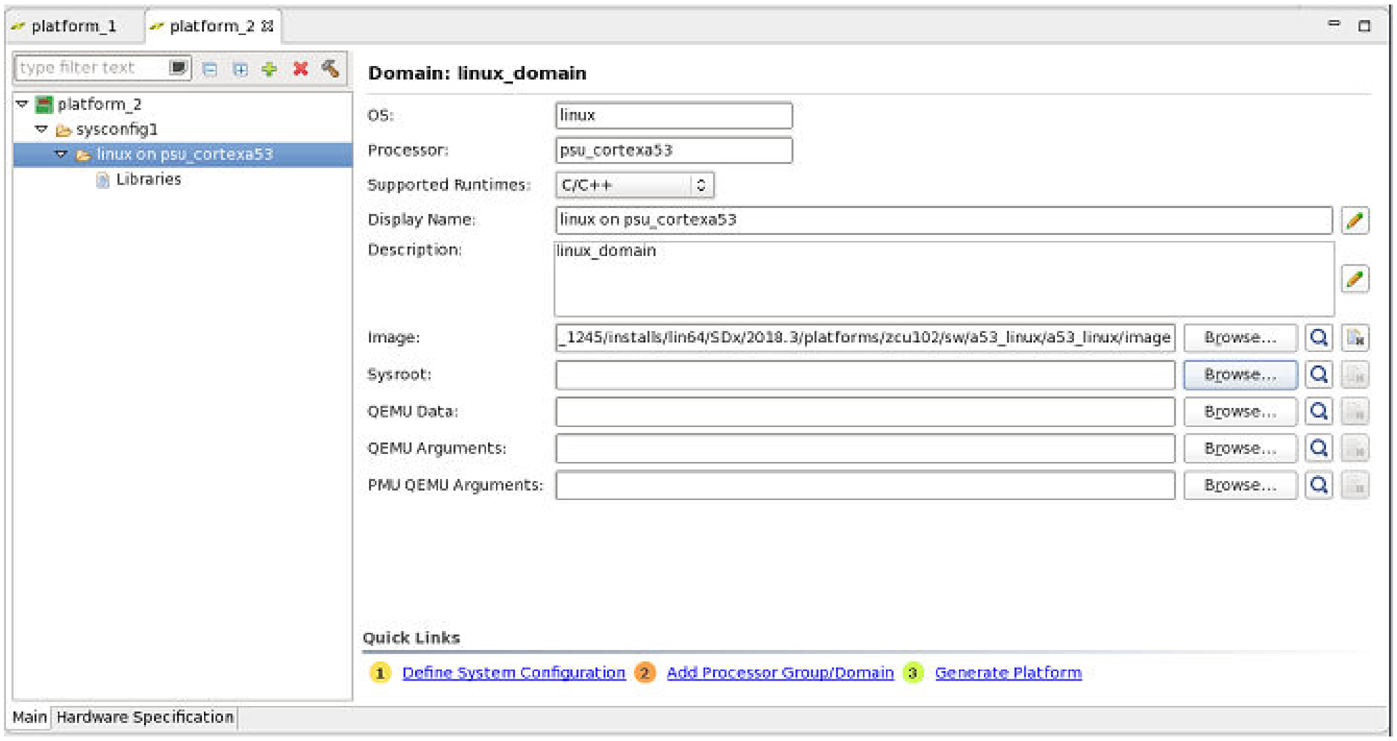
- Linux Image Directory
- Sysroot

The exisiting Linux `image.ub` file from the ZCU102 platform is referenced through the specified `image` directory. Optionally, a sysroot is associated with a platform by providing a reference within this Domain settings dialog.

Platforms with a sysroot enables the SDx IDE to create Linux application projects with make files containing default sysroot options for specifying include paths, library paths, and other options.

In the **linux_domain** view, browse to the Linux image at `<SDx_Install_Dir>/platforms/zcu102/sw/a53_linux/a53_linux/image`.

Figure 15: **Configured Linux Domain Settings**



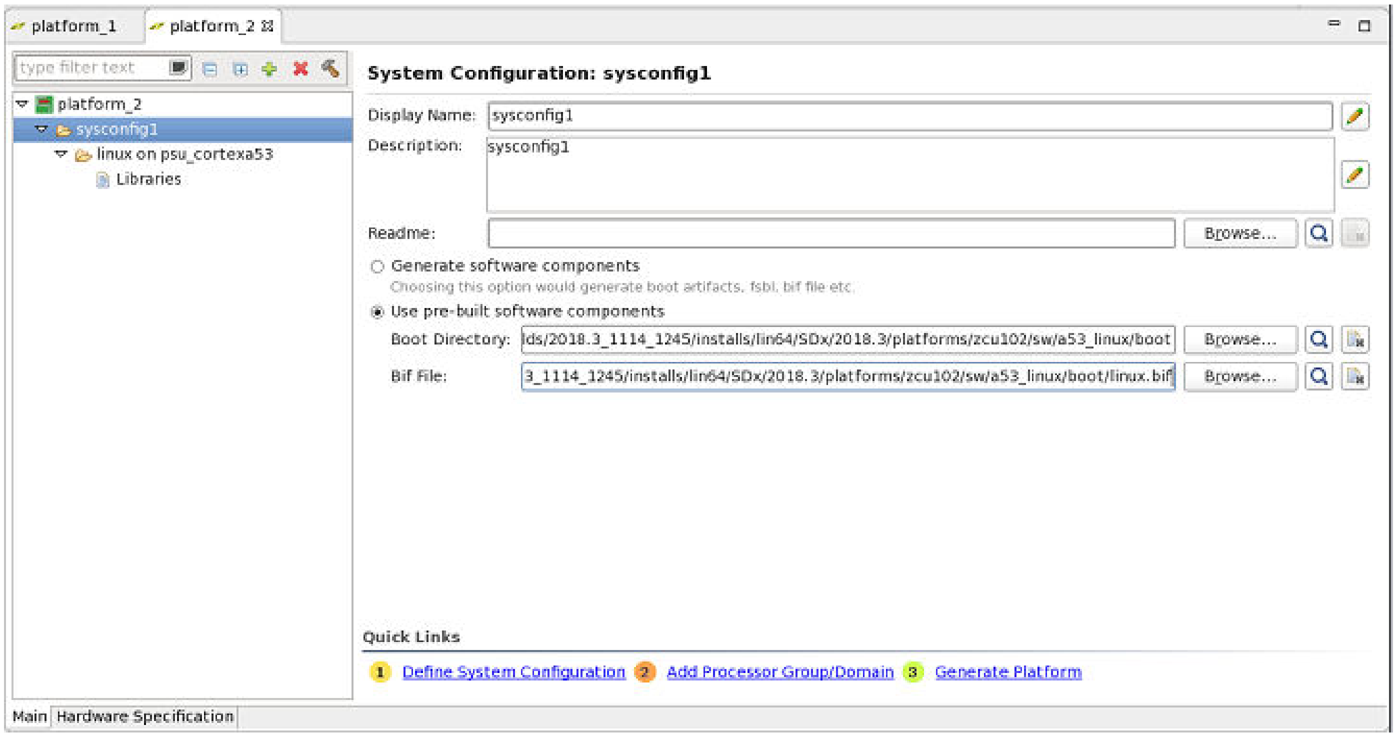
Domain: linux_domain

OS:	linux
Processor:	psu_cortexa53
Supported Runtimes:	C/C++
Display Name:	linux on psu_cortexa53
Description:	linux_domain
Image:	_1245/installs/lin64/SDx/2018.3/platforms/zcu102/sw/a53_linux/a53_linux/image
Sysroot:	
QEMU Data:	
QEMU Arguments:	
PMU QEMU Arguments:	

1 Define System Configuration 2 Add Processor Group/Domain 3 Generate Platform

The **sysconfig1** system configuration settings are automatically populated by the Linux domain settings entered for the Boot files directory and the BIF file are in place.

Figure 16: **Linux System Configuration Settings**



System Configuration: sysconfig1

Display Name: sysconfig1

Description: sysconfig1

Readme:

Generate software components

Choosing this option would generate boot artifacts, fsbl, bif file etc.

[✓ Use pre-built software components]

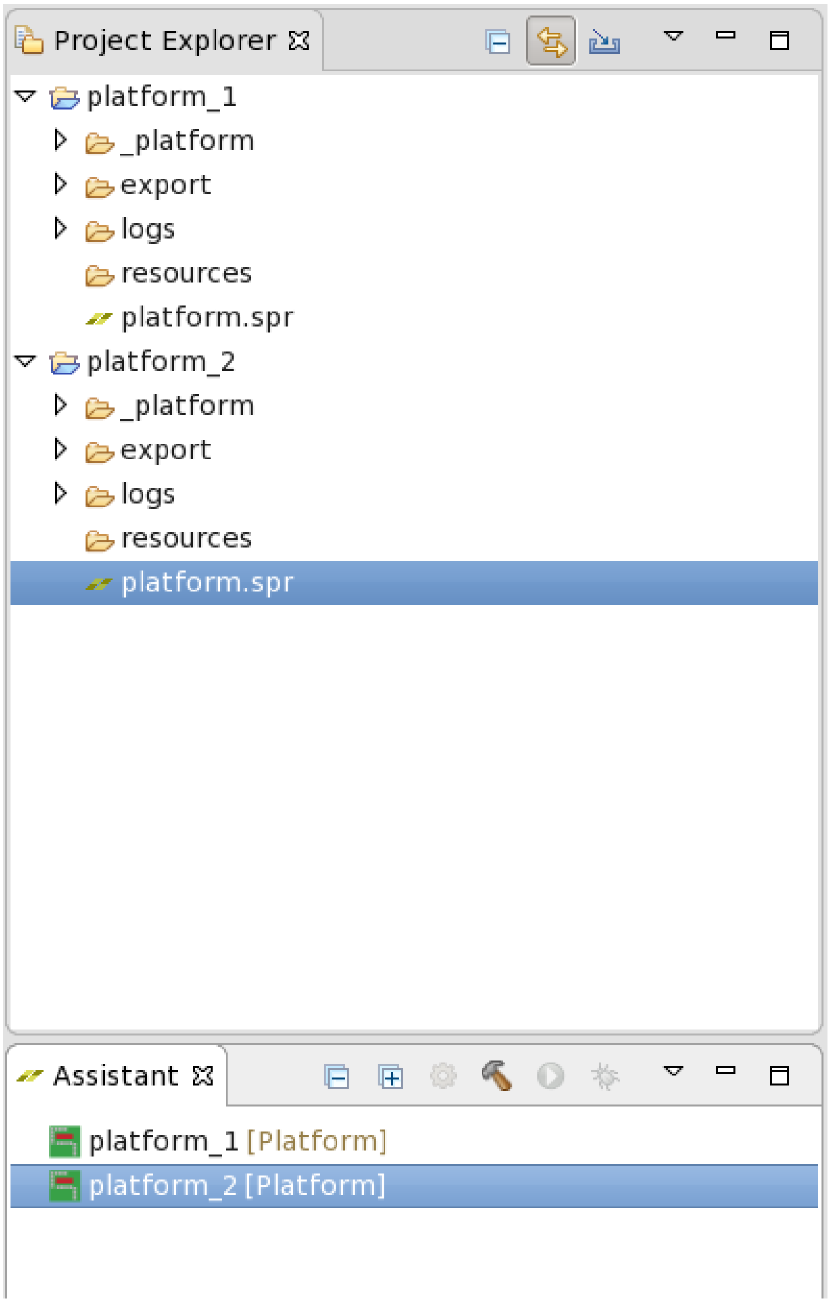
Boot Directory: lds/2018.3_1114_1245/installs/lin64/SDx/2018.3/platforms/zcu102/sw/a53_linux/boot

Bif File: 3_1114_1245/installs/lin64/SDx/2018.3/platforms/zcu102/sw/a53_linux/boot/linux.bif

At this point, set up references to all the files necessary to create a Linux platform for the ZCU102. Click on **Quick Links - #3 Generate Platform** to complete the platform generation process. After the platform is generated, build the SDx applications targeting the platform.

The SDx **Project Explorer** and **Assistant** views after platform generation completes is shown below.

Figure 17: **Generated Platforms**



Defining an SDSoC Application Project for Platform Testing

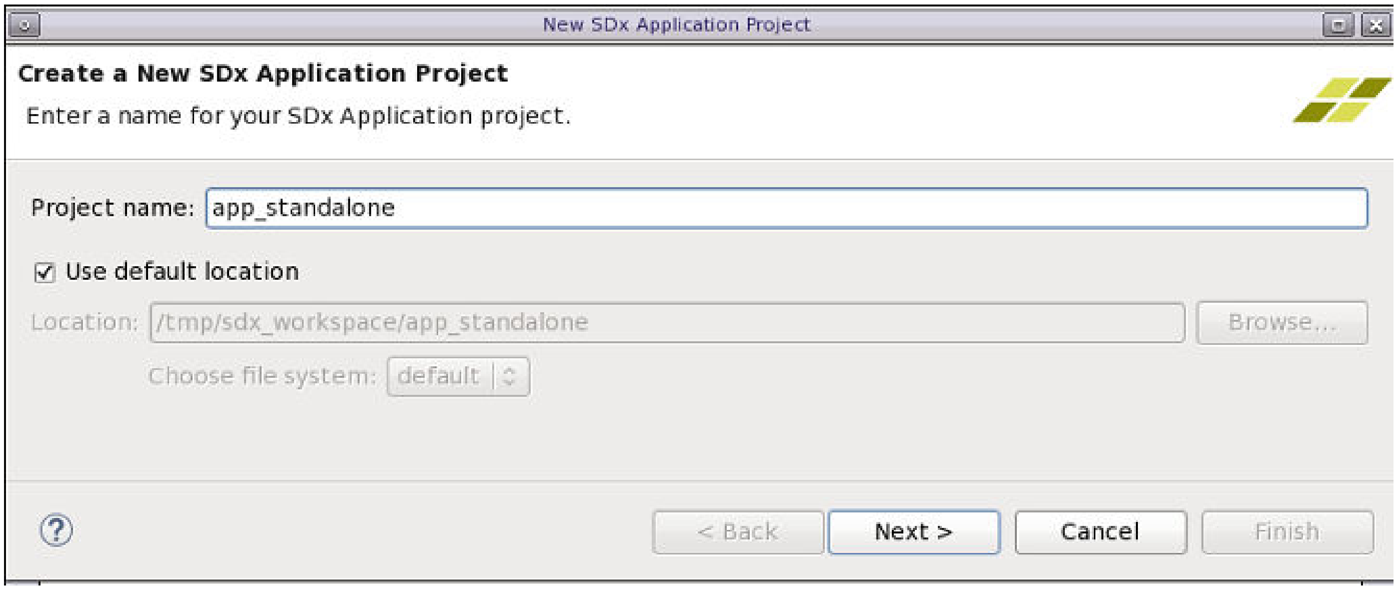
The SDx IDE provides compilation tools and example code templates for creating applications that run on SDSoC platforms. Create example applications for the standalone (`platform_1`) and Linux (`platform_2`) platforms using the **Array zero_copy** code template provided with the SDx tools.

The Compilation log contains the commands issued to build the application and is useful for observing the sequence of actions taken during the build process. The Data Motion Network report lists the accelerated functions and how their arguments were mapped and connected to platform interfaces. The type of data mover used for each argument is also listed.

Reuse the SDx workspace (`sdx_workspace`) containing the standalone and Linux platforms created in the section above.

Create a new SDx Application project using the menu bar and selecting **File → New → SDx Application Project**. The **New SDx Application Project** dialog opens and prompts you for a project name. Name the project `app_standalone` for this ZCU102-based example.

Figure 18: **New SDx Application Project**



New SDx Application Project

Create a New SDx Application Project

Enter a name for your SDx Application project.

Project name: app_standalone

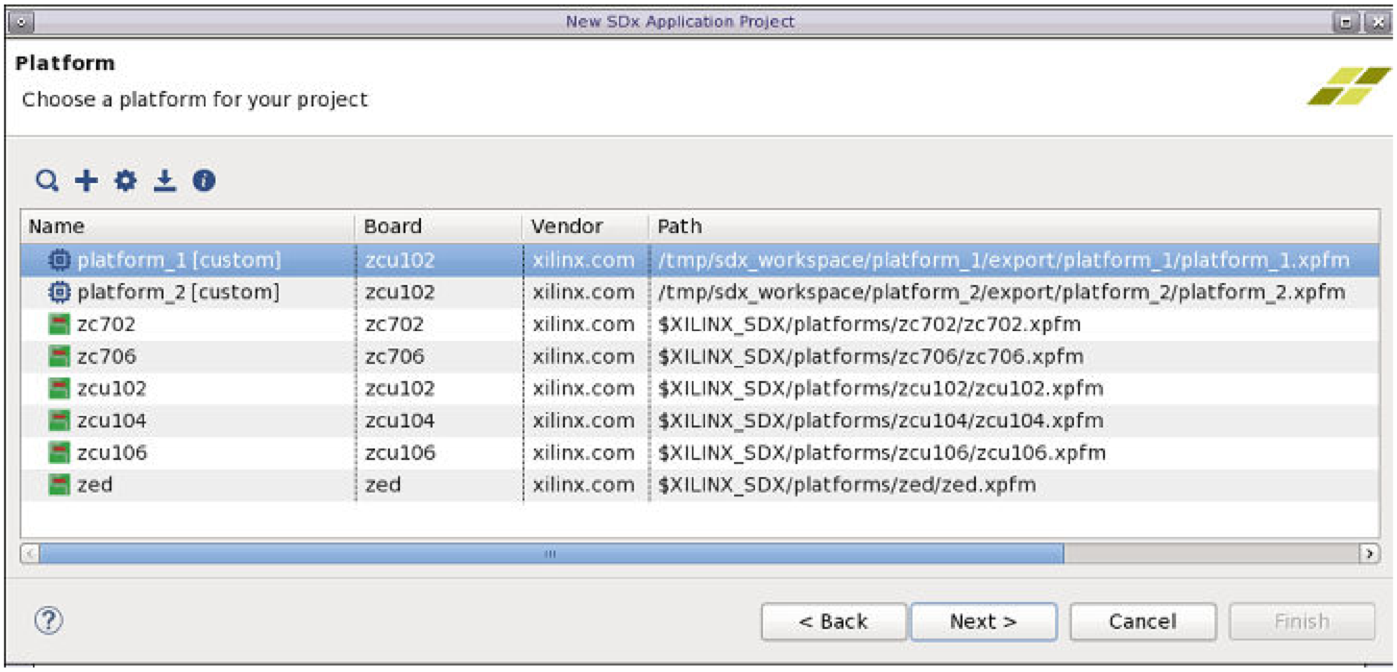
☑ Use default location

Location: /tmp/sdx_workspace/app_standalone

Choose file system: default

Click **Next** to advance to the **Platform** selection dialog. Select `platform_1 [custom]` as this platform was configured for the standalone domain when it was created.

Figure 19: **Application Platform Selection**



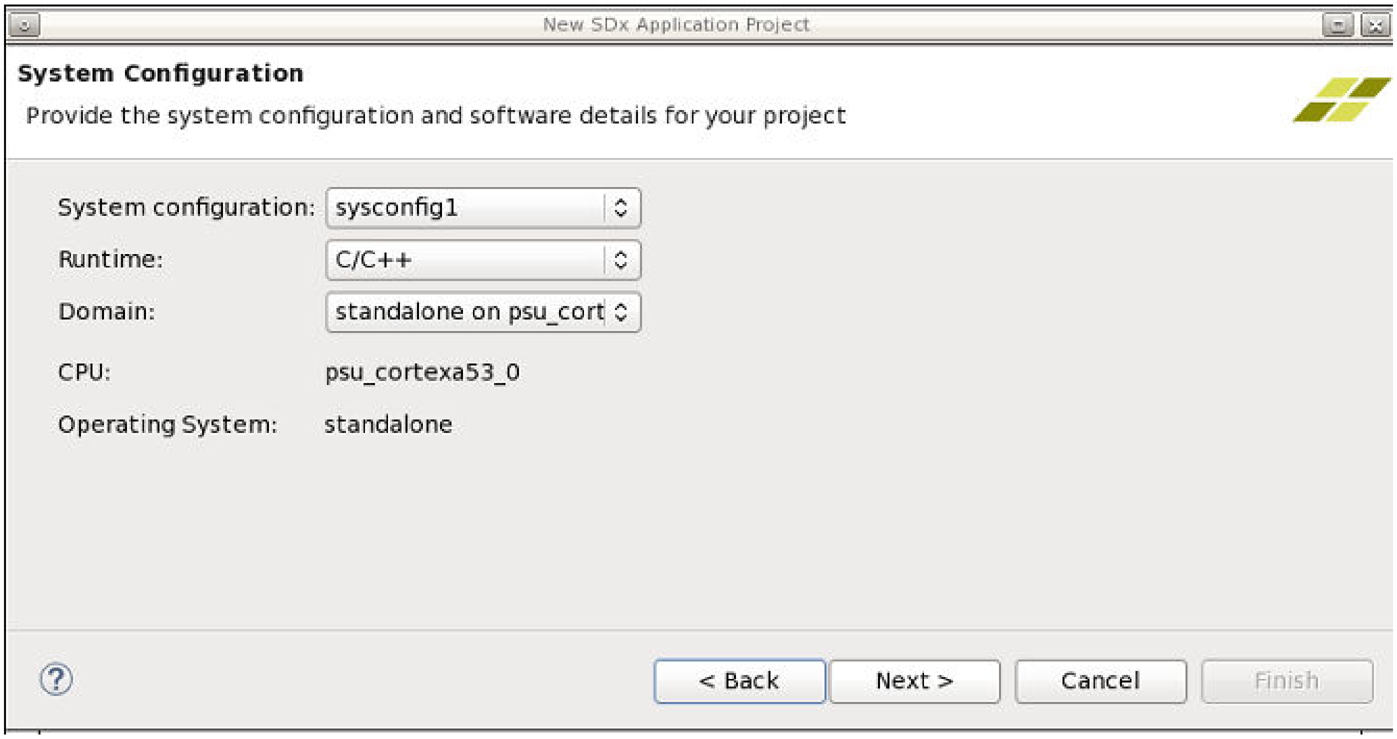
Platform

Choose a platform for your project

Name	Board	Vendor	Path
platform_1 [custom]	zcu102	xilinx.com	/tmp/sdx_workspace/platform_1/export/platform_1/platform_1.xpfm
platform_2 [custom]	zcu102	xilinx.com	/tmp/sdx_workspace/platform_2/export/platform_2/platform_2.xpfm
zc702	zc702	xilinx.com	$XILINX_SDX/platforms/zc702/zc702.xpfm
zc706	zc706	xilinx.com	$XILINX_SDX/platforms/zc706/zc706.xpfm
zcu102	zcu102	xilinx.com	$XILINX_SDX/platforms/zcu102/zcu102.xpfm
zcu104	zcu104	xilinx.com	$XILINX_SDX/platforms/zcu104/zcu104.xpfm
zcu106	zcu106	xilinx.com	$XILINX_SDX/platforms/zcu106/zcu106.xpfm
zed	zed	xilinx.com	$XILINX_SDX/platforms/zed/zed.xpfm

Click **Next** to advance to the **System Configuration** dialog. Leave the settings at their default values for **System configuration:** `sysconfig1`, **Runtime:** `C/C++`, and **Domain:** `standalone on psu_cortexa53_0`. Recall that when `platform_1` was created it was configured with a single system configuration containing a single standalone domain. If additional system configurations or domains are present in a platform an application can be customized to use them through this **System Configuration** dialog.

Figure 20: **Application System Configuration**



System Configuration

Provide the system configuration and software details for your project

System configuration: sysconfig1

Runtime: C/C++

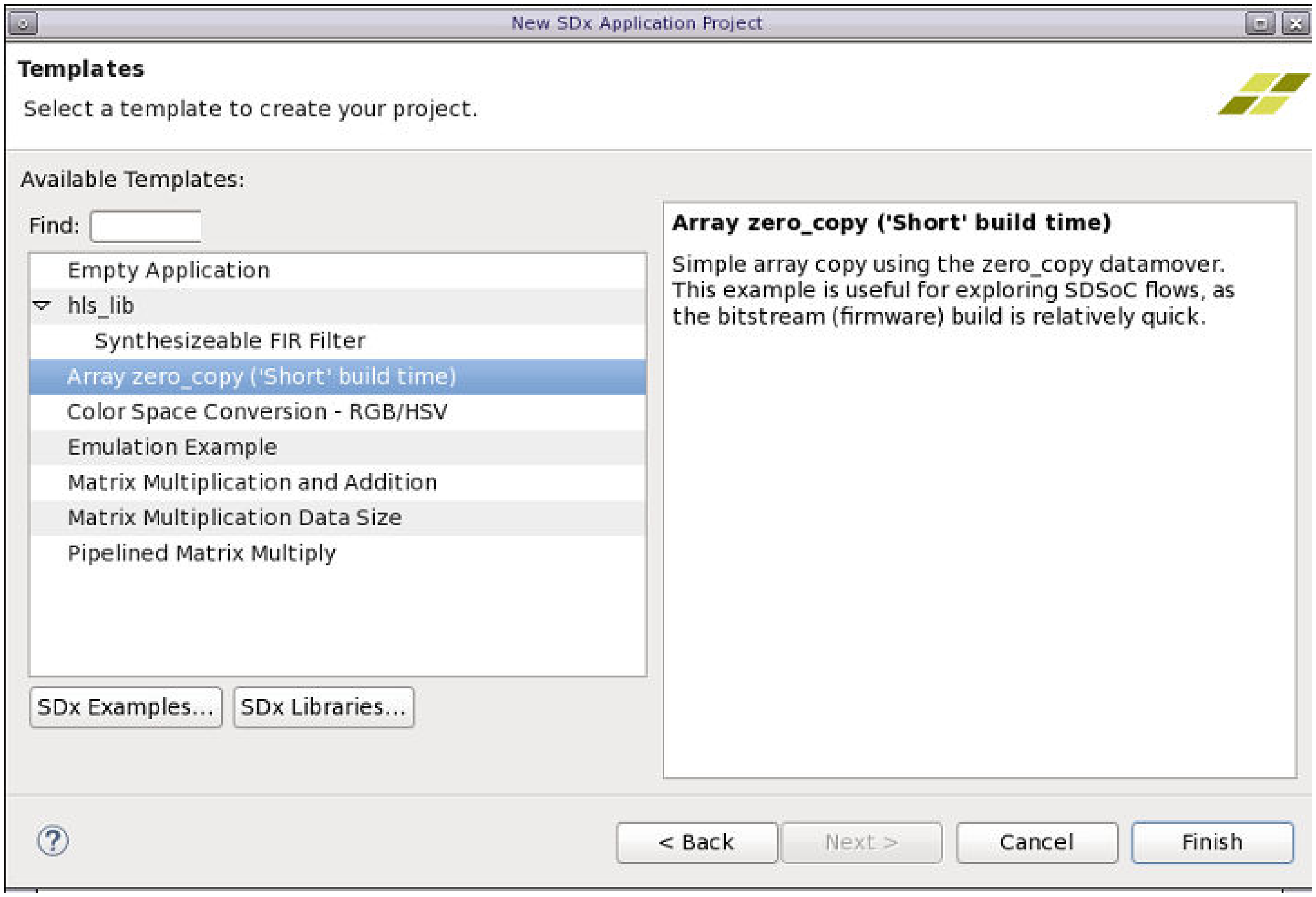
Domain: standalone on psu_cort

CPU: psu_cortexa53_0

Operating System: standalone

Click **Next** to advance to the **Templates** dialog and select the **Array zero_copy** example as the code base for the application.

Figure 21: **Application Template**



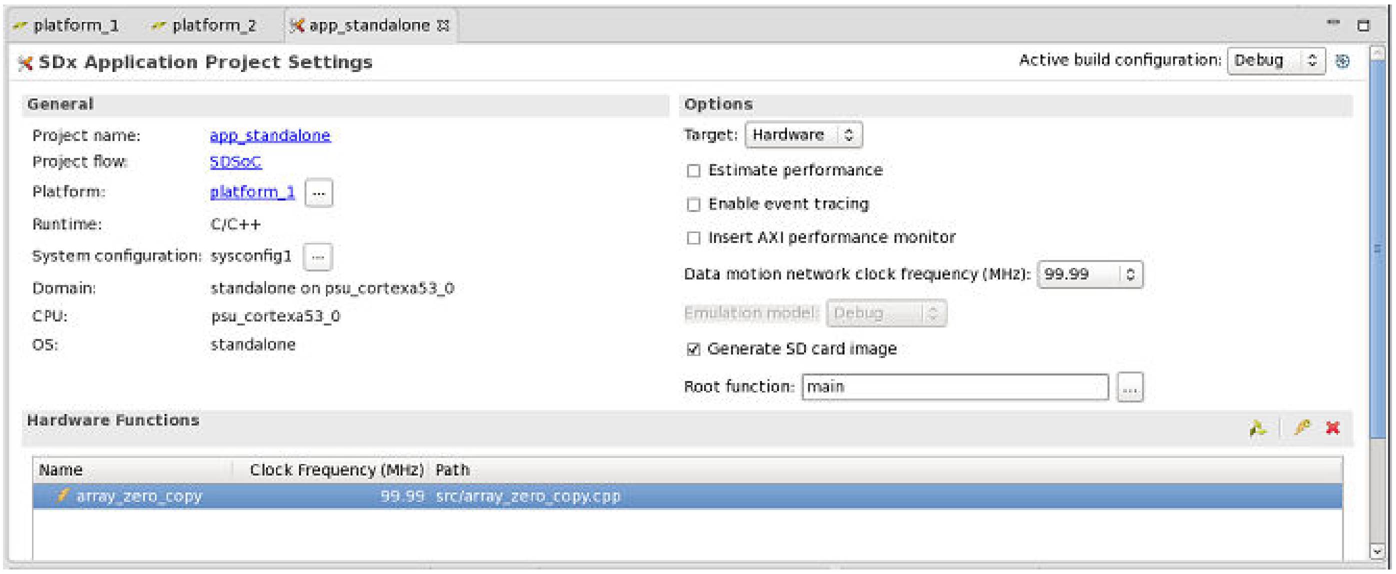
Empty Application
hls_lib
Synthesizeable FIR Filter
Array zero_copy ('Short' build time)
Color Space Conversion - RGB/HSV
Emulation Example
Matrix Multiplication and Addition
Matrix Multiplication Data Size
Pipelined Matrix Multiply

Array zero_copy ('Short' build time)

Simple array copy using the zero_copy datamover. This example is useful for exploring SDSoC flows, as the bitstream (firmware) build is relatively quick.

Click **Finish** to add the template code to the SDx application. The **Editor Area** now shows the application project settings including the functions that have been selected for hardware acceleration. The **Project Explorer** and **Assistant** views are updated with the new `app_standalone` application.

Figure 22: **Application Project Settings**

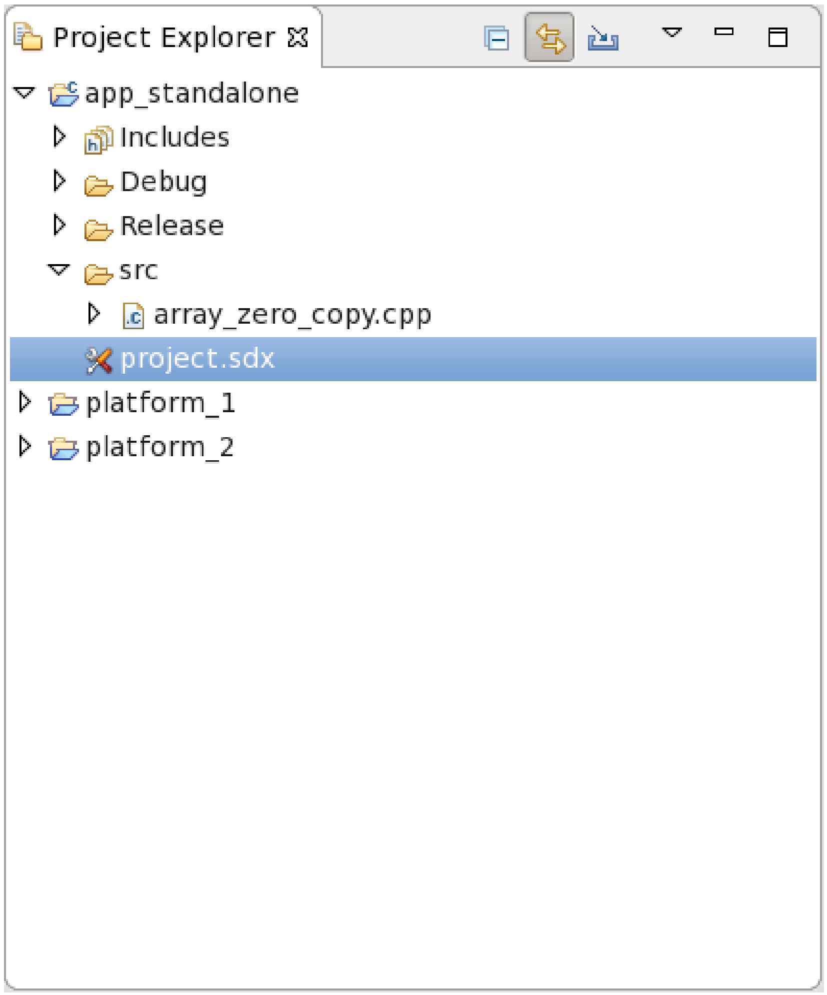


Project name:	app_standalone
Project flow:	SDSoC
Platform:	platform_1
Runtime:	C/C++
System configuration:	sysconfig1
Domain:	standalone on psu_cortexa53_0
CPU:	psu_cortexa53_0
OS:	standalone

Target: Hardware
☐ Estimate performance
☐ Enable event tracing
☐ Insert AXI performance monitor
Data motion network clock frequency (MHz): 99.99
Emulation model: Debug
☑ Generate SD card image
Root function: main

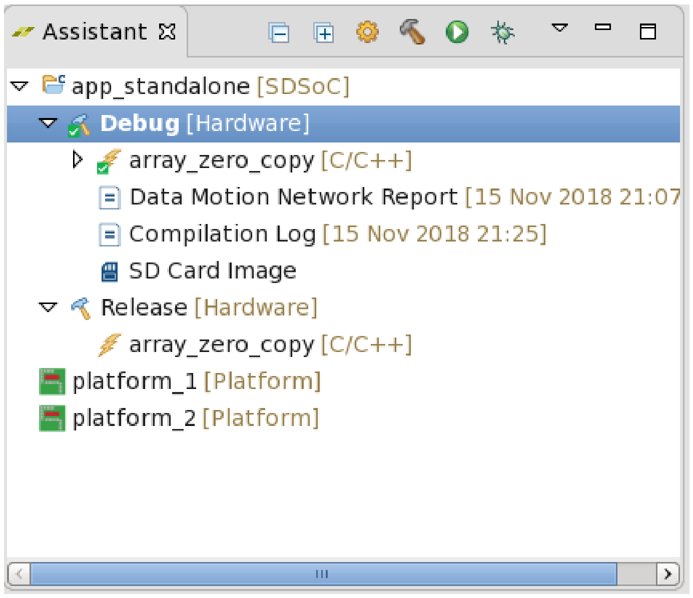
Name	Clock Frequency (MHz)	Path
array_zero_copy	99.99	src/array_zero_copy.cpp

Figure 23: **Project Explorer View**



Build the `app_standalone` application by expanding the **app_standalone [SDSoC]** listing in the **Assistant** view, right-click on **Debug [Hardware]** and select **Build**.

Figure 24: **Application Build Assistant View**



After the application builds the Data Motion Network Report and Compilation Log are available through the **Assistant** view. The Compilation log contains the commands issued to build the application and is useful for observing the sequence of actions taken during the build process. The Data Motion Network report lists the accelerated functions and how their arguments were mapped and connected to platform interfaces. The type of data mover used for each argument is also listed.

A set of files for booting and running the application on target hardware is generated and accessible by right-clicking on **SD Card Image** and selecting **Open > Open in Project Explorer**, **Open > Open in File Browser**, or **Open > Open in Terminal**. The files should be copied to a FAT32 formatted SD card and used to boot the target hardware, ZCU102 for this example, to test the generated platform by running application code.

The procedure for using the **Array zero_copy** template example for testing the Linux platform (`platform_2`) generated earlier is similar to the standalone flow. The workspace is reused again to add another SDx Application project, named `app_linux`, and use the Linux system configuration offered in `platform_2`. The **Project Explorer** and **Assistant** views after a successful build of the application are shown below. The contents of the `sd_card` directory are used to boot and run the Linux application on target hardware.

Figure 25: **Linux Application Project Explorer View**

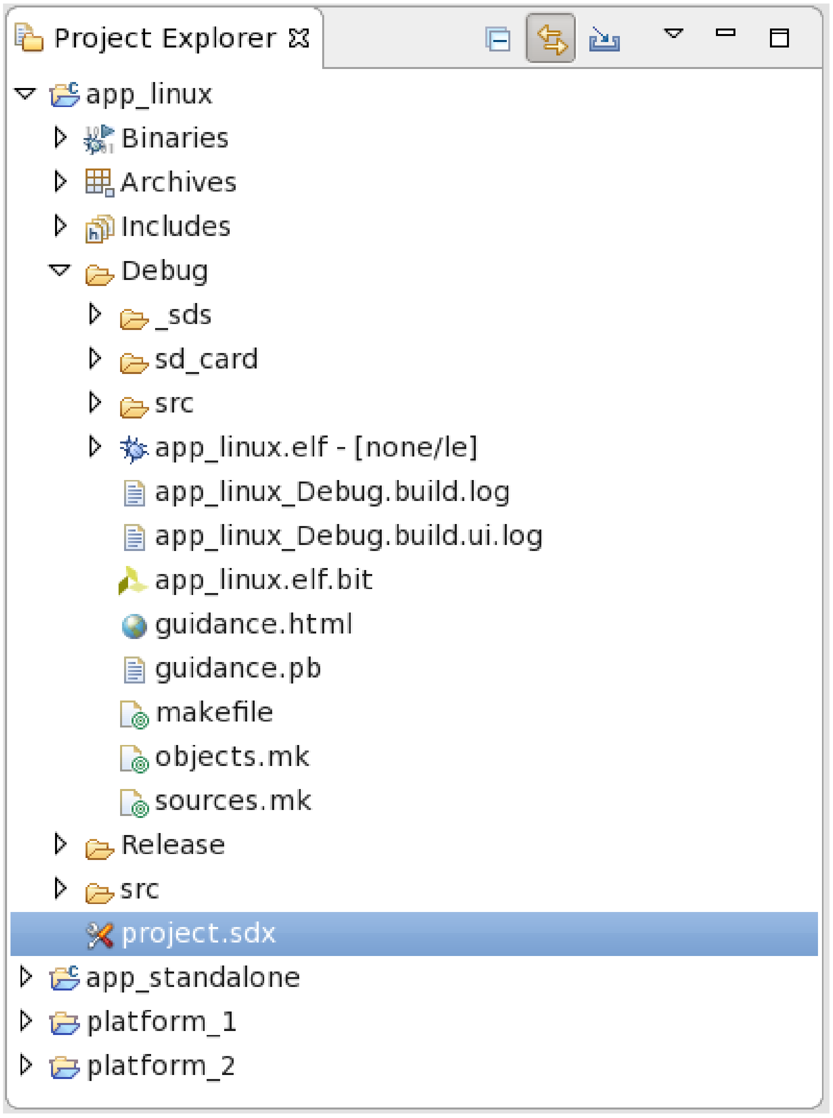
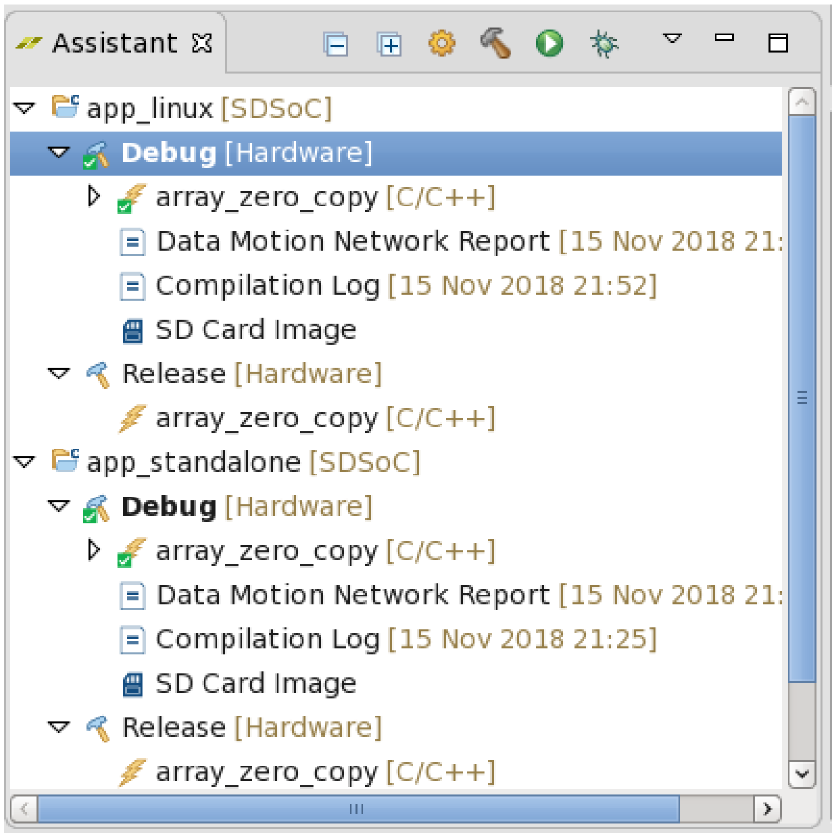


Figure 26: **Linux Application Build Assistant View**



```
Assistant
▽ app_linux [SDSoC]
  ▽ Debug [Hardware]
    ▷ array_zero_copy [C/C++]
      Data Motion Network Report [15 Nov 2018 21:
      Compilation Log [15 Nov 2018 21:52]
      SD Card Image
  ▽ Release [Hardware]
      array_zero_copy [C/C++]
▽ app_standalone [SDSoC]
  ▽ Debug [Hardware]
    ▷ array_zero_copy [C/C++]
      Data Motion Network Report [15 Nov 2018 21:
      Compilation Log [15 Nov 2018 21:25]
      SD Card Image
  ▽ Release [Hardware]
      array_zero_copy [C/C++]
```

Querying the Platform

The SDx environment provides tools to read and check the platform files you create. From within the SDx terminal window you can verify that the SDx IDE can correctly read the platform files created by the SDSoC platform project by executing the following command, from within the `workspace/project_name/export` where the generated platform is written:

```
> sds++ -sds-pf-list
```

This command lists the available SDx platforms by reading the platform folders in the current working directory, and reading the platforms in the SDx installation hierarchy. If you specify this command from a folder containing a custom platform it will read the platform found there.

Any platform listed by the previous command can be displayed in greater detail using the following command:

```
> sds++ -sds-pf-info <platform_name>
```

TIP: *For platforms that are not in the installation area, the `platform_name` is the path to the folder containing the `platform`.*

This command displays the details of the specified platform.

You can also specify the platform to use for a project using the following command:

```
> sds++ -sds-pf <platform_name>
```

The follow platform properties are reported after running the `sds++ -sds-pf-info zcu102` command:

```
Platform Information
====================
Name: zcu102

Device
------
 Architecture: zynquplus
       Device: xczu9eg
      Package: ffvb1156
  Speed grade: -2

System Clocks
-------------
  Clock ID   Frequency
 ----------|------------
           1199.880127
         0    74.992500
         1    99.990000
         2   149.985000
         3   199.980000
         4   299.970000
         5   399.960000
         6   599.940000

Platform:    zcu102 (<SDx_Install_Dir>/platforms/zcu102)

Description:
A basic platform targeting the ZCU102 evaluation board, which includes
4GB of DDR4 for the Processing System, 512MB of DDR4 for the Programmable
Logic, 2x64MB Quad-SPI Flash and an SDIO card interface. More information
at https://www.xilinx.com/products/boards-and-kits/ek-u1-zcu102-g.html

Available system configurations:
  a53_linux (a53_linux)
  a53_standalone (a53_standalone)
  r5_standalone (r5_standalone)

System Ports

Use the system port name in a sysport pragma, for example
#pragma SDS data sysport(parameter_name:system_port_name)

System Port Name (Vivado BD instance name, Vivado BD port name)
ps_e_S_AXI_HPC0_FPD (ps_e, S_AXI_HPC0_FPD)
```

```
ps_e_S_AXI_HPC1_FPD (ps_e, S_AXI_HPC1_FPD)
ps_e_S_AXI_HP0_FPD (ps_e, S_AXI_HP0_FPD)
ps_e_S_AXI_HP1_FPD (ps_e, S_AXI_HP1_FPD)
ps_e_S_AXI_HP2_FPD (ps_e, S_AXI_HP2_FPD)
ps_e_S_AXI_HP3_FPD (ps_e, S_AXI_HP3_FPD)
```

Refer to the *SDx Command and Utility Reference Guide* (UG1279) for more information.

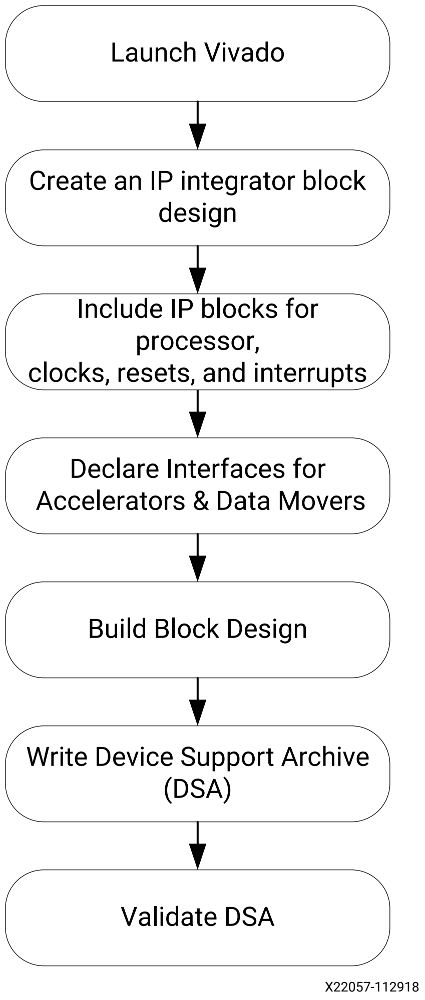
Creating the Platform Hardware Component

The Hardware Component of a Platform captures the logical and physical interfaces to the hardware functions accelerated through the SDx™ environment. The processor, memory, and all external board interfaces are configured using a combination of Vivado® IP, user custom IP, and RTL. This provides a logic "wrapper" for the hardware functions to be executed properly on the platform. Many configuration and customization options exist depending on the types of hardware functions being accelerated.

The hardware platform creation process consists of building a Vivado® Design Suite design, configuring platform and interface properties for clocks, interrupts, and bus interfaces, and then writing the device support archive (DSA) file for use in an SDSoC platform. The logic design can be captured using IP integrator and can include RTL sources. A top-level wrapper is used to instantiate the IP integrator design as well as any top-level RTL modules. RTL modules can also be added directly to the IP integrator block design.

The `write_dsa` command archives the Vivado platform project data and associated files into a DSA file to define the hardware component of the platform. This chapter assumes you are familiar with the general features and processes of the Vivado Design Suite, and that you are able to create a Vivado project for the hardware in your platform. It describes the general requirements for the hardware platform, and the Vivado project.

Figure 27: **Platform Hardware Component Design Flow**


Launch Vivado
Create an IP integrator block design
Include IP blocks for processor, clocks, resets, and interrupts
Declare Interfaces for Accelerators & Data Movers
Build Block Design
Write Device Support Archive (DSA)
Validate DSA
X22057-112918

Hardware Requirements

This section describes requirements on the hardware design component of an SDSoC™ platform. In general, nearly any design targeting the Zynq® UltraScale+™ MPSoC or UltraScale+™ device using the IP integrator within the Vivado Design Suite can be the basis for an SDSoC platform.

The process of capturing the SDSoC hardware platform is conceptually straightforward:

1. Build and verify the hardware system using the Vivado Design Suite and IP integrator feature.
2. Configure platform and interface properties.
3. Write the DSA file.

There are several rules that the platform hardware design must observe.

TIP: *If the Xilinx design project contains more than one block diagram, one block diagram must have the same name as the hardware platform, and that block diagram is used by the SDx platform project.*

- Every IP used in the platform design that is not part of the standard Vivado IP catalog must be local to the Vivado Design Suite project. References to external IP repository paths are not supported by the `write_dsa` command.
- Every hardware platform design must contain a Processing System IP block from the Xilinx IP catalog.
- Every hardware port interface to the SDSoC platform must be an AXI, AXI4-Stream, clock, reset, or interrupt type interface only. Custom bus types or hardware interfaces must remain internal to the hardware platform.
- Every platform must declare at least one general purpose AXI master port from the Processing System IP or an interconnect IP connected to such an AXI master port, that will be used by the SDSoC compilers for software control of datamover and accelerator IP.
- Every platform must declare at least one AXI slave port that will be used by the SDSoC compilers to access DDR from datamover and accelerator IP.
- To share an AXI port between the SDSoC environment and platform logic, for example `S_AXI_ACP`, you must export an unused AXI master or slave of an AXI Interconnect IP block connected to the corresponding AXI port, and the platform must use the ports with least significant indices.
- Every platform AXI interface will be connected to a single data motion clock by the SDSoC environment.

TIP: *Accelerator functions generated by the SDSoC compilers might run on a different clock that is provided by the platform.*

- Every exported platform clock must have an accompanying Processor System Reset IP block from the Vivado IP catalog.
- Platform interrupt inputs must be exported by a Concat (`xlconcat`) IP connected to the Processing System 7 IP `IRQ_F2P` port. IP blocks within a platform can use some of the sixteen available fabric interrupts, but must use the least significant bits of the `IRQ_F2P` port without gaps.

Begin with a Vivado Project

An SDSoC platform project begins with a Vivado Design Suite project file (`<platform>.xpr`) as the starting point to build the platform device support archive (DSA) file.

The project must include an IP integrator block diagram and can also contain any number of source files. Although nearly any project targeting a Zynq®-7000, Zynq UltraScale+ MPSoC, or MicroBlaze™ processsor can be the basis for an SDSoC project, there are a few restrictions as described in Hardware Requirements.

IMPORTANT! *If you are moving the project file from one location to another, you must place the complete Vivado Design Suite project in the same directory as the project* `xpr` *file. You cannot simply copy the files in a Vivado tools project from one location to another. The Vivado Design Suite manages internal project states and file relationships in a way that is not preserved through a simple file copy. To properly copy the Vivado Design Suite project use the* ***File→Archive Project*** *command from the Xilinx IDE to create a zip archive. Copy and unzip this archive file into the new location. If you encounter IP Locked errors when the SDx IDE invokes the Vivado tools, it is a result of failing to properly copy the Vivado project, or failing to upgrade the project, IP. and output products for the latest version of the tool.*

Design Flow for Generating the DSA

To create the Vivado Design Suite project for use in an SDSoC platform:

1. Launch the Vivado Design Suite IDE.
2. Use the **Quick Start > Create Project** link on the Vivado Design Suite home screen or select **File→Project→New** on the Vivado Design Suite menu bar to launch the **New Project** wizard. Use the default project name, `project_1`.

TIP: *You can also edit an existing project as a starting point for creating a new SDSoC hardware platform.*

3. Choose the **RTL Project** type and advance to the **Default Part** dialog to select the Xilinx device or a supported board to use for the SDSoC platform. For this example, use the ZCU102 board. For more information on creating projects and selecting parts or boards, refer to the *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994).
4. After the project opens in the Vivado Design Suite IDE, click on the **Create Block Design** command underneath **IP Integrator** in the **Flow Navigator** window. Use the default settings. The block design will be named `design_1`.
5. On the IP integrator canvas, instantiate the embedded processor IP using the **Add IP (+)** icon. Search for Zynq and select the Zynq UltraScale+ MPSoC IP for this example. Run **Block Automation** and use the **Apply Board Preset** option. Additional IP from the IP catalog or custom IP can be added as needed to complete the design. See the completed IP integrator design in the figure below. IP blocks and connections have been added to meet the hardware requirements stated earlier.

   For more information on creating a block design using IP integrator, refer to the *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994).

   For more information on creating an embedded processor block design, refer to *Vivado Design Suite User Guide: Embedded Processor Hardware Design* (UG898).

6. a. Clocking Wizard
      i. Customize Output Clocks to generate 3 PL clocks
         i. **clk_out1** at 100 MHz, **clk_out2** at 200 MHz, and **clk_out3** at 300 MHz
      ii. **Reset Type = Active Low**

b. Processor System Reset

   i. Add 3 instances to provide a reset for each of the 3 PL clocks

   ii. Associate **clk_out1** with **proc_sys_reset_0**, **clk_out2** with **proc_sys_reset_1**, **clk_out3** with **proc_sys_reset_2**

   iii. Connect all **dcm_locked** inputs to the **clk_wiz_0 locked** output

   iv. Connect all **ext_reset_in** inputs to **pl_resetn0** output of the processor block

c. Concat

   i. Customize to set **Number of Ports = 1**

d. Edit zynq_ultra_ps_i_0 PCW settings

   i. **PS-PL Configuration > PS-PL Interfaces > Master Interface**

      i. Uncheck AXI_HPM0_FPD

      ii. Uncheck AXI_HPM1_FPD

7. Declare platform interfaces for use by the sds++ system compiler by setting the PFM properties on the interface ports by using either the Platform Interfaces tab or TCL commands.
8. Validate the block design to ensure everything is correct, and save the design.
9. Optionally enable IP caching to reduce synthesis and compilation times.
10. **Generate Output Products** of the IP in the block design.
11. Use the **Create HDL Wrapper** command to create the top-level RTL design.
12. **Export Hardware** to SDK for additional software development. Note, the SDx IDE can also be used to generate software for standalone and Linux targets. Refer to Chapter 4: Creating the Platform Software Component for more information on defining the software components.
13. If you are using programmable logic device I/O pins, assign I/O port constraints.
14. Optionally, simulate and implement the design to validate functionality and performance
15. Archive the project for use as a backup.
16. Write and validate the DSA using `write_dsa` and `validate_dsa` at the Vivado Design Suite Tcl console.

TIP: *The Vivado IDE creates a journal file (.jou) that contains TCL commands that have been executed during the preceding steps. This file can be used to to create a script to automate hardware platform creation.*

Logic Design Using the IP Integrator

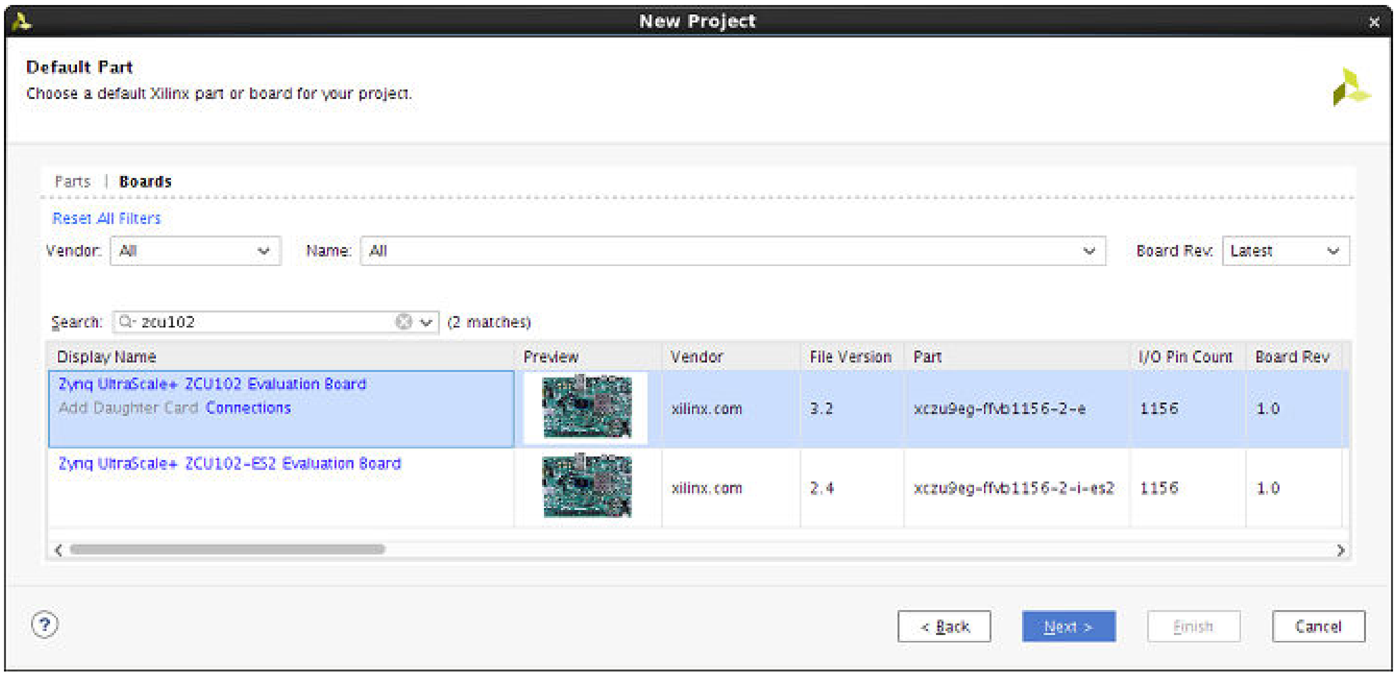
The Vivado Design Suite IP integrator offers interactive graphical design entry and configuration capabilities that are designed to streamline the design capture process. Various automatic designer assistance and configuration features are built into the environment. A large assortment of AXI4 compliant IP is available for most system design needs.

The logic design can be captured using IP integrator or with RTL sources. A top-level wrapper is used to instantiate the IP integrator design as well as any top-level RTL modules. RTL modules can also be added directly to the IP integrator block design (BD).

Capture your hardware platform logic design containing either a Zynq® SoC, Zynq UltraScale+ MPSoC, or MicroBlaze processor.

Using the instructions from the Design Flow for Generating the DSA section, a custom platform based on the ZCU102 board part is illustrated below. Use the description in the figure titles and settings shown in each figure as a guide in creating a hardware design with 3 PL clocks, the required platform IP and platform properties.

Figure 28: **Select Project Default Part**



Choose a default Xilinx part or board for your project.

Vendor: All Name: All Board Rev: Latest

Search: zcu102 (2 matches)

Display Name	Preview	Vendor	File Version	Part	I/O Pin Count	Board Rev
Zynq UltraScale+ ZCU102 Evaluation Board Add Daughter Card Connections		xilinx.com	3.2	xczu9eg-ffvb1156-2-e	1156	1.0
Zynq UltraScale+ ZCU102-ES2 Evaluation Board		xilinx.com	2.4	xczu9eg-ffvb1156-2-i-es2	1156	1.0

Figure 29: **Add Processor to IP Integrator Block Design**

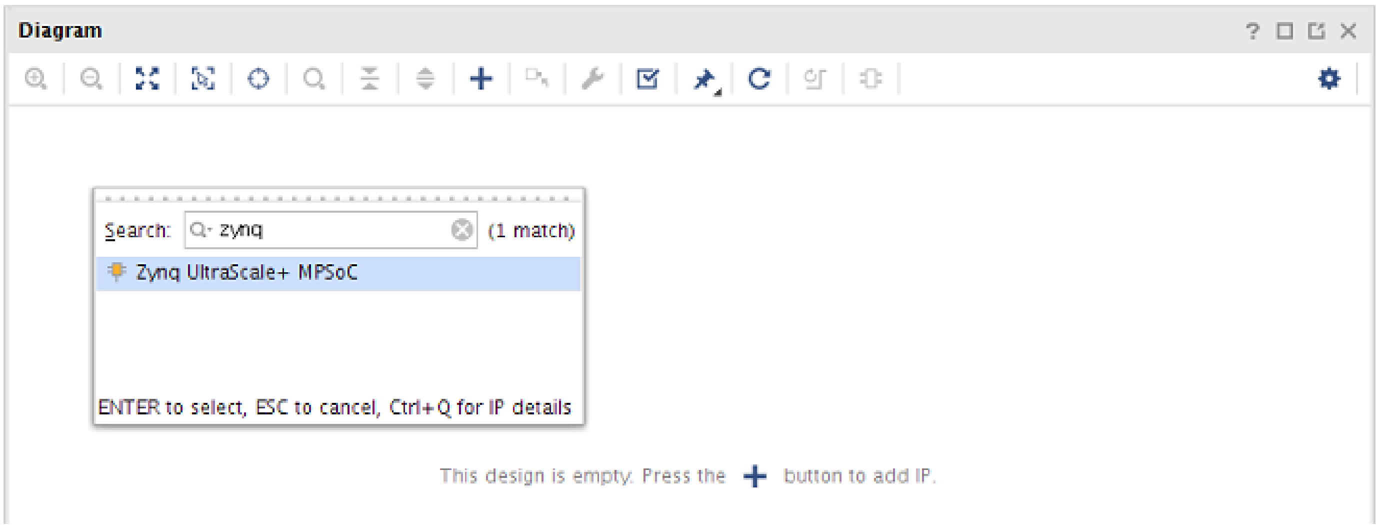


Diagram
Search: zynq (1 match)
Zynq UltraScale+ MPSoC
ENTER to select, ESC to cancel, Ctrl+Q for IP details
This design is empty. Press the + button to add IP.

Figure 30: **Run Board Automation**

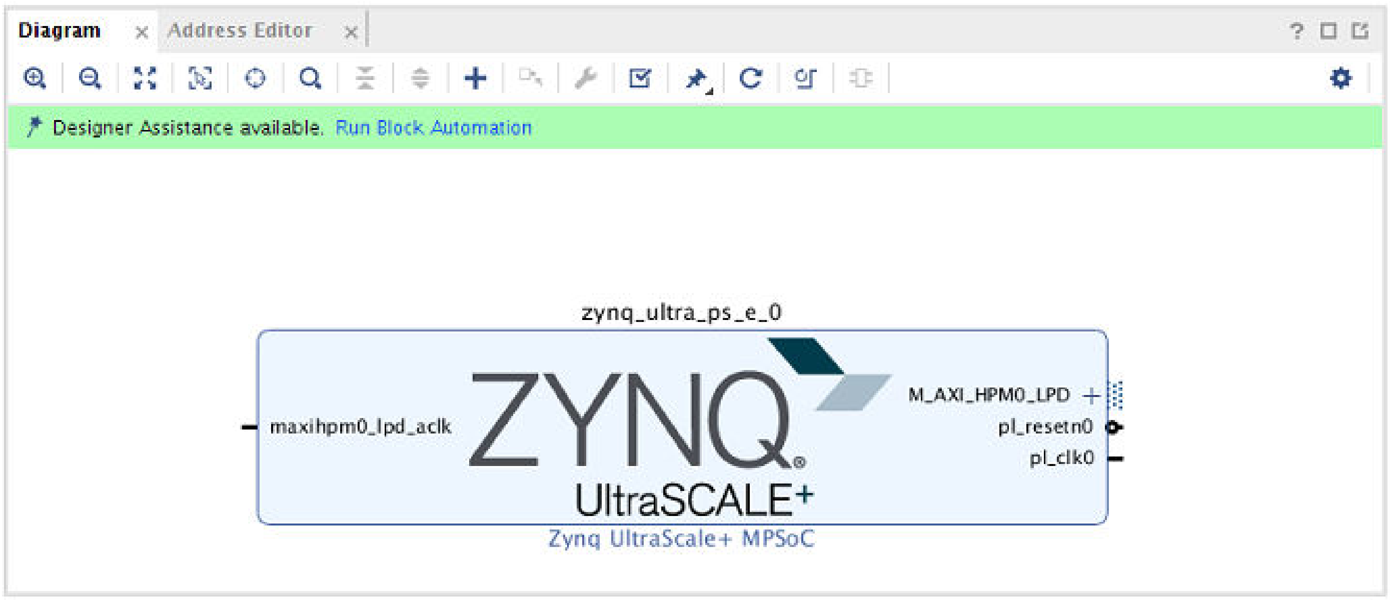
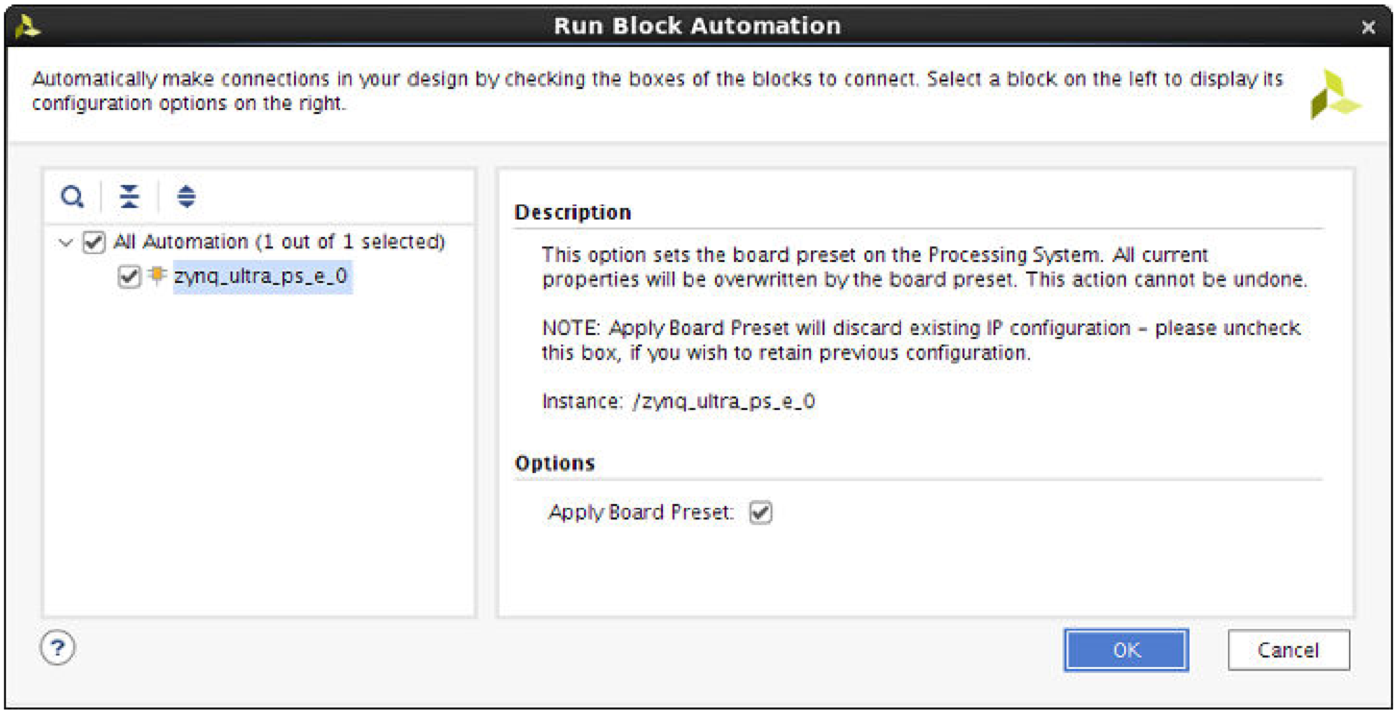


Figure 31: **Apply Presets**



Run Block Automation

Automatically make connections in your design by checking the boxes of the blocks to connect. Select a block on the left to display its configuration options on the right.

- [✓] All Automation (1 out of 1 selected)
  - [✓] zynq_ultra_ps_e_0

Description

This option sets the board preset on the Processing System. All current properties will be overwritten by the board preset. This action cannot be undone.

NOTE: Apply Board Preset will discard existing IP configuration - please uncheck this box, if you wish to retain previous configuration.

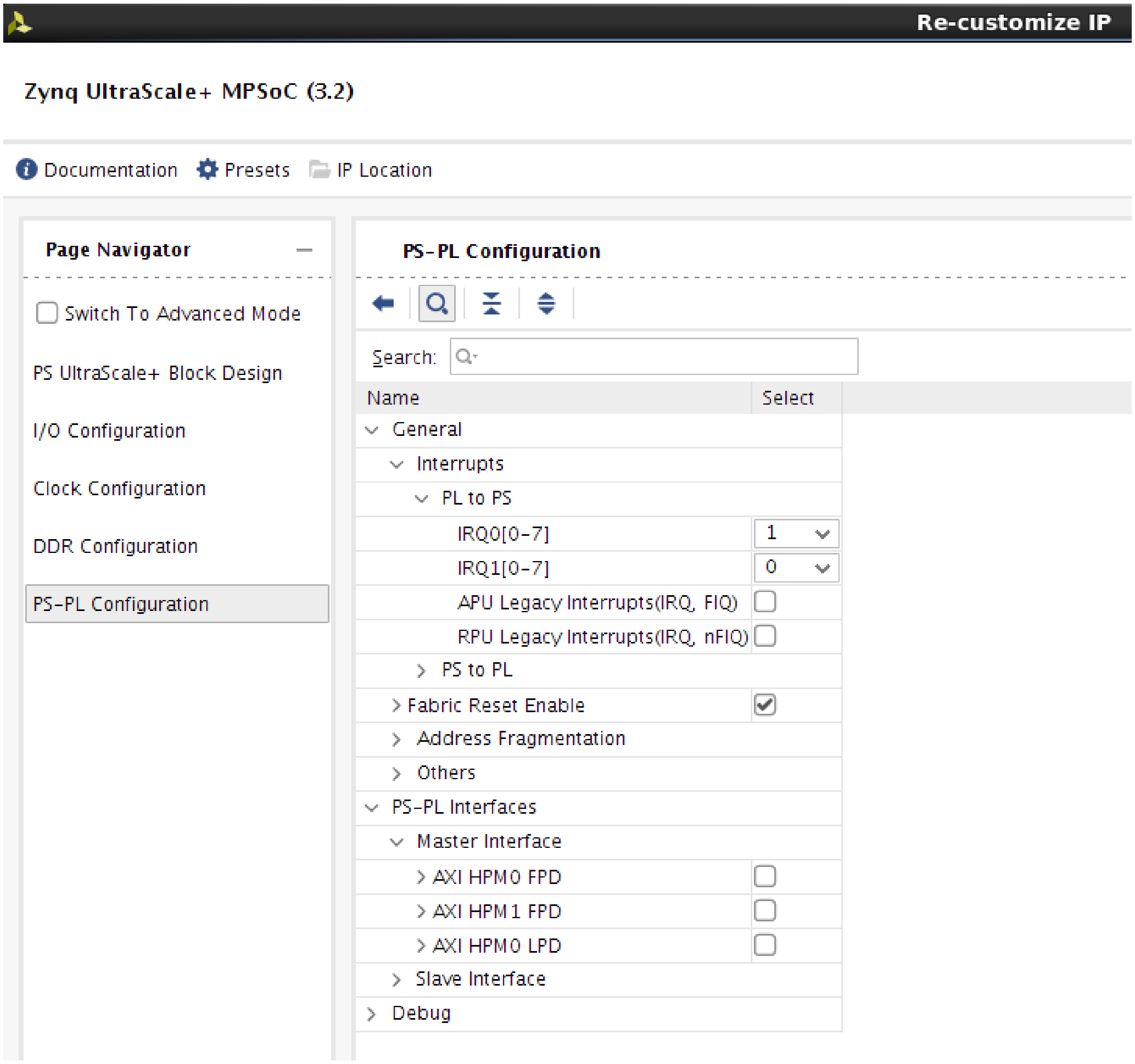
Instance: /zynq_ultra_ps_e_0

Options

Apply Board Preset: [✓]

After applying the ZCU102 board presets, the processor block is further customized by using the Processor Configuration Wizard (PCW). Double-clicking on the **zynq_ultra_ps_e_0** invokes the customization wizard. Ensure a path for PL to PS interrupts exists. The PCW figure below shows the **IRQ0[0-7]** input to the processor is enabled. The **AXI HPM0 FPD** and **AXI HPM1 FPD** master PS-PL interfaces have been unchecked so they are available for acclerator attachment.

Figure 32: **Apply Processor Configuration Wizard (PCW) Edits**


Re-customize IP

Zynq UltraScale+ MPSoC (3.2)

Documentation Presets IP Location

Page Navigator

☐ Switch To Advanced Mode

PS UltraScale+ Block Design

I/O Configuration

Clock Configuration

DDR Configuration

PS-PL Configuration

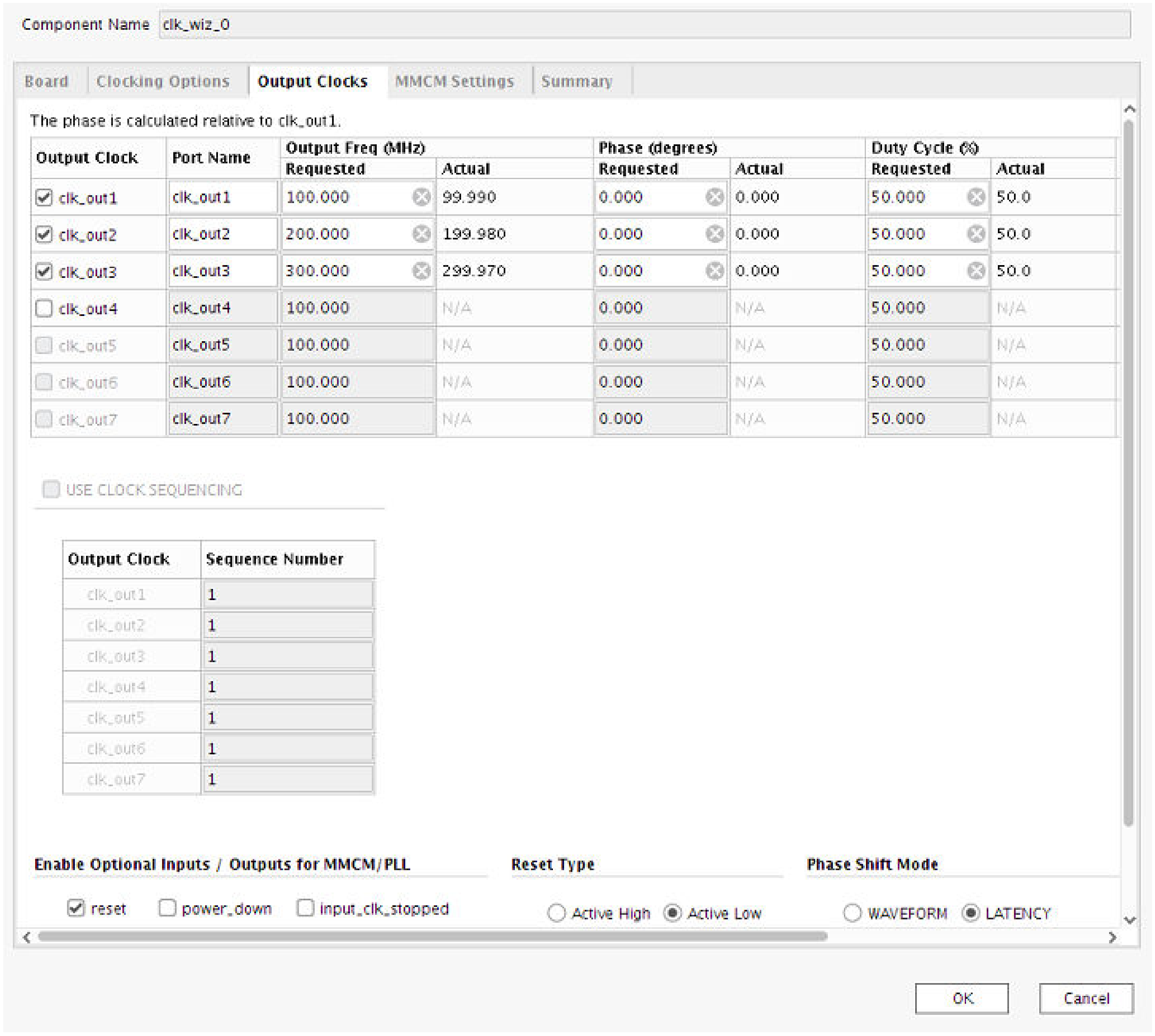
PS-PL Configuration

Search:

Name	Select
General	
Interrupts	
PL to PS	
IRQ0[0-7]	1
IRQ1[0-7]	0
APU Legacy Interrupts(IRQ, FIQ)	☐
RPU Legacy Interrupts(IRQ, nFIQ)	☐
PS to PL	
Fabric Reset Enable	✓
Address Fragmentation	
Others	
PS-PL Interfaces	
Master Interface	
AXI HPM0 FPD	☐
AXI HPM1 FPD	☐
AXI HPM0 LPD	☐
Slave Interface	
Debug	

After applying the PCW settings, Add and customize the Clocking Wizard and Processor Sysetem Reset IP blocks.

Figure 33: **Add Output Clocks with Clocking Wizard**



Component Name: clk_wiz_0

The phase is calculated relative to clk_out1.

Output Clock	Port Name	Output Freq (MHz) Requested	Output Freq (MHz) Actual	Phase (degrees) Requested	Phase (degrees) Actual	Duty Cycle (%) Requested	Duty Cycle (%) Actual
✓ clk_out1	clk_out1	100.000	99.990	0.000	0.000	50.000	50.0
✓ clk_out2	clk_out2	200.000	199.980	0.000	0.000	50.000	50.0
✓ clk_out3	clk_out3	300.000	299.970	0.000	0.000	50.000	50.0
clk_out4	clk_out4	100.000	N/A	0.000	N/A	50.000	N/A
clk_out5	clk_out5	100.000	N/A	0.000	N/A	50.000	N/A
clk_out6	clk_out6	100.000	N/A	0.000	N/A	50.000	N/A
clk_out7	clk_out7	100.000	N/A	0.000	N/A	50.000	N/A

USE CLOCK SEQUENCING

Output Clock	Sequence Number
clk_out1	1
clk_out2	1
clk_out3	1
clk_out4	1
clk_out5	1
clk_out6	1
clk_out7	1

Enable Optional Inputs / Outputs for MMCM/PLL

✓ reset ☐ power_down ☐ input_clk_stopped

Reset Type

Active High [✓ Active Low]

Phase Shift Mode

WAVEFORM [✓ LATENCY]

Figure 34: **Clocks and Processor Reset Blocks**

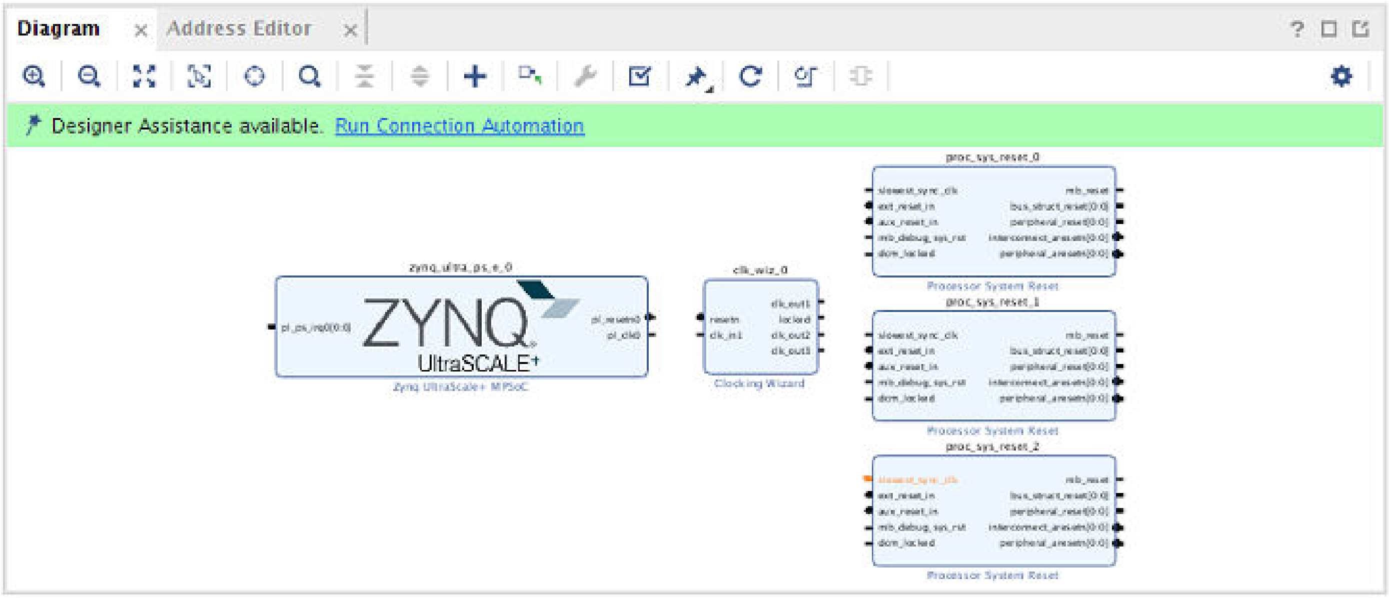
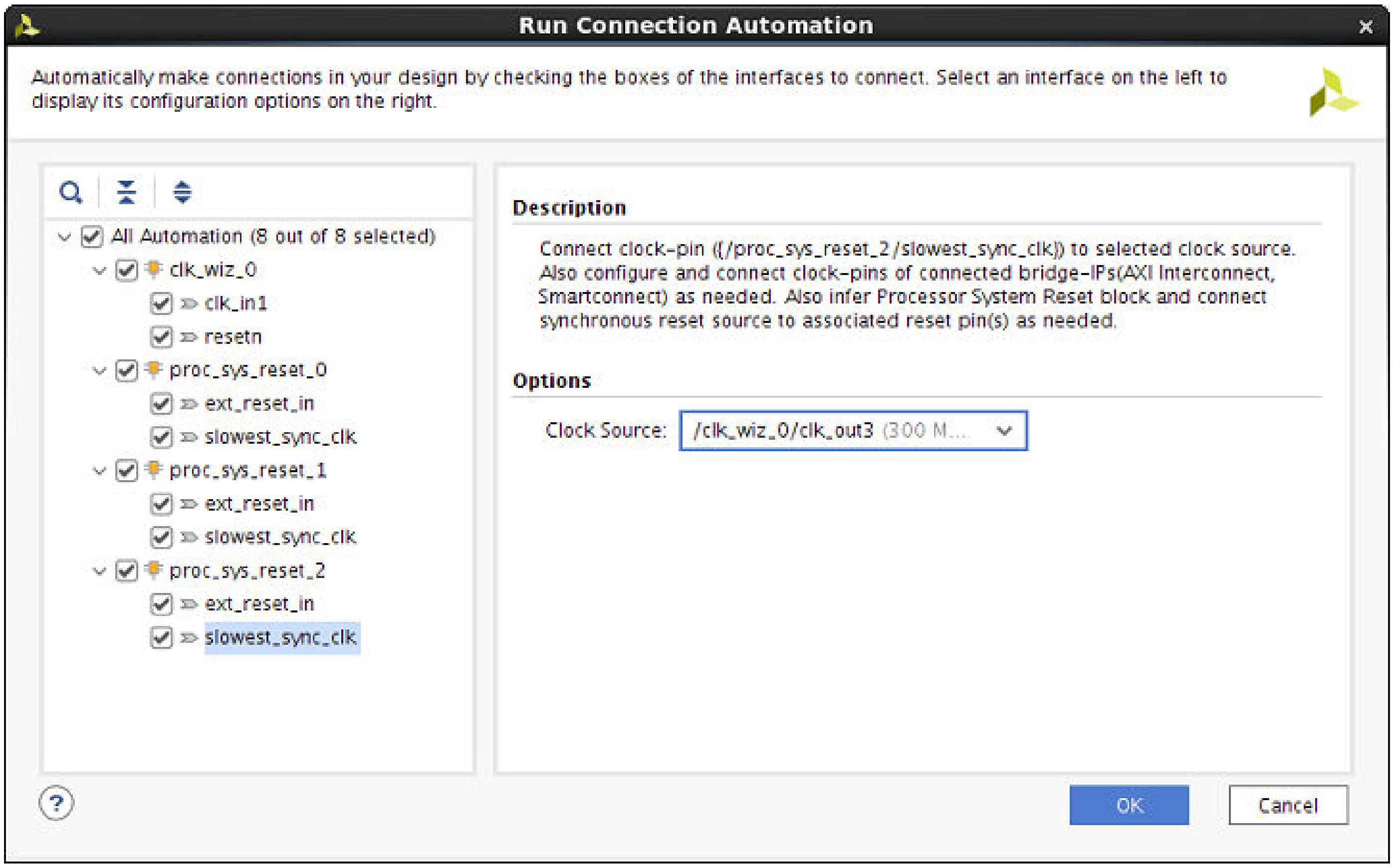


Figure 35: **Run Connection Automation**


Run Connection Automation

Automatically make connections in your design by checking the boxes of the interfaces to connect. Select an interface on the left to display its configuration options on the right.

- [✓] All Automation (8 out of 8 selected)
  - [✓] clk_wiz_0
    - [✓] clk_in1
    - [✓] resetn
  - [✓] proc_sys_reset_0
    - [✓] ext_reset_in
    - [✓] slowest_sync_clk
  - [✓] proc_sys_reset_1
    - [✓] ext_reset_in
    - [✓] slowest_sync_clk
  - [✓] proc_sys_reset_2
    - [✓] ext_reset_in
    - [✓] slowest_sync_clk

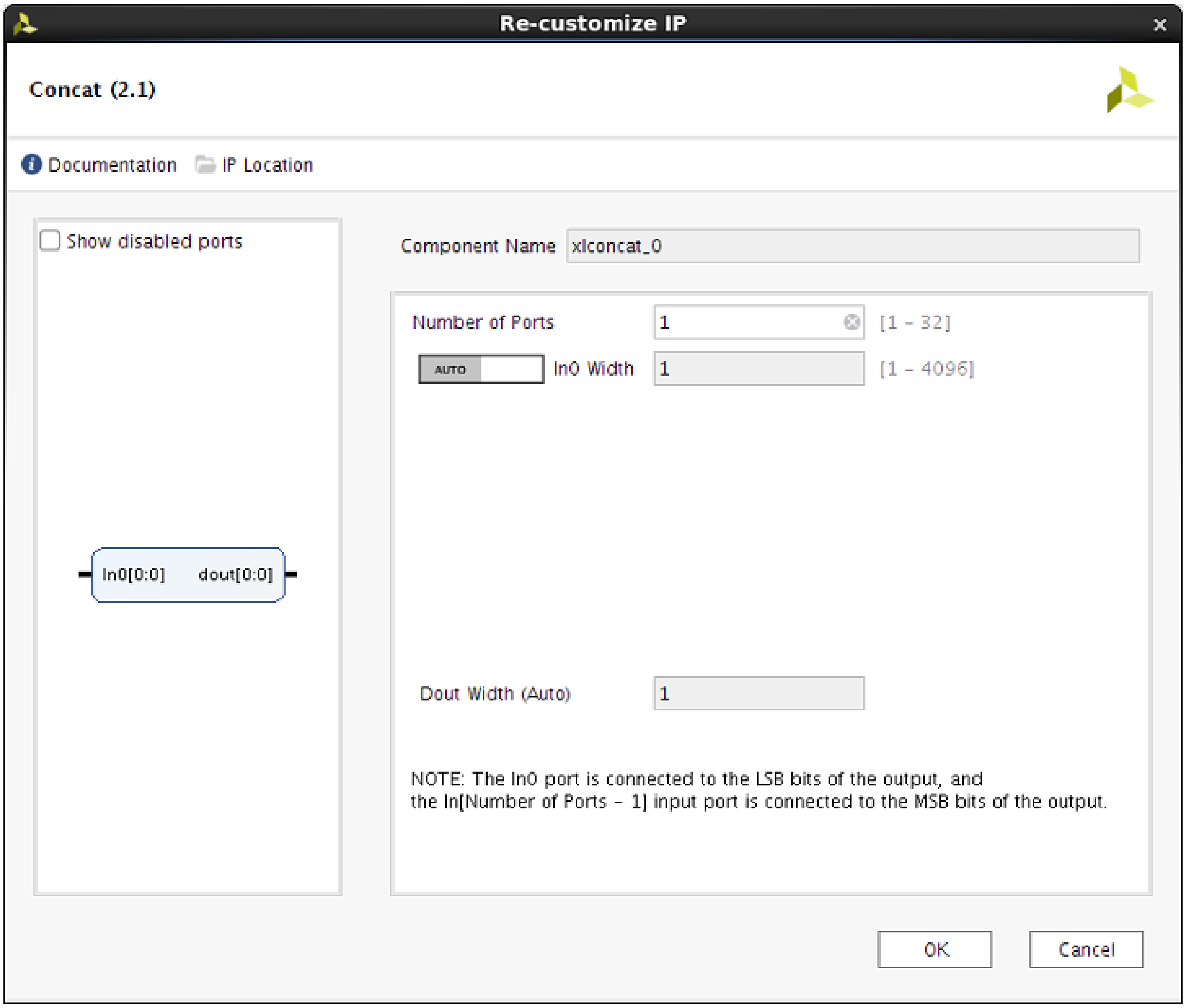
Description

Connect clock-pin ({/proc_sys_reset_2/slowest_sync_clk}) to selected clock source. Also configure and connect clock-pins of connected bridge-IPs(AXI Interconnect, Smartconnect) as needed. Also infer Processor System Reset block and connect synchronous reset source to associated reset pin(s) as needed.

Options

Clock Source: /clk_wiz_0/clk_out3 (300 M...

Figure 36: **Concat Block for Interrupt Customization**



Concat (2.1)

Show disabled ports

Component Name xlconcat_0

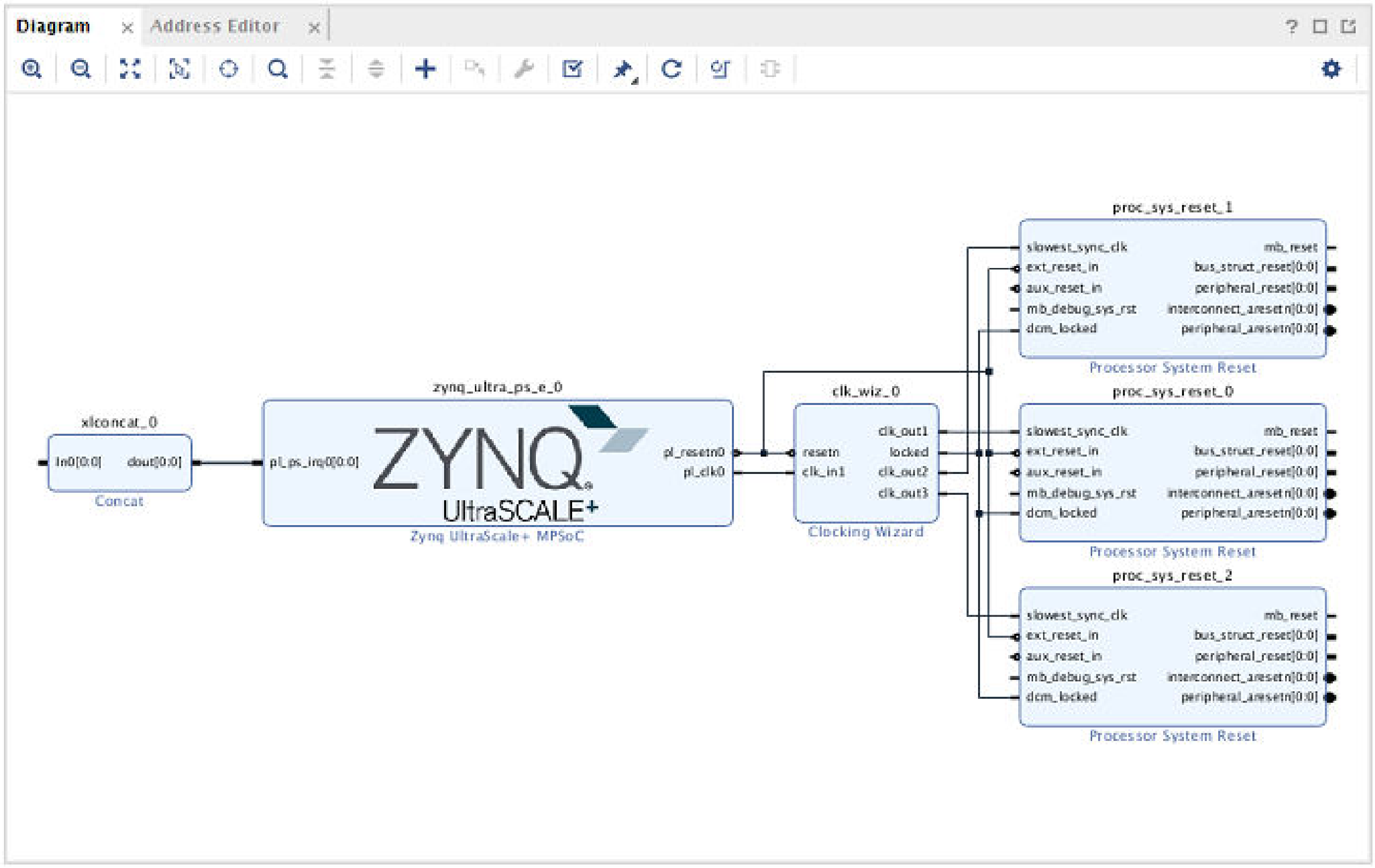
Number of Ports 1 [1 - 32]

In0 Width 1 [1 - 4096]

Dout Width (Auto) 1

NOTE: The In0 port is connected to the LSB bits of the output, and
the In[Number of Ports - 1] input port is connected to the MSB bits of the output.

Figure 37: **Completed IP Integrator Hardware Design for Custom Platform**



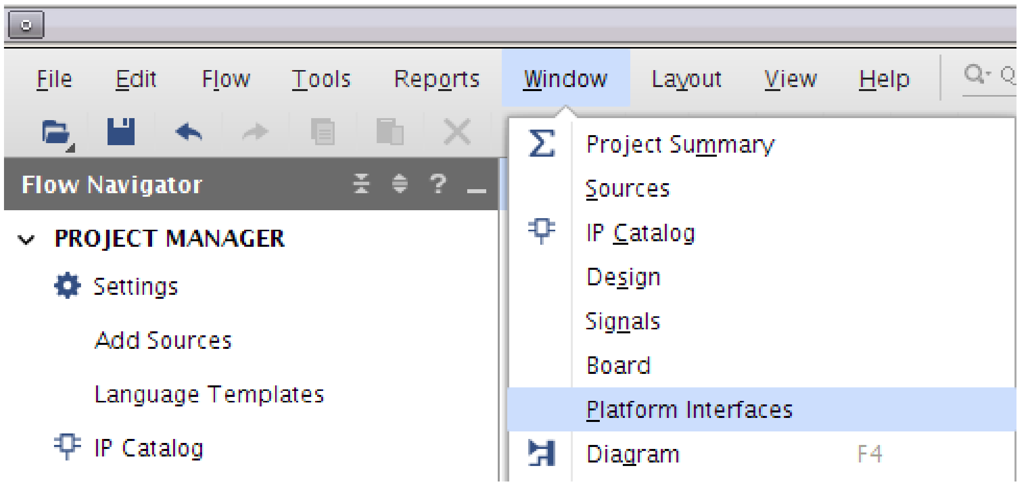
After the block design is complete, you can apply platform properties to different interfaces to be used by the hardware function(s) within the SDx environment.

For more information on creating block designs using IP integrator and applying platform properties to available interfaces in the block design, refer to *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994).

Declaring Platform (PFM) Interfaces and Properties

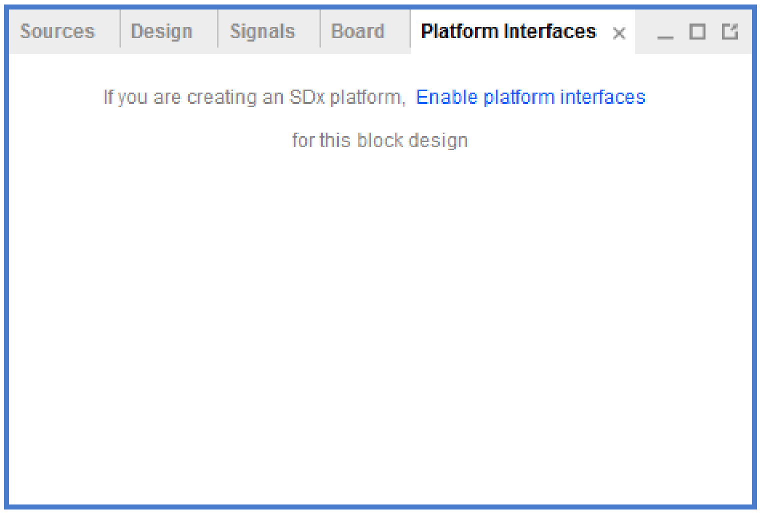
After you complete the IP integrator hardware block design in the Vivado Design Suite, you must declare and add platform (PFM) properties on IP blocks for clocking (Clocking Wizard), interrupts (Concat), resets (Processor System Reset), and the processor (Zynq UltraScale+ MPSoC) AXI interfaces. These declared interfaces will then be available for hardware function(s) within the SDx environment. The simplest and easiest way to declare these interfaces and their properties is through the Platform Interfaces tab of the block design. Enable the Platform Interfaces tab, by selecting **Window → Platform Interfaces** from the Vivado menu bar and clicking on the **Enable platform interfaces** link. These properties are set once and stored in the project. A description of the underlying Tcl commands executed to set the PFM properties are also shown for reference.

Figure 38: **Enabling the Platform Interfaces Tab**



This opens up the Platform Interfaces tab.

Figure 39: **Enabling the Interfaces in the Platform Interfaces Tab**

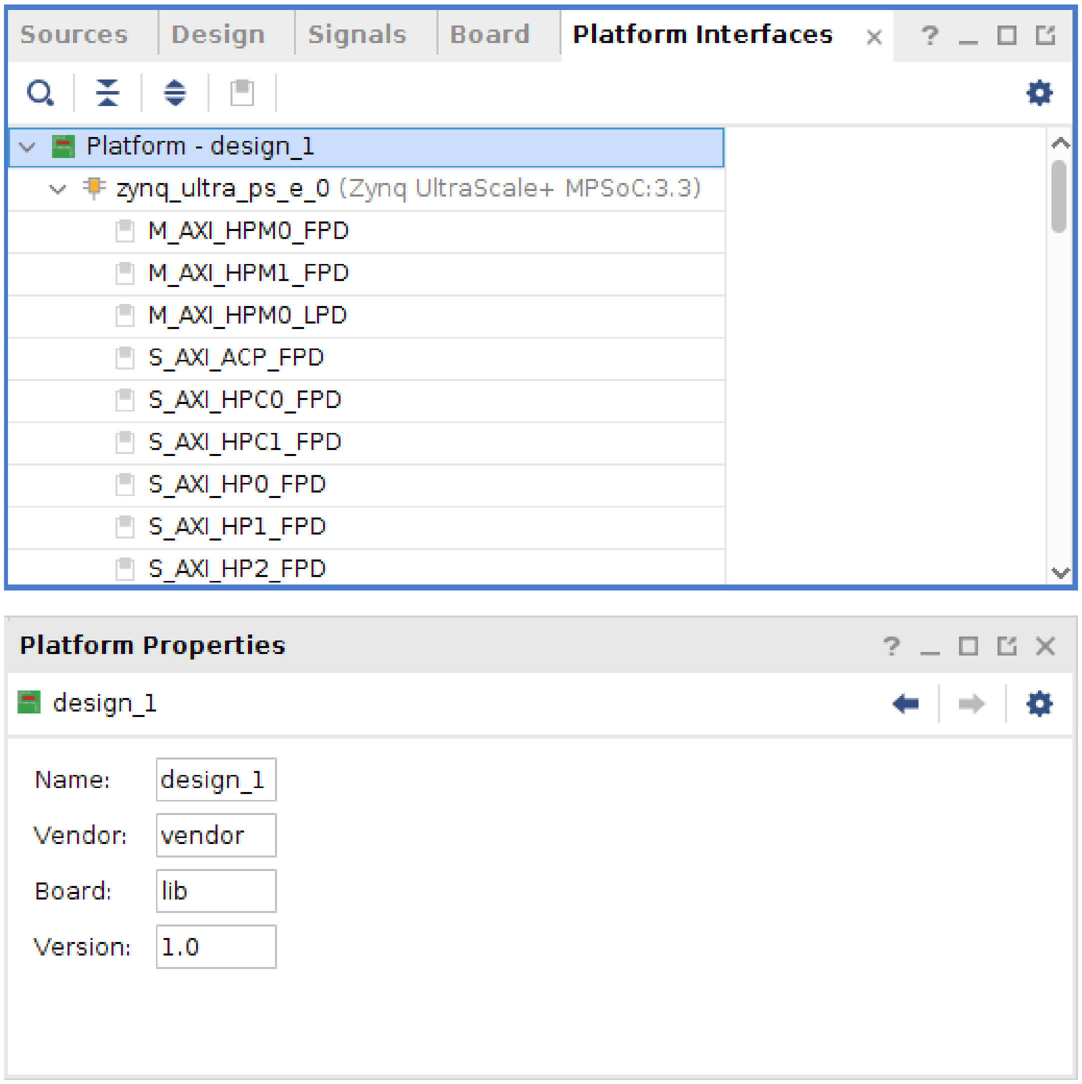


If you are creating an SDx platform, Enable platform interfaces for this block design

Clicking on the **Enable platform interfaces** link automatically populates the PFM_NAME property and lists all the interfaces that can be enabled for use by hardware accelerators within the SDx environment.

Selecting the Platform in the Platform Interfaces window shows the settings for the platform Name, Vendor, Board, and Version in the Platform Properties window. These platform properties are user editable by selecting the text box associated with each property.

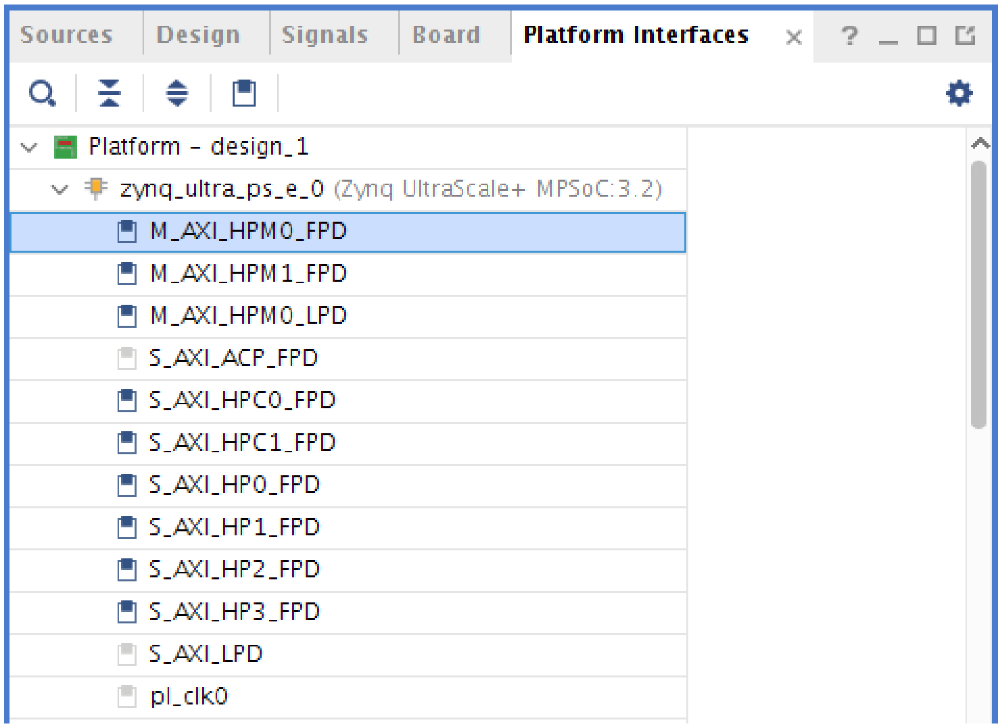
Figure 40: **Setting the Platform PFM_NAME: Name, Vendor, Board, and Version**



Name: design_1
Vendor: vendor
Board: lib
Version: 1.0

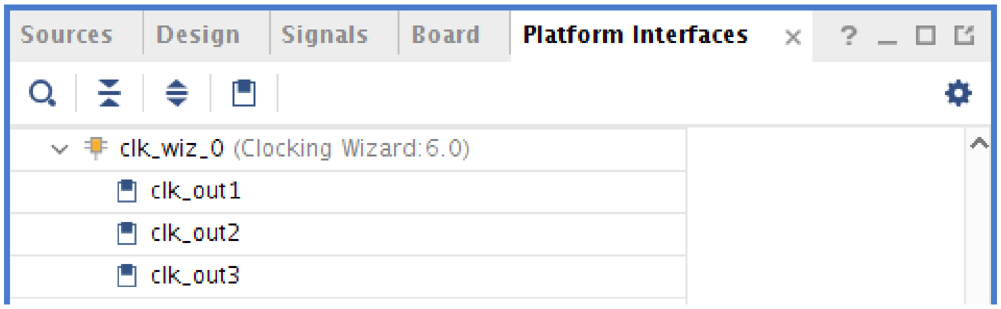
The Platform Interfaces tab should now show all the interfaces available in the block design that can be tagged with platform specific properties. To enable an interface, right-click an interface and select **Enable**. For the example ZCU102 hardware design, enable the zynq_ultra_ps_e_0, clk_wiz_0, and xlconcat_0 ports as shown below.

Figure 41: **Enabling an Interface in the Platform**



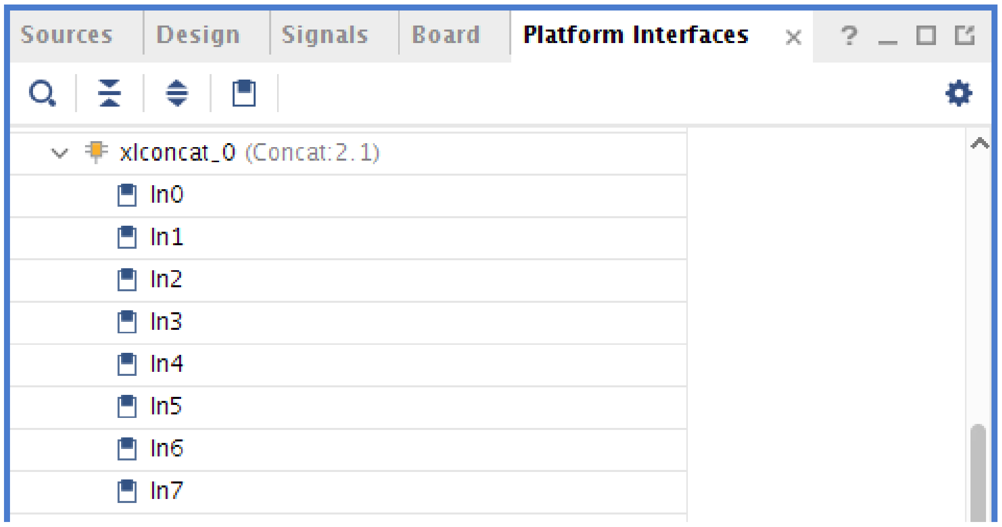
Platform – design_1
zynq_ultra_ps_e_0 (Zynq UltraScale+ MPSoC:3.2)
M_AXI_HPM0_FPD
M_AXI_HPM1_FPD
M_AXI_HPM0_LPD
S_AXI_ACP_FPD
S_AXI_HPC0_FPD
S_AXI_HPC1_FPD
S_AXI_HP0_FPD
S_AXI_HP1_FPD
S_AXI_HP2_FPD
S_AXI_HP3_FPD
S_AXI_LPD
pl_clk0

Figure 42: **Clocking Wizard Interfaces Enabled**



clk_wiz_0 (Clocking Wizard:6.0)
clk_out1
clk_out2
clk_out3

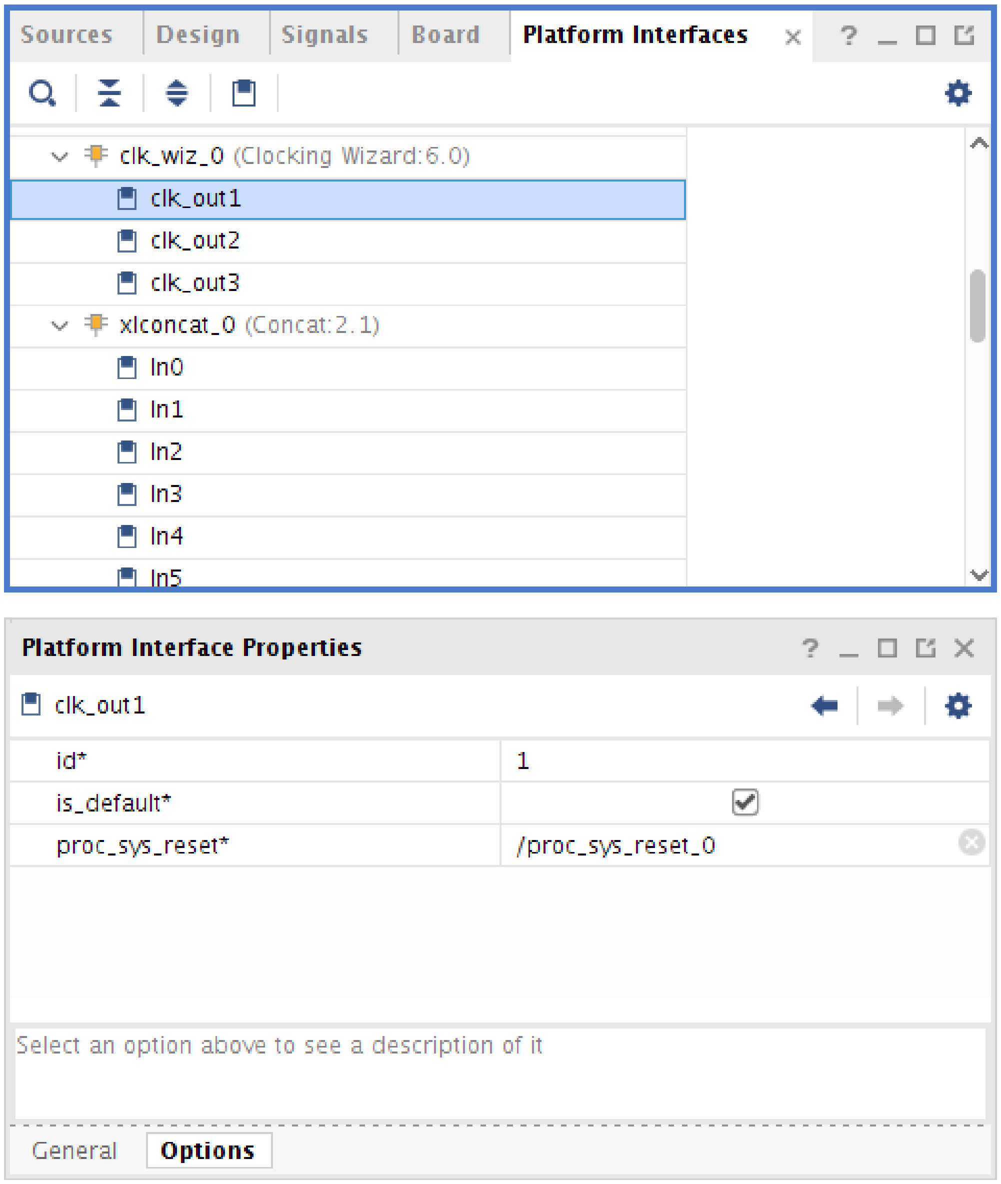
Figure 43: **Concat Block Interrupt Interfaces Enabled**



xlconcat_0 (Concat:2.1)
In0
In1
In2
In3
In4
In5
In6
In7

The default PL clock setting for the platform is required and can be set by selecting the desired clock in the **Platform Interfaces** view and selecting the **Options** tab in the selected clocks **Platform Interface Properties** dialog. Click on the check-box associated with the **is_default** property to toggle the setting.

Figure 44: **Setting the Default Clock**



clk_wiz_0 (Clocking Wizard:6.0)
clk_out1
clk_out2
clk_out3
xlconcat_0 (Concat:2.1)
In0
In1
In2
In3
In4
In5

clk_out1

id*	1
is_default*	✓
proc_sys_reset*	/proc_sys_reset_0

Select an option above to see a description of it

Perform the remaining steps to build the hardware and generate the DSA. The remainder of this chapter adds further details on platform properties and implementing the hardware design.

1. Validate the IP integrator block design by right-clicking on the IP integrator canvas and selecting **Validate Design**.
2. In the **Sources** tab, right-click on `design_1.bd` and select **Generate Output Products**. Use the default Out of context per IP synthesis option and run settings then click **Generate**.
3. In the **Sources** tab, right-click on `design_1.bd` and select **Create HDL Wrapper**.
4. In the **Flow Navigator** underneath **Program and Debug** click on **Generate Bitstream**.

5. Use the **File → Export → Export Hardware** command to write the hardware description file for the project. Select **Include bitstream** when prompted in the Export Hardware dialog box.
6. In the Tcl Console command box write and validate the DSA:
   - `write_dsa design_1.dsa`
   - `validate_dsa design_1.dsa`

For more information on creating block designs using IP integrator and applying platform properties to available interfaces in the block design, refer to *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994).

Setting the Platform Name

The Platform Identification property (PFM_NAME) must be set in the hardware design to define the Vendor, Library, Name, and Version (VLNV) of the platform.

```
set_property PFM_NAME string [get_files design.bd]
```

Where:

- `string` is defined in the standard VLNV format, for example:

  ```
  xilinx.com:my_lib:platformA:1.0
  ```

- `design.bd` specifies the file name of the block design.

TIP: *`PFM_NAME` can also be specified in simple form with just the `Name` from the VLNV form. The Vendor, Library, and Version fields will be populated with default values: `vendor`, `lib`, and `1.0`.*

Example:

```
set_property PFM_NAME zcu102 [get_files zcu102.bd]
```

This results in the PFM_NAME: `vendor:library:zcu102:1.0`.

The Vivado block design and the DSA will store this property.

IMPORTANT! *The `write_bd_tcl` command does not write the PFM properties to the resulting bd_Tcl script. These properties must be exported manually to be preserved. Xilinx recommends using the Archive Project command to backup the project.*

Configuring Platform Interface Properties

The Platform Interfaces are defined using the four PFM properties described below. They can be defined manually in the Tcl Console, or by a Tcl script for the design.

The four Platform Interfaces Tcl APIs are:

```
set_property PFM.AXI_PORT { <port_name> {parameters} \
<port2> {parameters} ...} [get_bd_cells <cell_name>]
set_property PFM.AXIS_PORT { <port_name> {parameters} \
<port2> {parameters} ...} [get_bd_cells <cell_name>]
set_property PFM.CLOCK { <port_name> {parameters} \
<port2> {parameters} ...} [get_bd_cells <cell_name>]
set_property PFM.IRQ { <port_name> {} <port2> {} ...} \
[get_bd_cells <cell_name>]
```

The requirements for the PFM Properties are:

- The value of the PFM interface properties must be specified as a Tcl dictionary, a list of name/"value" pairs.

IMPORTANT! *The "value" must be quoted, and both the name and value are case sensitive.*

- A bd_cell can have multiple PFM interface definitions. However, for each type of PFM interface, all ports are required to be set in a single `set_property` Tcl command.
- For each PFM interface property, the name specified for the port object must match the name of an external port or interface on a bd_cell. Each external port or interface object may only have one PFM interface definition.
- Each different type of PFM interface may have different parameters.
- Setting the PFM property with a NULL ("") string will delete previously defined PFM interfaces.

Declaring Clocks

You can export any clock source with the platform, but for each clock you must also export synchronized reset signals using a Processor System Reset IP block in the platform. The PFM.CLOCK property can be set on a BD cell, external port, or external interface.

The Tcl command for setting the PFM.CLOCK property is:

```
set_property PFM.CLOCK { <port_name> {parameters} \
<port2> {parameters} ...} [get_bd_cells <cell_name>]
```

Argument Description

- `Port_name`: Clock port name.
- `Parameters`:
  - `id` <value>: Clock ID is a user-defined value that must be a unique non-negative integer.
  - `is_default` <value>: Specify "true" if this is the default clock, "false" otherwise. The default is "false."

  - `proc_sys_reset` <value>: This name/value pair specifies the corresponding `proc_sys_reset` block instance for synchronized reset signals connected to the clock port.

IMPORTANT! *Every platform must declare one default clock with the `is_default` parameter set to "true" for the SDSoC environment to use when no explicit clock has been specified.*

Examples:

```
set_property PFM.CLOCK {
PL_CLK0 {id "0" is_default "true" proc_sys_reset \
"proc_sys_reset_0"}
PL_CLK1 {id "1" is_default "false" proc_sys_reset \
"proc_sys_reset_1"}
PL_CLK2 {id "2" is_default "false" proc_sys_reset \
"proc_sys_reset_2"}
PL_CLK3 {id "3" is_default "false" proc_sys_reset \
"proc_sys_reset_3"}
} [get_bd_cells /zynq_ultra_ps_e_0]
```

To set a CLOCK on an external PORT:

```
set_property PFM.CLOCK
{ACLK_0 {id "4" is_default "false" proc_sys_reset \
"proc_sys_reset_4"}} [get_bd_ports /ACLK_0]
```

Declaring AXI Ports

The Tcl command for setting the PFM.AXI_PORT property is:

```
set_property PFM.AXI_PORT { <port_name> {parameters} \
<port2> {parameters} ...} [get_bd_cells <cell_name>]
```

Argument Description

- `Port_name`: AXI port name.
- `Parameters`:
  - `memport type`: Corresponding memory interface port type. Valid `type` values include:
    - `M_AXI_GP`: A general-purpose AXI master port
    - `S_AXI_HP`: A high-performance AXI slave port
    - `S_AXI_ACP`: An accelerator coherent slave port
    - `S_AXI_HPC`: A high-performance accelerator coherent slave port
    - `MIG`: An AXI slave connected to a MIG memory controller. The default is MIG.

- `sptag ID`: (Optional) A user-defined ID that should start with an alphabetic character. The ID is case-sensitive. The system port tag (sptag) is a symbolic identifier that represents a class of platform port connections, e.g., S_AXI_HP, S_AXI_ACP, M_AXI_GP. Multiple block design platform ports can share the same **sptag**.
- `memory`: (Optional) Specify the associated MIG IP instance and address_segment. The memory tag is a unique identifier that combines the `Cell` name and `Base Name` columns in the IP integrator **Address Editor**. This tag will be associated with connections to the Memory Subsystem HIP, where multiple block design platform ports can share the same memory tag.

IMPORTANT! *ACE and ACP ports are not supported on Zynq UltraScale+ MPSoC platforms. However, ACP is supported on Zynq-7000 SoC platforms.*

Cache-coherent Support for HPC Ports on Zynq UltraScale+ Devices

Platforms that use HPC ports assuming 2018.2 behavior (non-coherent with cache flushing) must label the ports with the type S_AXI_HP, instead of S_AXI_HPC. Both behave as though they are HP ports. However, ports labeled S_AXI_HPC are handled to enable coherence. The platform author should also adjust the BIF file, as shown below, for HPC ports with coherence enabled.

This is the recommended method for Linux boot as it guarantees that the register is written prior to the APU coming out of reset.

The Boot ROM can be used to write the register by using an init value in the boot image. Bootgen allows the init value to be added to the boot image. The following bif file snippet for bootgen illustrates the addition of the file containing an init value.

```
//arch = zynqmp; split = false; format = BIN
the_ROM_image:
{
    ...
    [init]<path>\regs.init
}
```

The following line of code illustrates the init value that should be in the `regs.init` file to cause outer shareable transactions to be broadcast to the CCI.

```
.set. 0xFF41A040 = 0x3;
```

For more information, see the Zynq UltraScale+ MPSoC Cache Coherency wiki page.

Example for an AXI Interconnect

```
set_property PFM.AXI_PORT { \
    M_AXI_GP0 {memport "M_AXI_GP"} \
    M_AXI_GP1 {memport "M_AXI_GP"} \
    S_AXI_ACP {memport "S_AXI_ACP" sptag "ACP" memory \
"processing_system7_0 ACP_DDR_LOWOCM"} \
    S_AXI_HP0 {memport "S_AXI_HP" sptag "HP0" memory \
"processing_system7_0 HP0_DDR_LOWOCM"} \
    S_AXI_HP1 {memport "S_AXI_HP" sptag "HP1" memory \
"processing_system7_0 HP1_DDR_LOWOCM"} \
    S_AXI_HP2 {memport "S_AXI_HP" sptag "HP2" memory \
"processing_system7_0 HP2_DDR_LOWOCM"} \
    S_AXI_HP3 {memport "S_AXI_HP" sptag "HP3" memory \
"processing_system7_0 HP3_DDR_LOWOCM"} \
    } [get_bd_cells /processing_system7_0]
```

Exporting AXI interconnect master and slave ports involves several requirements.

1. All ports on the interconnect used within the platform must precede in index order any declared platform interfaces.
2. There can be no gaps in the port indexing.
3. The maximum number of master IDs for the S_AXI_ACP port is eight, so on a connected AXI interconnect, available ports to declare must be one of {S00_AXI, S01_AXI, ..., S07_AXI}. Do not declare any ports that are used within the platform itself. Declaring as many as possible will allow `sds++` to avoid cascaded axi_interconnects in generated user systems.
4. The maximum number of master IDs for an S_AXI_HP or MIG port is sixteen, so on an connected AXI interconnect, available ports to declare must be one of {S00_AXI, S01_AXI, ..., S15_AXI}. Do not declare any ports that are used within the platform itself. Declaring as many as possible will allow `sds++` to avoid cascaded axi_interconnects in generated user systems.
5. The maximum number of master ports declared on an interconnect connected to an M_AXI_GP port is sixty-four, so on an connected AXI interconnect, available ports to declare must be one of {M00_AXI, M01_AXI, ..., M63_AXI}. Do not declare any ports that are use within the platform itself. Declaring as many as possible will allow `sds++` to avoid cascaded axi_interconnects in generated user systems.

Additional Examples

To define an AXI_port on interconnect:

```
set parVal []
for {set i 2} {$i < 64} {incr i} {
    lappend parVal M[format %02d $i]_AXI \
{memport "M_AXI_GP"}
}
set_property PFM.AXI_PORT $parVal [get_bd_cells /axi_interconnect_0]
```

To define an AXI_port on SmartConnect IP:

```
set parVal []
for {set i 1} {$i < 16} {incr i} {
    lappend parVal S[format %02d $i]_AXI \
{memport "MIG" sptag "Bank0"}
}
set_property PFM.AXI_PORT $parVal [get_bd_cells /smartconnect_0]
```

To define an AXI_PORT that connects with MIG IP:

```
set parVal []
for {set i 1} {$i < 16} {incr i} {
    lappend parVal S[format %02d $i]_AXI \
{memport "MIG" sptag "bank0" memory "ddrmem_0 C0_DDR4_ADDRESS_BLOCK"}
}
set_property PFM.AXI_PORT $parVal [get_bd_cells \
/memory_subsystem/interconnect_data/interconnect_aximm_ddrmem0]
```

Declaring AXI4-Stream Ports

The Tcl command for setting the PFM.AXIS_PORT property is:

```
set_property PFM.AXIS_PORT { <port_name> {parameters} \
<port_name_2> {parameters} .. } [get_bd_cells <cell_name>]
```

Argument Description

- `Port_name`: AXI4-Stream port name.
- `Parameters`:
    - `type value`: Streaming interface port type. Valid values for type include:
        - `M_AXIS`: A general-purpose AXI master port
        - `S_AXIS`: A high-performance AXI slave port

Examples

```
set_property PFM.AXIS_PORT {AXIS_P0 {type "S_AXIS"}} \
[get_bd_cells /zynq_ultra_ps_e_0]
```

Declaring Interrupt Ports

Interrupts must be connected to IP integrator Concat (xlconcat) blocks that are connected to the processing system. For Zynq-7000 family it is the `F2P_irq` port. For Zynq UltraScale+ MPSoC devices the interrupts are split into two 8-bit ports: `pl_ps_irq0[7:1]` and `pl_ps_irq1[7:1]`.

IMPORTANT! *If any IP within the platform includes interrupts, these must occupy the least significant bits of the Concat block without gaps.*

The Tcl command for setting the PFM.IRQ property is:

```
set_property PFM.IRQ { <port_name> {} <port2> {} ...} \
[get_bd_cells <cell_name>]
```

Argument Description

- `Port_name`: IRQ port name
- `{}`: Empty list that serves as a placeholder.

Example

```
set irqProp []
for {set i 0} {$i < 8} {incr i}
{ lappend irqProp In$i {} }
set_property PFM.IRQ $irqProp [get_bd_cells /xlconcat_0]
set_property PFM.IRQ $irqProp [get_bd_cells /xlconcat_1
```

TIP: *The FOR loop results in a PFM.IRQ property as defined by `$irqProp` that looks like:*

```
In0 {} In1 {} In2 {} In3 {} In4 {} In5 {} In6 {} In7 {}
```

Example PFM Property Tcl Script

This example script assigns the PFM properties to the block design on the Xilinx supplied ZCU102 platform.

```
# set_property PFM_NAME "xilinx.com:zcu102:zcu102:1.0" \
[get_files ./zcu102/zcu102.srcs/sources_1/bd/zcu102/zcu102.bd]
# set_property PFM.CLOCK { \
# clk_out1 {id "0" is_default "false" proc_sys_reset \
"proc_sys_reset_0" } \
# clk_out2 {id "1" is_default "true" proc_sys_reset \
"proc_sys_reset_0" } \
# clk_out3 {id "2" is_default "false" proc_sys_reset \
"proc_sys_reset_0" } \
# clk_out4 {id "3" is_default "false" proc_sys_reset \
"proc_sys_reset_0" } \
# clk_out5 {id "4" is_default "false" proc_sys_reset \
"proc_sys_reset_0" } \
# clk_out6 {id "5" is_default "false" proc_sys_reset \
"proc_sys_reset_0" } \
# clk_out7 {id "6" is_default "false" proc_sys_reset \
"proc_sys_reset_0" } \
# } [get_bd_cells /clk_wiz_0]
# set_property PFM.AXI_PORT { \
# M_AXI_HPM0_FPD {memport "M_AXI_GP"} \
# M_AXI_HPM1_FPD {memport "M_AXI_GP"} \
# M_AXI_HPM0_LPD {memport "M_AXI_GP"} \
# S_AXI_HPC0_FPD {memport "S_AXI_HPC" sptag "HPC0"} \
# S_AXI_HPC1_FPD {memport "S_AXI_HPC" sptag "HPC1"} \
# S_AXI_HP0_FPD {memport "S_AXI_HP" sptag "HP0"} \
# S_AXI_HP1_FPD {memport "S_AXI_HP" sptag "HP1"} \
# S_AXI_HP2_FPD {memport "S_AXI_HP" sptag "HP2"} \
# S_AXI_HP3_FPD {memport "S_AXI_HP" sptag "HP3"} \
# } [get_bd_cells /ps_e]
# set intVar []
# for {set i 0} {$i < 8} {incr i} {
# lappend intVar In$i {}
# }
# set_property PFM.IRQ $intVar [get_bd_cells /xlconcat_0]
# set_property PFM.IRQ $intVar [get_bd_cells /xlconcat_1]
```

Implementing the Hardware Platform Design

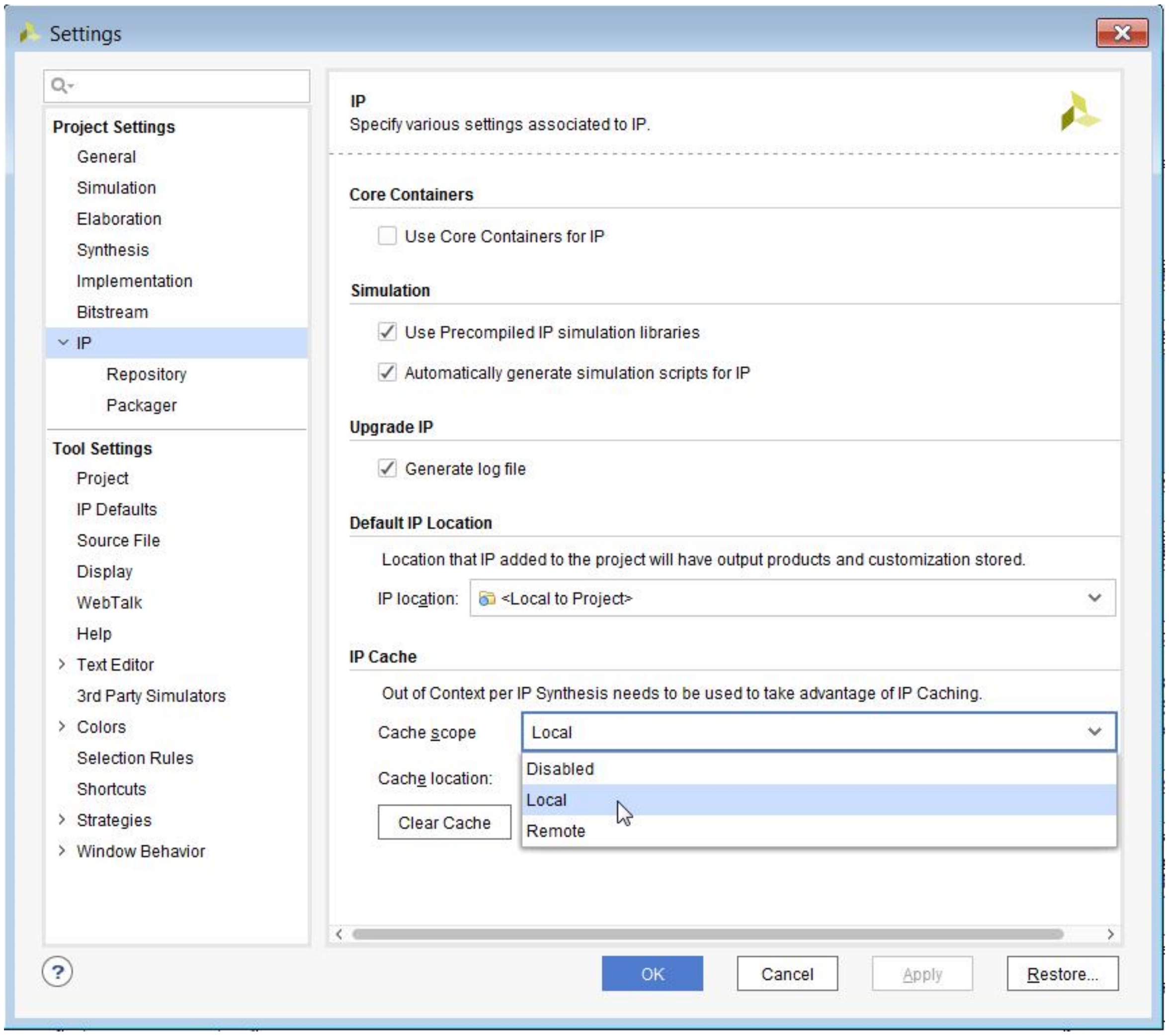
The hardware platform design should be implemented and validated to ensure it works as expected in the Xilinx SDSoC flow. The first step in that validation process should be to ensure the hardware platform design itself is performing as expected. This can be done using test kernel logic to populate the dynamic region.

Using the IP Cache

Significant synthesis run-time savings can be achieved by taking advantage of the IP caching capabilities in Vivado synthesis. IP caching stores the synthesis results for each IP configuration and uses the cached results in place of re-synthesizing the IP during output generation, and for additional IP instances that have matching configurations.

In order for the IP to be cached successfully for use in the DSA, the Vivado Settings need to be configured so the Cache location is local to the Vivado project prior to generating the IP integrator block design. This is the default setting, as shown in the following figure.

Figure 45: **Vivado Settings - IP Cache**



IP
Specify various settings associated to IP.

Core Containers

- [ ] Use Core Containers for IP

Simulation

- [x] Use Precompiled IP simulation libraries
- [x] Automatically generate simulation scripts for IP

Upgrade IP

- [x] Generate log file

Default IP Location

Location that IP added to the project will have output products and customization stored.

IP location: <Local to Project>

IP Cache

Out of Context per IP Synthesis needs to be used to take advantage of IP Caching.

Cache scope: Local

Disabled
Local
Remote

Cache location:

Clear Cache

Setting the IP caching repository involves pointing to the IP cache repository. Use the following Tcl command to set the cache prior to creating the DSA.

```
set_property dsa.ip_cache_dir [get_property ip_output_repo \
[current_project]] [current_project]
```

Creating Design Constraints

This section discusses the various types of physical constraints that are needed to support the hardware platform.

Timing Constraints

Timing constraints are specified using the same methods for any Vivado design project. At a minimum, constraints need to be defined for all clocks. Refer to the *Vivado Design Suite User Guide: Using Constraints* (UG903) for more information.

I/O and Clock Constraints

One of the key considerations in the design of a DSA is to identify the I/O interfaces necessary for the board requirements. The Processing System related I/Os are fixed, but any external interfaces from the programmable logic (PL) need to have I/O constraints assigned to drive the implementation tools. The physical I/O locations will influence performance and must be considered as part of the platform planning process.

Refer to the *Vivado Design Suite User Guide: I/O and Clock Planning* (UG899) for more information on I/O and clock planning.

Simulating the Design

The Vivado Design Suite has extensive logic simulation capabilities to enable block or system level validation of the design. Available third party FPGA simulation tools are also supported. Refer to the *Vivado Design Suite User Guide: Logic Simulation* (UG900) for more information.

Implementation and Timing Validation

The design should be synthesized and implemented to ensure desired performance is achieved. It is often required to iterate on floorplanning and implementation strategies to ensure optimal performance.

It is often important to implement, analyze, and iterate on the hardware platform design to ensure that it continues to meet timing during kernel implementation. Using a test kernel, implement the design and then check that the design meets timing by opening the Implemented Design.

The floorplan can be examined and modified if need be to optimize implementation results. Refer to the *Vivado Design Suite User Guide: Design Analysis and Closure Techniques* (UG906) for more information.

Generating a Device Support Archive

After completing your hardware platform design, setting the PFM properties, and generating a valid bitstream using the Vivado Design Suite, you are ready to create a Device Support Archive (DSA) file for use with the SDSoC Development Environment. The DSA is a single-file that captures the complete hardware platform design, to be used in creating an SDSoC platform project.

IMPORTANT! *After creating the DSA file you should retain the source Vivado Design Suite project files so you can recreate or update the DSA file as needed. You can archive the project using the* `archive_project` *Vivado Tcl command.*

Once the required properties have been set, you generate a DSA file using the `write_dsa` command from the Tcl console in the Vivado tool:

```
write_dsa <filename>.dsa -include_bit
```

This creates an archive of the hardware platform that contains all the relevant files and data needed by the SDSoC Development Environment. The `write_dsa` command will also create a bitstream file if one has not yet been created.

The syntax and short help for the `write_dsa` is shown below::

```
write_dsa  [-force] [-include_bit] [-include_emulation] [-legacy] [-minimal]
           [-quiet] [-verbose] [<file>]

Returns:
The name of the dsa file

Usage:
  Name                       Description
  ---------------------------------
  [-force]                   Overwrite existing device support
                             archive file
  [-include_bit]             Include bit file(s) in the dsa.
  [-include_emulation]       Generate and include hardware
                             emulation support in the dsa.
  [-legacy]                  Write a legacy DSA (based on OCL
                             Block IP)
  [-minimal]                 Add only minimal files in the dsa.
  [-quiet]                   Ignore command errors
  [-verbose]                 Suspend message limits during
                             command execution
  [<file>]                   Device Support Archive filename
                             with alphanumeric characters and
                             .dsa extension.
```

Validating the DSA

You can use the `validate_dsa` command to validate a custom DSA file to ensure it contains the proper content and metadata needed to support the hardware platform in the Xilinx SDSoC™ environment. Use the following command to validate a DSA file:

```
validate_dsa <dsa file> -verbose
```

Creating the Platform Software Component

Introduction

The software components of an SDSoC™ platform can be generated directly from the SDx™ IDE. Using the DSA as the input hardware specification an SDx Platform project can be configured to generate the software files necessary for standalone, FreeRTOS, or Linux targets. For Linux targets, the SDx IDE invokes the PetaLinux tools to build the Linux image and the necessary software object files for constructing an SDSoC platform. The DSA file used by the SDx IDE is created in the Vivado® Design Suite using the `write_dsa` command.

Developers can also continue to create software with the Xilinx® SDK for standalone and FreeRTOS applications and utilize the PetaLinux tools to create Linux images and applications to construct an SDSoC platform through the SDx Platform project flow. An HDF file is used by the software creation tools as an input that describes the hardware design. To generate an HDF file, in Vivado, use the **File→Export→Export Hardware** command.

IMPORTANT! *Because of the different configuration requirements for the different tools, such as the Vivado Design Suite and PetaLinux, running the tools in separate terminal shells is the recommended practice.*

The software platform data creation process consists of building software components, such as libraries and header files, boot files, and others, for each supported operating system (OS) running on the device, and generating a software platform metadata file (`.spfm`) that captures how the components are used and where they are located. The platform folder `<path_to_platform>/sw` contains the software components, while the software platform metadata file is found in `<path_to_platform>/sw/<platform>.spfm`.

Every software platform should also include one or more sample designs that provide example usage.

This chapter describes required and optional components of a software platform, and assumes the platform creator is able to create these components. For example, if your platform supports Linux, you will need:

- Boot files - first stage bootloader or FSBL; U-boot; Linux FIT image `image.ub` or separate `devicetree.dtb`, kernel and ramdisk files; boot image file or BIF used to create `BOOT.BIN` boot files.

- Optional prebuilt data used by SDSoC when building applications without hardware accelerators, such as a pre-generated hardware bitstream and SDSoC data files to reduce compile time.
- Optional header and library files if the platform provides software libraries.
- Optional emulation data files, if the platform supports emulation flows using the Vivado Simulator for programmable logic and QEMU for the processing subsystem.

If your platform supports the Xilinx Standalone OS (a bare-metal board support package or BSP), the software components are similar to those for Linux, but the boot files include the FSBL and BIF files.

TIP: *Zynq® UltraScale+™ MPSoC boot files also require ELF files for the Platform Management Unit firmware (PMUFW) and Arm® Trusted firmware (ATF).*

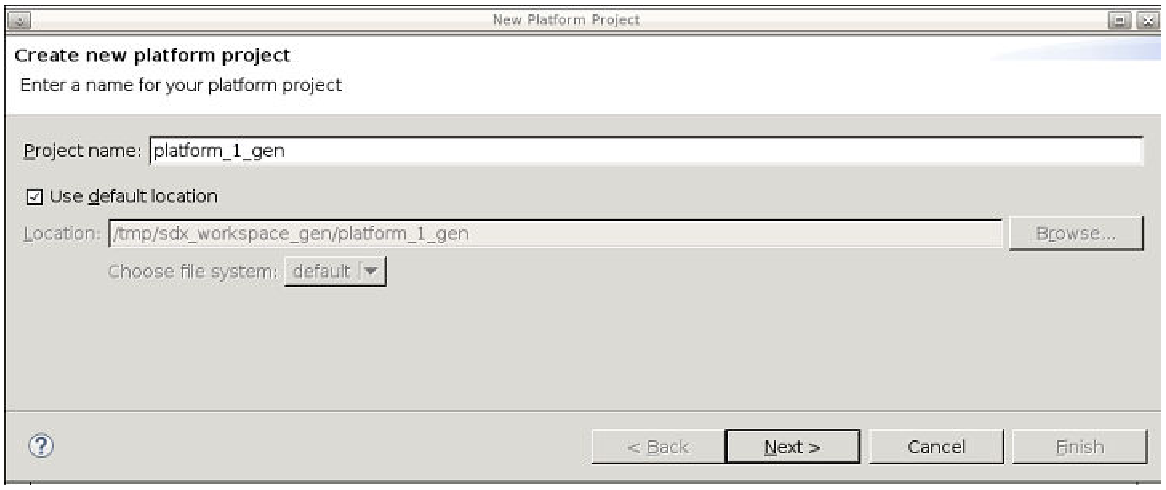
Once you build the software components for a target OS, use the SDSoC platform project to add these components to the platform as described in Creating an SDSoC Platform Project.

Begin with an SDx Platform Project

In this chapter, two platform projects are created to illustrate the generation of standalone and Linux software objects and files using the SDx IDE. The first platform shows what files are needed and used in creating a platform for standalone use. The second platform illustrates the same flow but for the case of a platform that runs Linux on the target hardware. In general, a Linux and standalone system configuration can coexist in a single platform and they are not required to be in separate platform projects.

After launching the SDx IDE, define a new workspace (`sdx_workspace_gen`) for this example. This workspace contains the two platform projects for which the SDx tools will generate the software components. Use the Welcome screen or the SDx menu bar by selecting **File → New → SDx Platform Project** to create the first platform for a standalone target. The **New Platform Project** dialog opens and prompts you for a project name. Name the project `platform_1_gen` for this ZCU102-based example.

Figure 46: **New Platform Project**


New Platform Project

Create new platform project

Enter a name for your platform project

Project name: platform_1_gen

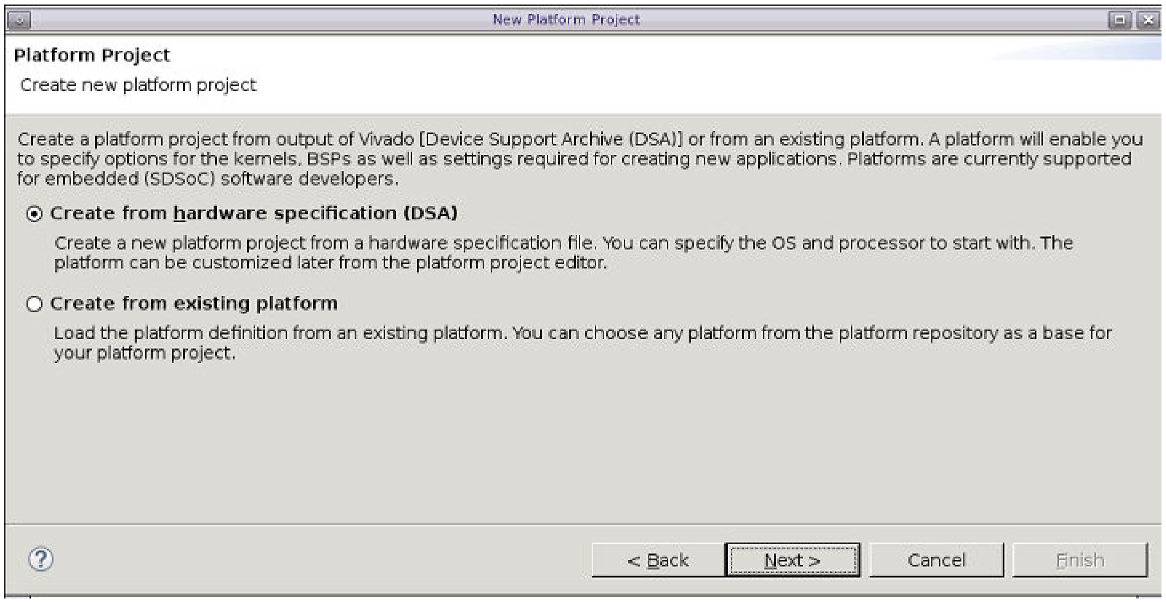
[✓] Use default location

Location: /tmp/sdx_workspace_gen/platform_1_gen

Choose file system: default

Click **Next** to advance to the next dialog and select the source of the hardware specification for the platform. You can choose to use a DSA or an existing platform as the source for the hardware component of the platform. Select DSA for this example.

Figure 47: **New Platform Source - DSA**


New Platform Project

Platform Project

Create new platform project

Create a platform project from output of Vivado [Device Support Archive (DSA)] or from an existing platform. A platform will enable you to specify options for the kernels, BSPs as well as settings required for creating new applications. Platforms are currently supported for embedded (SDSoC) software developers.

● Create from hardware specification (DSA)

Create a new platform project from a hardware specification file. You can specify the OS and processor to start with. The platform can be customized later from the platform project editor.

○ Create from existing platform

Load the platform definition from an existing platform. You can choose any platform from the platform repository as a base for your platform project.

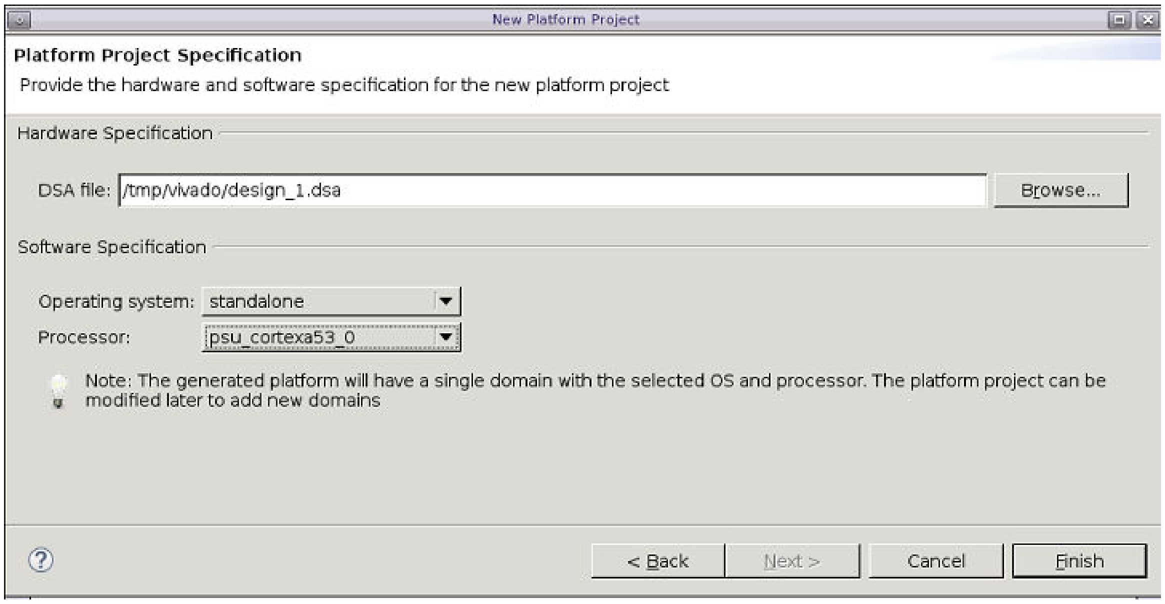
Click **Next** to specify the DSA filename as the **Hardware Specification**. Use the DSA generated in the example from the Chapter 3: Creating the Platform Hardware Component chapter. For this example a copy of the generated DSA is located at `/tmp/vivado/design_1.dsa`.

Click on **Generate Platform** on the **Quick Links** selections. The platform generation process takes about 10 minutes to produce all the software output products for the Standalone target.

Standalone

A standalone target provides software applications complete access to the hardware design within the platform. This is also referred to as "bare-metal" since there are no layers of protection between software applications and the underlying hardware.

Figure 48: **Platform Hardware Specification**


New Platform Project

Platform Project Specification

Provide the hardware and software specification for the new platform project

Hardware Specification

DSA file: /tmp/vivado/design_1.dsa

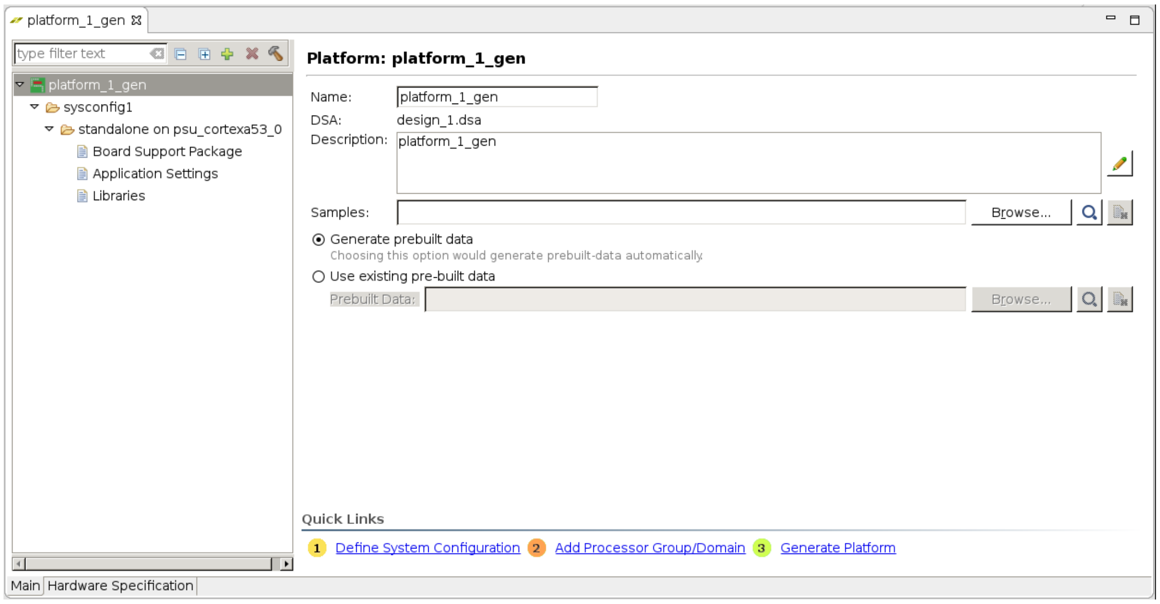
Software Specification

Operating system: standalone

Processor: psu_cortexa53_0

Note: The generated platform will have a single domain with the selected OS and processor. The platform project can be modified later to add new domains

Figure 49: **Platform Configuration Settings - Standalone**


platform_1_gen

- platform_1_gen
  - sysconfig1
    - standalone on psu_cortexa53_0
      - Board Support Package
      - Application Settings
      - Libraries

Platform: platform_1_gen

Name: platform_1_gen

DSA: design_1.dsa

Description: platform_1_gen

Samples:

Generate prebuilt data (selected)

Choosing this option would generate prebuilt-data automatically.

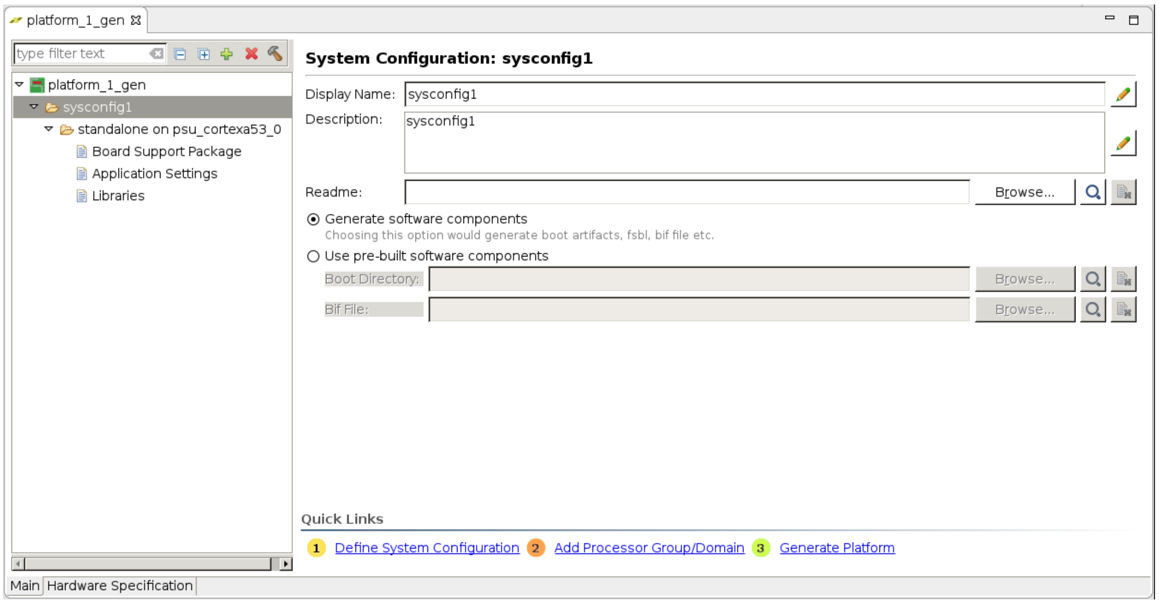
Use existing pre-built data

Prebuilt Data:

Quick Links

1 Define System Configuration 2 Add Processor Group/Domain 3 Generate Platform

Figure 50: **System Configuration Settings - Standalone**



System Configuration: sysconfig1

Display Name: sysconfig1

Description: sysconfig1

Readme:

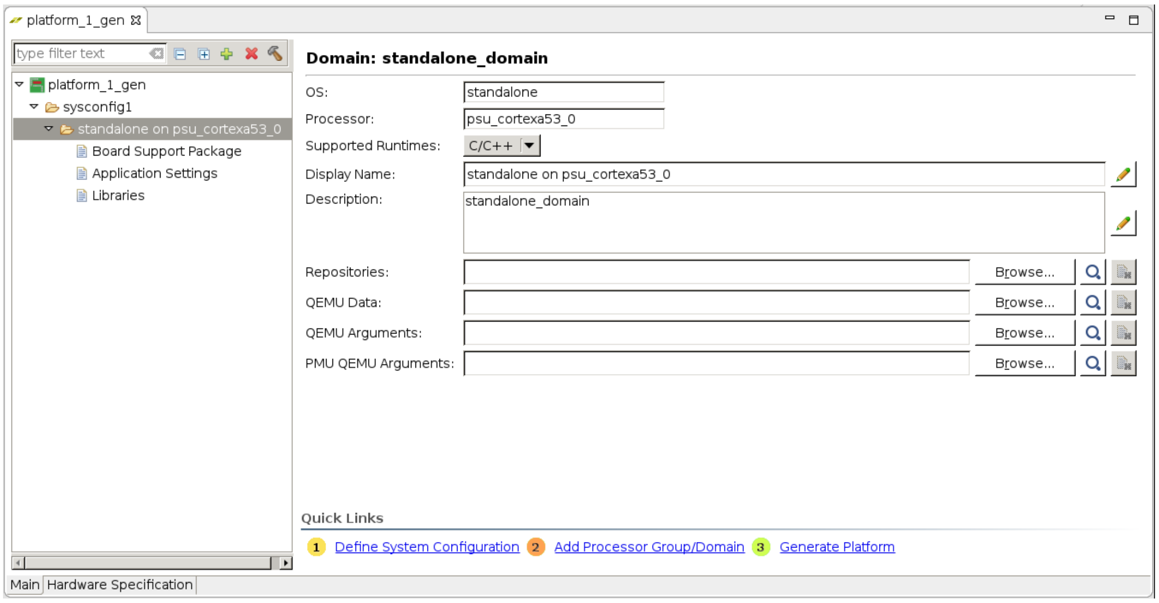
(●) Generate software components
Choosing this option would generate boot artifacts, fsbl, bif file etc.

( ) Use pre-built software components

Boot Directory:

Bif File:

Figure 51: **Domain Configuration Settings - Standalone**



Domain: standalone_domain

OS: standalone

Processor: psu_cortexa53_0

Supported Runtimes: C/C++

Display Name: standalone on psu_cortexa53_0

Description: standalone_domain

Repositories:

QEMU Data:

QEMU Arguments:

PMU QEMU Arguments:

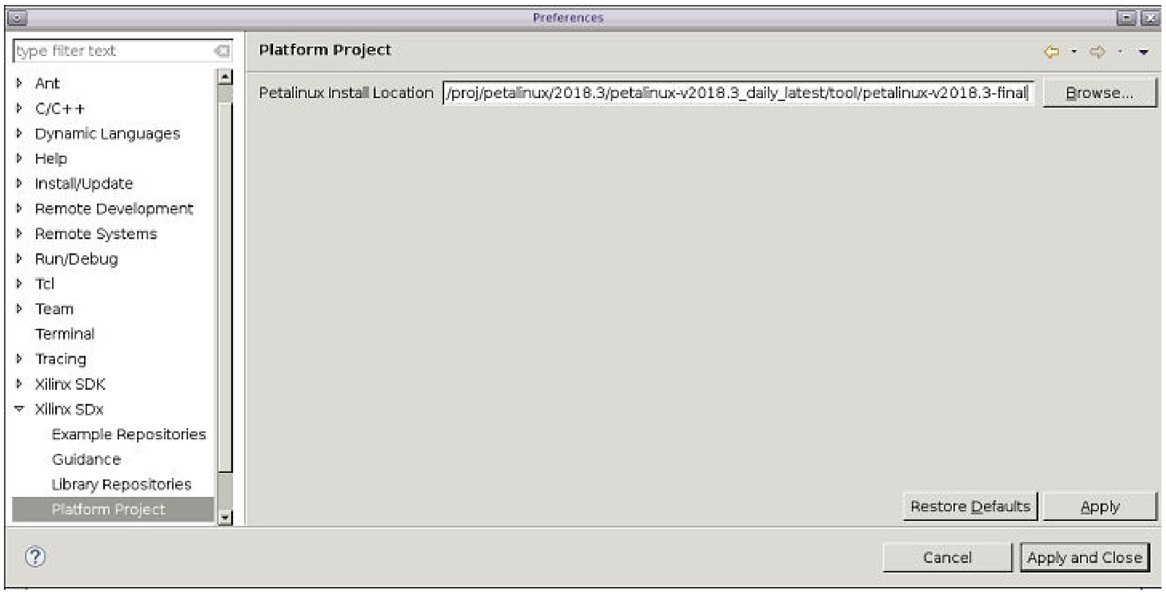
Linux

A Linux target offers multi-tasking, virtual memory, and a variety of drivers to support many different hardware interfaces. Multiple software applications can run simultaneously or appear to do so through the Linux scheduler.

The SDx IDE can also generate the software files needed for a Linux target. Using the same `sdx_workspace_gen` workspace a new SDx Platform project named `platform_2_gen` is illustrated below. The SDx IDE generates the required Linux software objects and files by invoking the PetaLinux tools. Prior to creating the Linux platform project it is necessary to set a path to the PetaLinux tools through the SDx IDE.

Setting the PetaLinux path is accomplished through the SDx menu bar using **Window → Preferences → Xilinx SDx → Platform Project**. Click on **Apply and Close** to save settings.

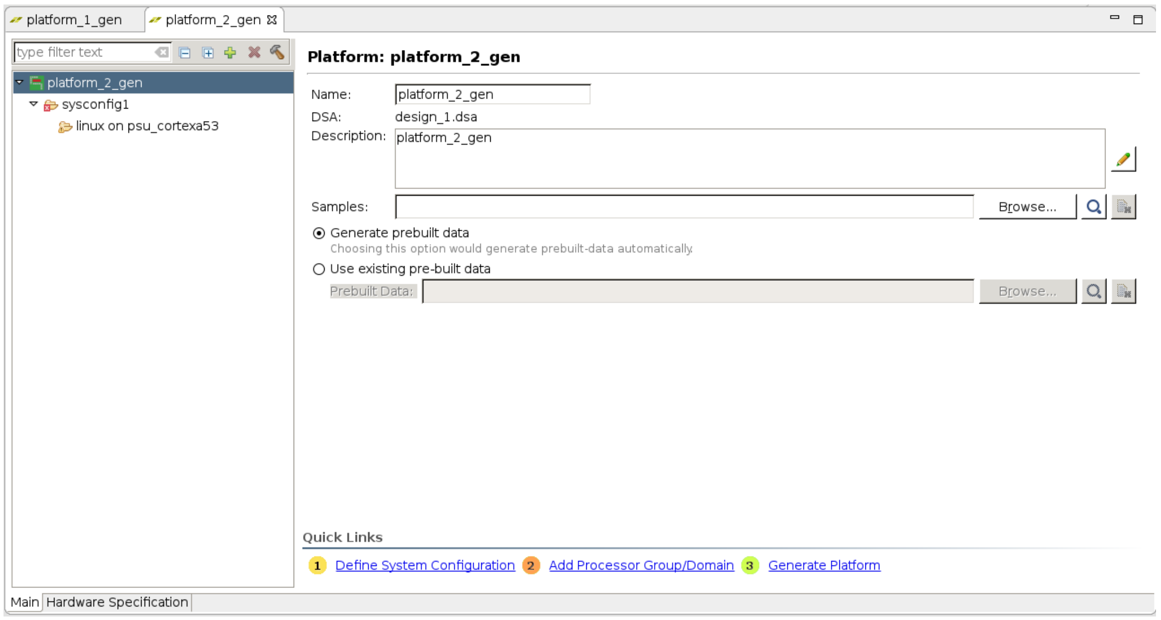
Figure 52: **PetaLinux Path**



Platform Project

Petalinux Install Location: /proj/petalinux/2018.3/petalinux-v2018.3_daily_latest/tool/petalinux-v2018.3-final

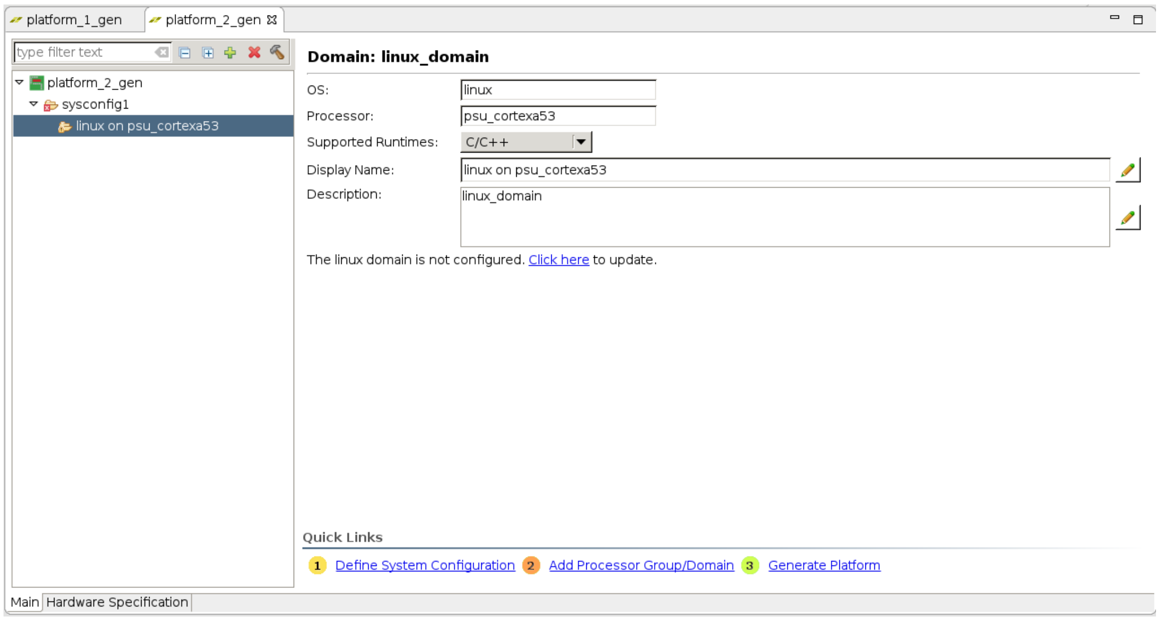
Figure 53: **Platform Configuration Settings - Linux**



Platform: platform_2_gen

Name: platform_2_gen
DSA: design_1.dsa
Description: platform_2_gen
Samples:
(●) Generate prebuilt data
Choosing this option would generate prebuilt-data automatically.
( ) Use existing pre-built data
Prebuilt Data:

Figure 54: **Linux Domain Settings**

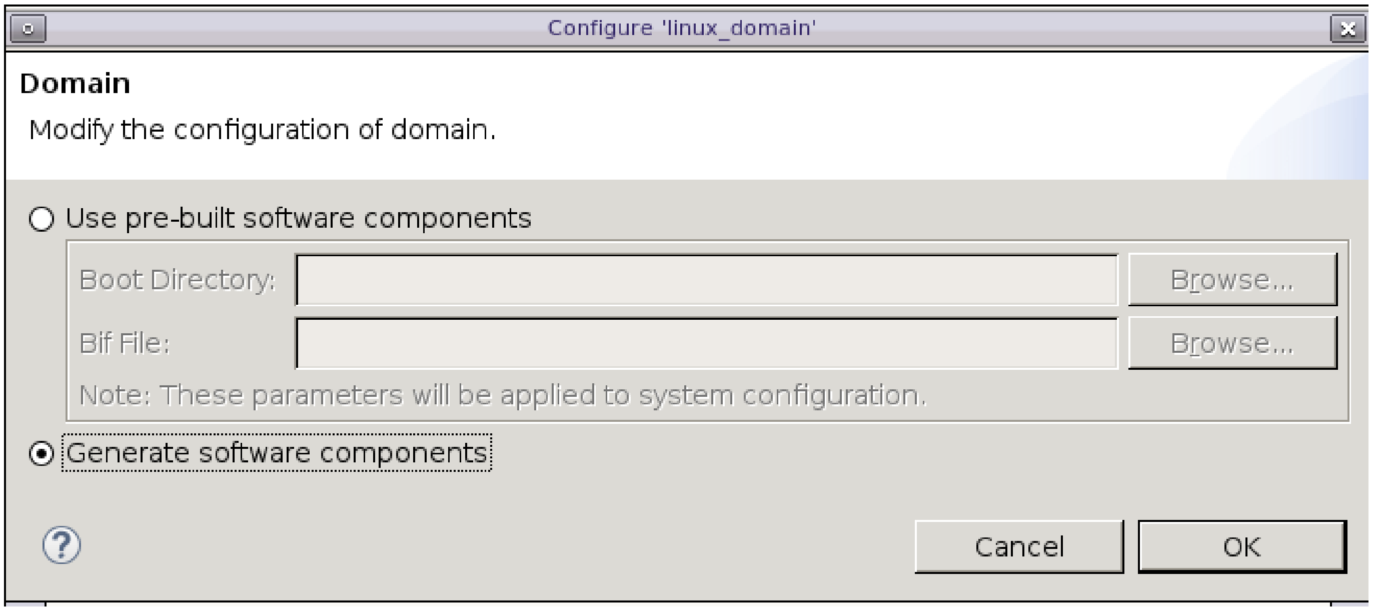


Domain: linux_domain

OS: linux
Processor: psu_cortexa53
Supported Runtimes: C/C++
Display Name: linux on psu_cortexa53
Description: linux_domain

The linux domain is not configured. Click here to update.

Figure 55: **Generate Software - Linux**


Configure 'linux_domain'

Domain

Modify the configuration of domain.

( ) Use pre-built software components

Boot Directory:

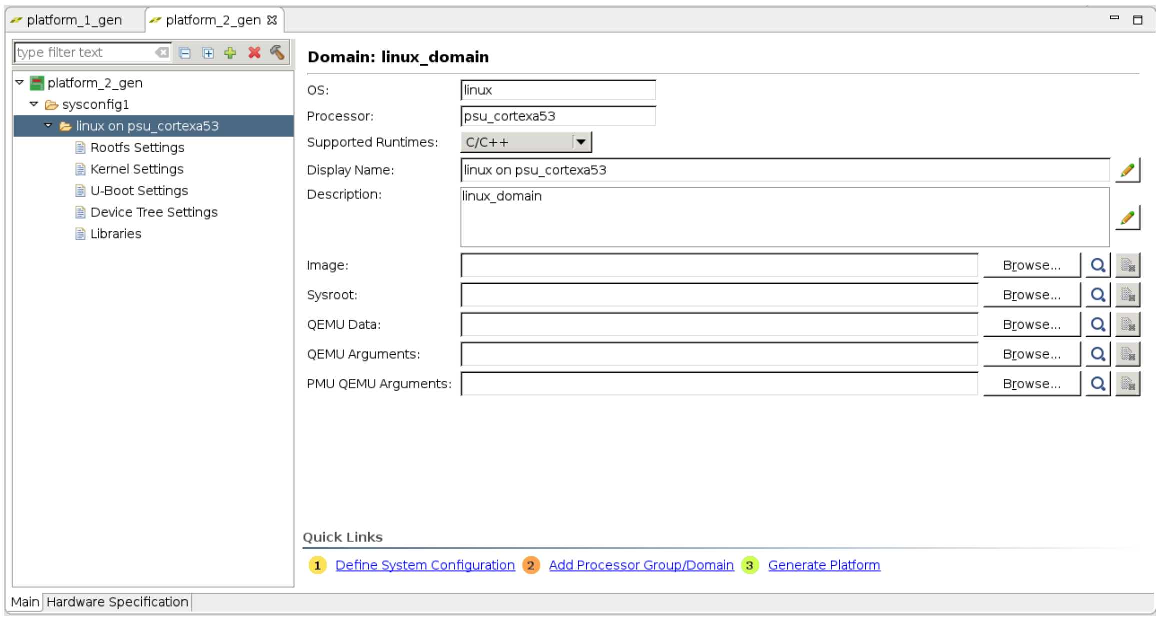
Bif File:

Note: These parameters will be applied to system configuration.

(●) Generate software components

As part of the Linux software generation process, a root file system that includes the `libsds_lib.so` shared library is placed in the `/usr/lib` directory. This can be viewed by using the SDx **Project Explorer** and navigating to `<platform_name>/export/<platform_name>/sw/<system_name>/<domain_name>/sysroot`. For our Linux `platform_2_gen` example this translates to `platform_2_gen/export/platform_2_gen/sw/sysconfig1/linux_domain/sysroot`. Platform developers should be aware that this previously statically linked library is now a dynamically linked library that must be included in the Linux file system image that runs on a board.

Figure 56: **Linux Domain**



platform_1_gen | platform_2_gen

- platform_2_gen
  - sysconfig1
    - linux on psu_cortexa53
      - Rootfs Settings
      - Kernel Settings
      - U-Boot Settings
      - Device Tree Settings
      - Libraries

Domain: linux_domain

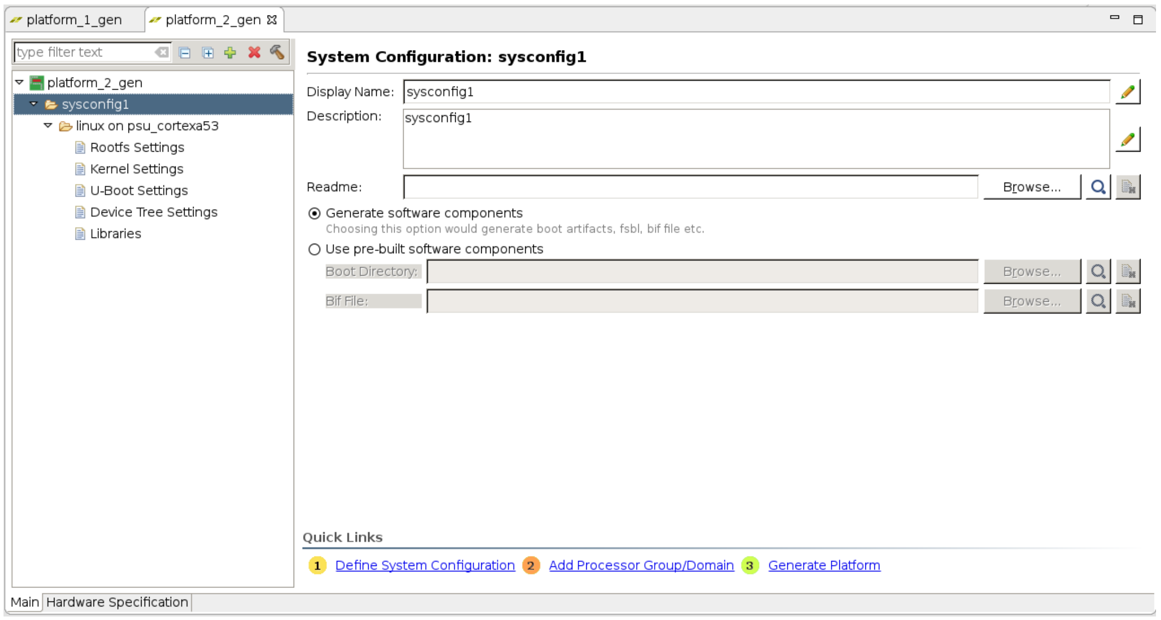
OS:	linux
Processor:	psu_cortexa53
Supported Runtimes:	C/C++
Display Name:	linux on psu_cortexa53
Description:	linux_domain
Image:	
Sysroot:	
QEMU Data:	
QEMU Arguments:	
PMU QEMU Arguments:	

Quick Links

1 Define System Configuration 2 Add Processor Group/Domain 3 Generate Platform

Main | Hardware Specification

Figure 57: **System Configuration**



platform_1_gen | platform_2_gen

- platform_2_gen
  - sysconfig1
    - linux on psu_cortexa53
      - Rootfs Settings
      - Kernel Settings
      - U-Boot Settings
      - Device Tree Settings
      - Libraries

System Configuration: sysconfig1

Display Name:	sysconfig1
Description:	sysconfig1
Readme:	

● Generate software components
Choosing this option would generate boot artifacts, fsbl, bif file etc.

○ Use pre-built software components

Boot Directory:

Bif File:

Quick Links

1 Define System Configuration 2 Add Processor Group/Domain 3 Generate Platform

Main | Hardware Specification

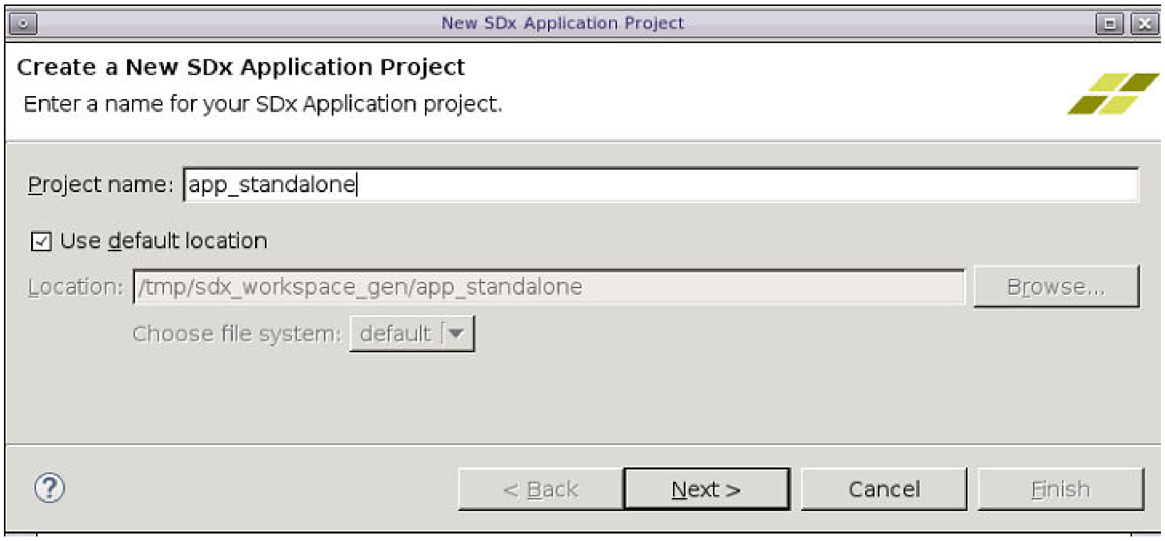
Click on **Generate Platform** on the **Quick Links** selections. The PetaLinux tools are invoked and the generated software output products build time is approximately an hour for this example.

You have now successfully generated two SDSoC platforms based on a custom ZCU102 DSA and the SDx tools created the necessary software components for the platforms. One set for a standalone target (`platform_1_gen`) and another set for a Linux target (`platform_2_gen`).

Build an SDx Application - Standalone

As a part of the SDx environment a set of application code templates are available to test platforms or explore hardware acceleration features. Using the same workspace where you generated the custom platforms for Standalone and Linux targets, two SDx Application projects are added. The `Array Partitioning` template is used for both targets and creates an SD card image for testing the platforms on a ZCU102 board.

Figure 58: **Application Project**


New SDx Application Project

Create a New SDx Application Project

Enter a name for your SDx Application project.

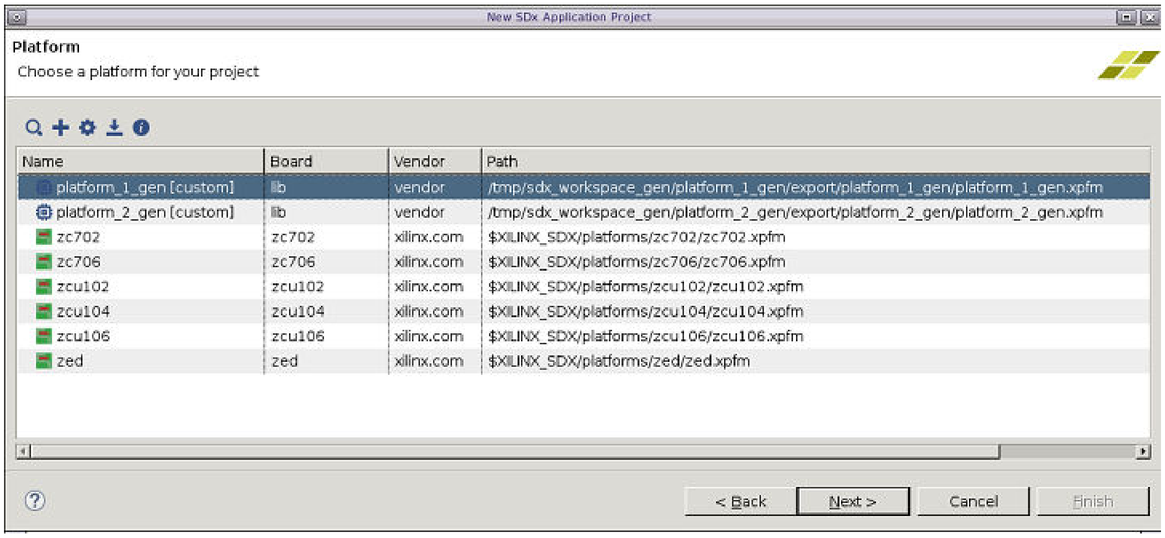
Project name: app_standalone

☑ Use default location

Location: /tmp/sdx_workspace_gen/app_standalone

Choose file system: default

Figure 59: **Platform Selection**

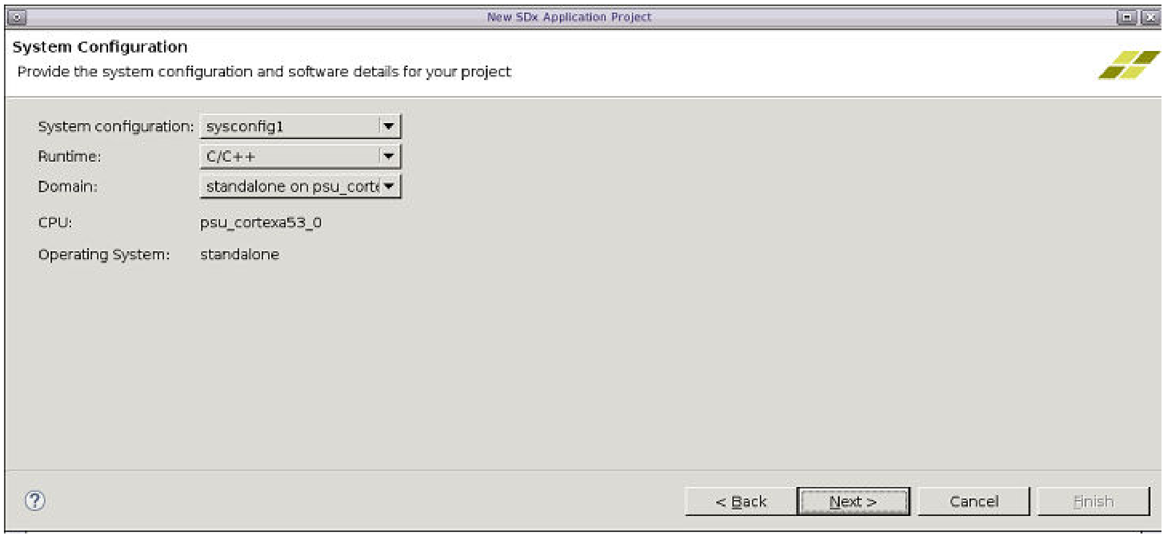

New SDx Application Project

Platform

Choose a platform for your project

Name	Board	Vendor	Path
platform_1_gen [custom]	lib	vendor	/tmp/sdx_workspace_gen/platform_1_gen/export/platform_1_gen/platform_1_gen.xpfm
platform_2_gen [custom]	lib	vendor	/tmp/sdx_workspace_gen/platform_2_gen/export/platform_2_gen/platform_2_gen.xpfm
zc702	zc702	xilinx.com	$XILINX_SDX/platforms/zc702/zc702.xpfm
zc706	zc706	xilinx.com	$XILINX_SDX/platforms/zc706/zc706.xpfm
zcu102	zcu102	xilinx.com	$XILINX_SDX/platforms/zcu102/zcu102.xpfm
zcu104	zcu104	xilinx.com	$XILINX_SDX/platforms/zcu104/zcu104.xpfm
zcu106	zcu106	xilinx.com	$XILINX_SDX/platforms/zcu106/zcu106.xpfm
zed	zed	xilinx.com	$XILINX_SDX/platforms/zed/zed.xpfm

Figure 60: **System Configuration**



New SDx Application Project

System Configuration

Provide the system configuration and software details for your project

System configuration: sysconfig1

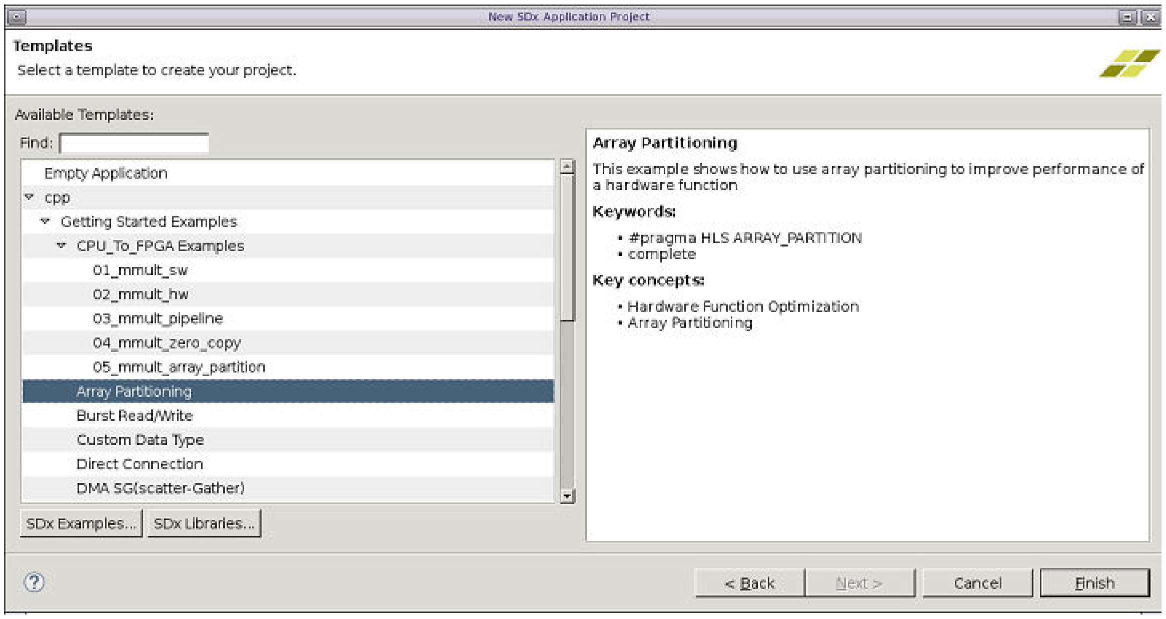
Runtime: C/C++

Domain: standalone on psu_corte

CPU: psu_cortexa53_0

Operating System: standalone

Figure 61: **Array Partitioning Template Application**



New SDx Application Project

Templates

Select a template to create your project.

Available Templates:

Find:

- Empty Application
- cpp
  - Getting Started Examples
    - CPU_To_FPGA Examples
      - 01_mmult_sw
      - 02_mmult_hw
      - 03_mmult_pipeline
      - 04_mmult_zero_copy
      - 05_mmult_array_partition
    - Array Partitioning
    - Burst Read/Write
    - Custom Data Type
    - Direct Connection
    - DMA SG(scatter-Gather)

Array Partitioning

This example shows how to use array partitioning to improve performance of a hardware function

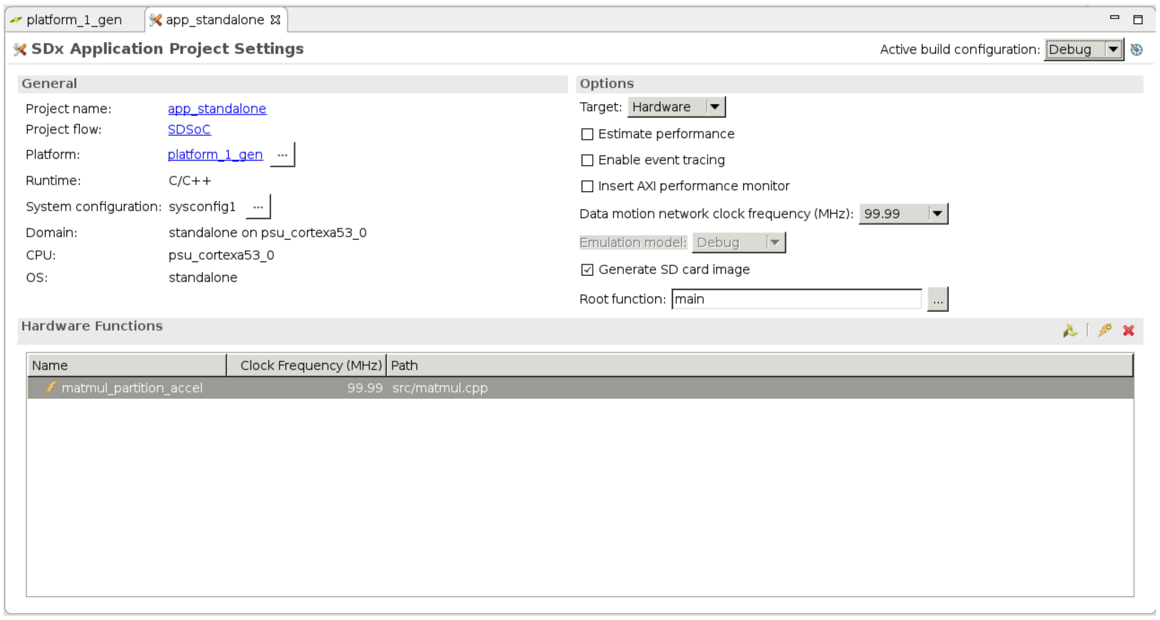
Keywords:

- #pragma HLS ARRAY_PARTITION
- complete

Key concepts:

- Hardware Function Optimization
- Array Partitioning

Figure 62: **Application Settings**



platform_1_gen | app_standalone

SDx Application Project Settings

Active build configuration: Debug

General

Project name:	app_standalone
Project flow:	SDSoC
Platform:	platform_1_gen
Runtime:	C/C++
System configuration:	sysconfig1
Domain:	standalone on psu_cortexa53_0
CPU:	psu_cortexa53_0
OS:	standalone

Options

Target: Hardware

☐ Estimate performance

☐ Enable event tracing

☐ Insert AXI performance monitor

Data motion network clock frequency (MHz): 99.99

Emulation model: Debug

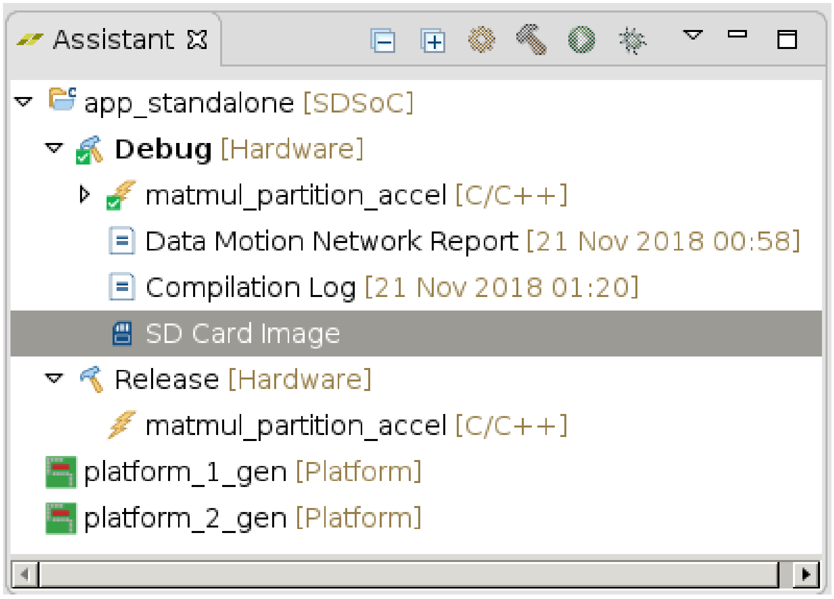
☑ Generate SD card image

Root function: main

Hardware Functions

Name	Clock Frequency (MHz)	Path
matmul_partition_accel	99.99	src/matmul.cpp

Figure 63: **Standalone Build - Assistant View**



```
Assistant
▽ app_standalone [SDSoC]
  ▽ Debug [Hardware]
    ▷ matmul_partition_accel [C/C++]
      Data Motion Network Report [21 Nov 2018 00:58]
      Compilation Log [21 Nov 2018 01:20]
      SD Card Image
  ▽ Release [Hardware]
      matmul_partition_accel [C/C++]
  platform_1_gen [Platform]
  platform_2_gen [Platform]
```

An `Array Partitioning` application project for Linux can also be created for `platform_2_gen` as was done for the Standalone application using `platform_1_gen`. Results of booting and running the contents of the SD Card Image generated by the SDx IDE for both a Standalone target and a Linux target are shown below.

Figure 64: **Linux Build - Assistant View**

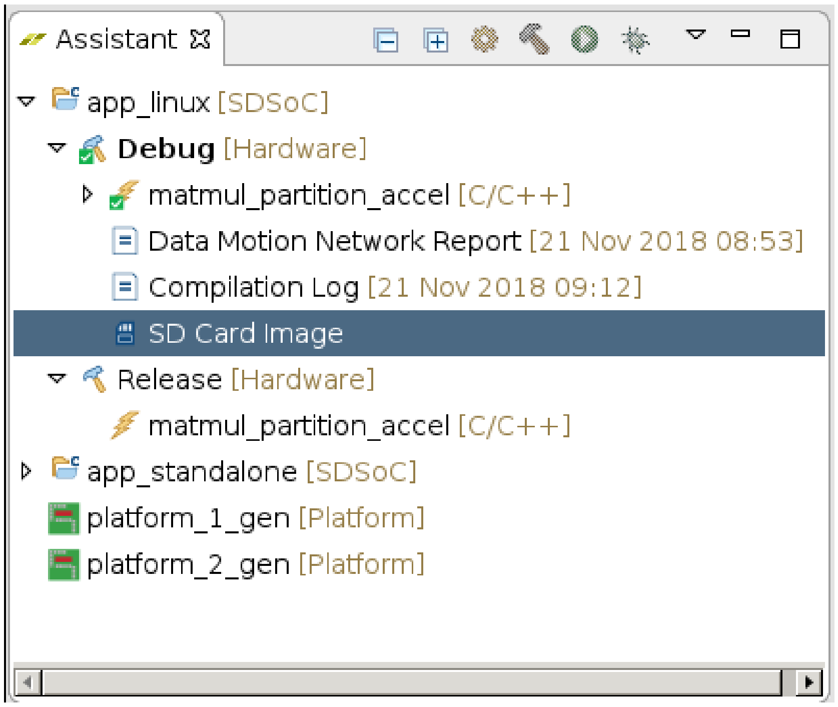
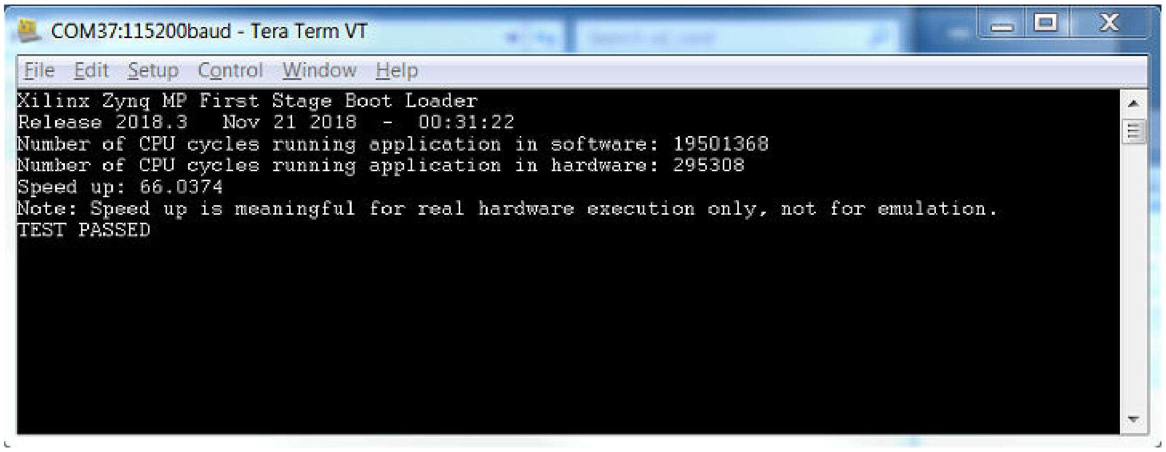

Assistant
app_linux [SDSoC]
Debug [Hardware]
matmul_partition_accel [C/C++]
Data Motion Network Report [21 Nov 2018 08:53]
Compilation Log [21 Nov 2018 09:12]
SD Card Image
Release [Hardware]
matmul_partition_accel [C/C++]
app_standalone [SDSoC]
platform_1_gen [Platform]
platform_2_gen [Platform]

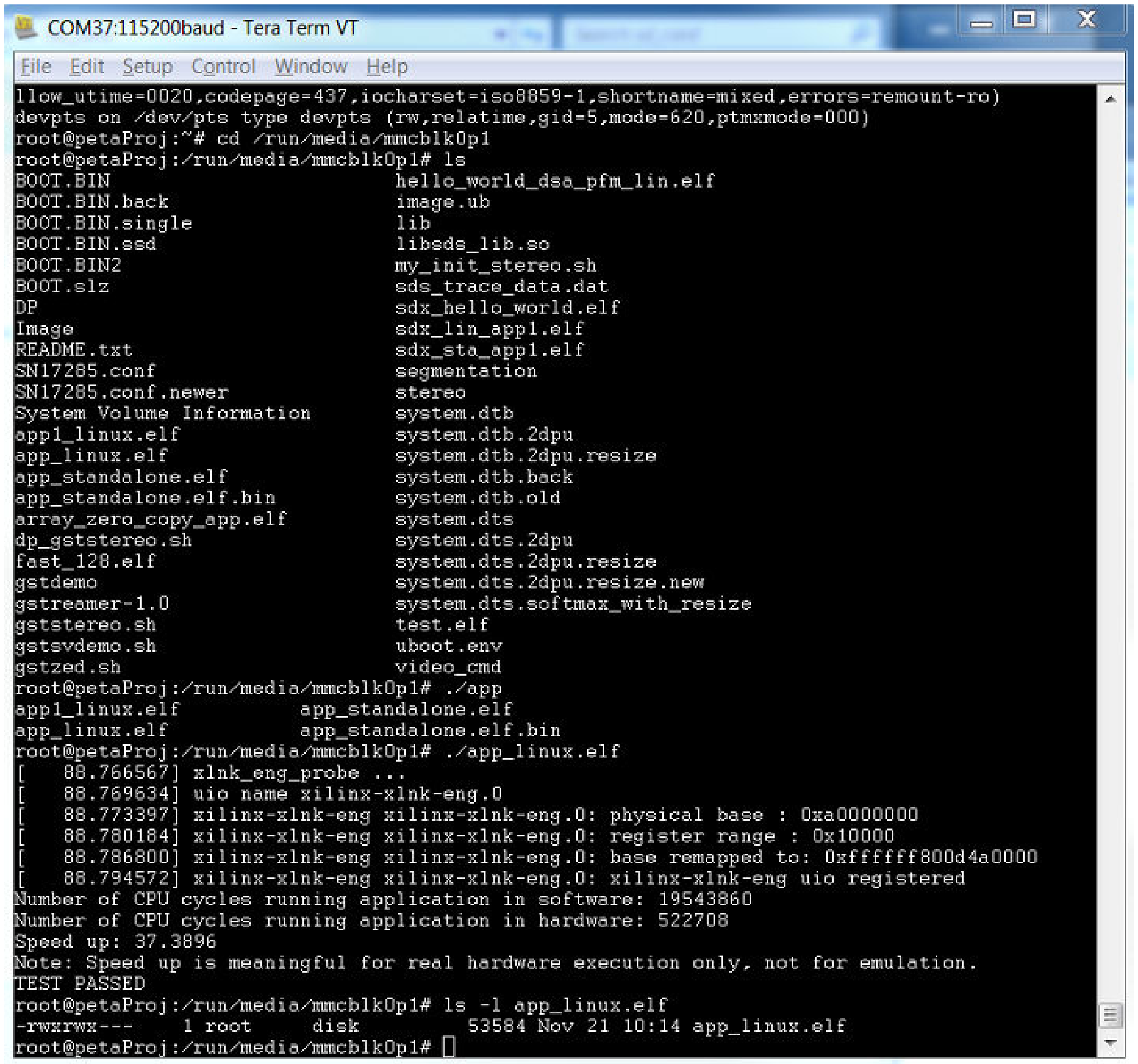
Figure 65: **Application Run Output - Standalone**


COM37:115200baud - Tera Term VT
File Edit Setup Control Window Help

```
Xilinx Zynq MP First Stage Boot Loader
Release 2018.3   Nov 21 2018  -  00:31:22
Number of CPU cycles running application in software: 19501368
Number of CPU cycles running application in hardware: 295308
Speed up: 66.0374
Note: Speed up is meaningful for real hardware execution only, not for emulation.
TEST PASSED
```

Build an SDx Application - Linux

Figure 66: **Application Run Output - Linux**


COM37:115200baud - Tera Term VT
File Edit Setup Control Window Help

```
llow_utime=0020,codepage=437,iocharset=iso8859-1,shortname=mixed,errors=remount-ro)
devpts on /dev/pts type devpts (rw,relatime,gid=5,mode=620,ptmxmode=000)
root@petaProj:~# cd /run/media/mmcblk0p1
root@petaProj:/run/media/mmcblk0p1# ls
BOOT.BIN                        hello_world_dsa_pfm_lin.elf
BOOT.BIN.back                   image.ub
BOOT.BIN.single                 lib
BOOT.BIN.ssd                    libsds_lib.so
BOOT.BIN2                       my_init_stereo.sh
BOOT.slz                        sds_trace_data.dat
DP                              sdx_hello_world.elf
Image                           sdx_lin_appl.elf
README.txt                      sdx_sta_appl.elf
SN17285.conf                    segmentation
SN17285.conf.newer              stereo
System Volume Information       system.dtb
appl_linux.elf                  system.dtb.2dpu
app_linux.elf                   system.dtb.2dpu.resize
app_standalone.elf              system.dtb.back
app_standalone.elf.bin          system.dtb.old
array_zero_copy_app.elf         system.dts
dp_gststereo.sh                 system.dts.2dpu
fast_128.elf                    system.dts.2dpu.resize
gstdemo                         system.dts.2dpu.resize.new
gstreamer-1.0                   system.dts.softmax_with_resize
gststereo.sh                    test.elf
gstsvdemo.sh                    uboot.env
gstzed.sh                       video_cmd
root@petaProj:/run/media/mmcblk0p1# ./app
appl_linux.elf          app_standalone.elf
app_linux.elf           app_standalone.elf.bin
root@petaProj:/run/media/mmcblk0p1# ./app_linux.elf
[   88.766567] xlnk_eng_probe ...
[   88.769634] uio name xilinx-xlnk-eng.0
[   88.773397] xilinx-xlnk-eng xilinx-xlnk-eng.0: physical base : 0xa0000000
[   88.780184] xilinx-xlnk-eng xilinx-xlnk-eng.0: register range : 0x10000
[   88.786800] xilinx-xlnk-eng xilinx-xlnk-eng.0: base remapped to: 0xffffff800d4a0000
[   88.794572] xilinx-xlnk-eng xilinx-xlnk-eng.0: xilinx-xlnk-eng uio registered
Number of CPU cycles running application in software: 19543860
Number of CPU cycles running application in hardware: 522708
Speed up: 37.3896
Note: Speed up is meaningful for real hardware execution only, not for emulation.
TEST PASSED
root@petaProj:/run/media/mmcblk0p1# ls -l app_linux.elf
-rwxrwx---    1 root     disk         53584 Nov 21 10:14 app_linux.elf
root@petaProj:/run/media/mmcblk0p1# 
```

Prebuilt Hardware

A platform can optionally include prebuilt configurations to be used when the platform user does not specify any hardware functions in an SDSoC application. This can save significant run-time as the user should not need to wait for the compilation and implementation of the platform to create a bitstream. The prebuilt bitstream and other required files will be used to configure the hardware when needed.

TIP: *An SDSoC application project with no hardware functions will compile without the prebuilt hardware, but it will take longer. Providing the prebuilt hardware is simply a way to reduce run-time in this situation.*

When defining the SDSoC platform project, as described in Appendix B: SDx IDE Glossary, you can specify the **Prebuilt Data**, which is a folder containing prebuilt hardware information to be included in the platform. The prebuilt hardware will be copied into a subdirectory of the platform software directory as part of generating the SDSoC platform project. Data in the subdirectory is pointed to by metadata in the software platform file (`.spfm`). As shown in Directory Structure for a Typical SDSoC Platform, the path to prebuilt hardware data in an SDSoC platform is:

```
<path_to_platform>/sw/prebuilt
```

The `prebuilt` folder for the ZCU102 platform contains `bitstream.bit`, `zcu102.hdf`, `partitions.xml`, `apsys_0.xml`, `portinfo.c` and `portinfo.h` files.

In the SDx IDE platform project, selecting **Generate prebuilt data** in the Platform configuration settings causes the prebuilt data to be generated automatically when selecting the Quick Link **Generate Platform**. The generation process requires additional time to run, since Vivado synthesis and implementation are run in order to produce a bitstream, which is one of the prebuilt data files. If you created prebuilt data manually, select **Use exisiting prebuilt data** in the Platform configuration settings view and specify a path to the directory containing the manually created data files.

Library Header Files

If the platform requires application code to `#include` platform-specific header files, these should be defined in the platform software description file in a subdirectory relative to the platform directory for the corresponding OS. When defining the SDSoC platform project, you can specify the path to one or more folders containing header files.

For a given `<relative_include_path>` in a platform software description file, the location is:

```
platform/sw/os/os/relative_include_path
```

RECOMMENDED: *If header files are not put in the standard area, users need to point to them using the `-I` switch in the SDSoC environment compile command.*

Static Libraries

If the platform requires users to link against static libraries provided in the platform, these should reside in a subdirectory of the platform directory for the corresponding OS in the platform software description file. When defining the SDSoC platform project, you can specify static libraries to be included as platform software data.

For a given `<relative_lib_path>` in a platform software description file, the location is:

```
<platform_root>/sw/<relative_lib_path>
```

RECOMMENDED: *If static libraries are not put in the standard area, every application needs to point to them using the `-L` option to the `sdscc` link command.*

Linux Boot Files

PetaLinux can generate the Linux boot files for an SDSoC platform using the process documented in *PetaLinux Tools Documentation: Workflow Tutorial* (UG1156). The overall workflow for SDSoC platforms is the same, and the basic steps are outlined below. If you are familiar with the PetaLinux tools, you should be able to complete these steps for Zynq UltraScale+ MPSoC or Zynq-7000 SoC designs.

Platform developers should be aware that the previously statically linked library is now a dynamically linked library (`libsds_lib.so`) that must be included in the Linux file system image that runs on a board. As part of the Linux software generation process, a root file system that includes the `libsds_lib.so` shared library is placed in the `/usr/lib` directory. When running the PetaLinux tools manually this library must be included as part of the root file system. The libraries can be found in `<SDX_Install_Dir>/target/<architecture>-linux/lib`. Refer to *PetaLinux Tools Documentation: Reference Guide* (UG1144) for more information on how to include libraries with PetaLinux.

IMPORTANT!

Custom platforms that support Linux must include the SDSoC control API shared libraries. If the shared libraries are not included with the platform, applications targeting the custom platform must explicitly use static linking. Legacy support for static linking is provided by the sds++ -static-sds option, but you lose the capabilities made possible by shared library support, including startup of multiple hardware subsystems without conflicts.

Before starting, you should complete the following:

1. Set up your shell environment with PetaLinux tools in your PATH environment variable.

IMPORTANT! *Because of the different configuration requirements for the different tools, such as the Vivado Design Suite and PetaLinux, running the tools in separate terminal shells is the recommended practice.*

2. Create and `cd` into a working directory.
3. Create a new PetaLinux project targeting a BSP that corresponds to the type of board you are targeting:

```
petalinux-create -t project -n <project_name> \
-s <path_to_base_BSP>
```

4. Obtain a copy of the hardware handoff file (`.hdf`) from the Vivado project for your hardware platform.

IMPORTANT! *This guide assumes the existence of a valid hardware description file (HDF) for the platform, which is generated from the Vivado Design Suite project. Refer to* Chapter 3: Creating the Platform Hardware Component *for more information.*

The steps below include basic setup, loading the hardware handoff file, kernel configuration, root file system configuration, and building the Linux image, fsbl, pmufw, and atf. The steps include the actions to perform, or the PetaLinux command to run, with arguments. Once the build completes, your working directory contains a FIT image file (`image.ub`) that includes the devicetree, kernel and ramdisk. The basic setup is the procedure used to configure the Linux images packaged in all base platforms shipped with SDSoC platforms.

When using the `petalinux-config` command, a text-based user interface appears with a hierarchical menu system. The steps present a hierarchy of commands and the settings to use. Selections with the same indentation are at the same level of hierarchy. For example, the `petalinx-config -c kernel` step asks you to select Device Drivers from the top-level menu, select Generic Driver Options, go down one level to apply settings, go back up to Staging drivers, and apply settings to its sub-menu items.

Building the PetaLinux Image

To build the PetaLinux image, use the following steps:

1. Configure PetaLinux with the HDF derived earlier for the associated platform (the production of which is described in the introduction):

   ```
   petalinux-config -p <petalinux_project> \
   --get-hw-description=<HDF path>
   ```

   Optionally, change boot args to include "quiet" at the end of whatever is the default:

   - Kernel Bootargs→generate boot args automatically (OFF)
   - for Zynq MPSoC: Kernel Bootargs→ user set kernel bootargs (earlycon clk_ignore_unused quiet)
   - for Zynq-7000: Kernel Bootargs→ user set kernel bootargs (console=ttyPS0,115200 earlyprintk quiet)

2. Configure PetaLinux kernel:

   ```
   petalinux-config -p <petalinux_project> \
   -c kernel
   ```

   Set CMA size to be larger, for SDS-alloc buffers:

   - for Zynq UltraScale+ MPSoC: Device Drivers→ Generic Driver Options → Size in Mega Bytes(1024)
   - for Zynq-7000 SoC: Device Drivers→ Generic Driver Options → Size in Mega Bytes(256)

Enable staging drivers:

- Device Drivers → Staging drivers (ON)

Enable APF management driver:

- Device Drivers → Staging drivers → Xilinx APF Accelerator driver (ON)

Enable APF DMA driver:

- Device Drivers → Staging drivers → Xilinx APF Accelerator driver → Xilinx APF DMA engines support (ON)

***Note*:**

For Zynq UltraScale+ MPSoC, you must turn off CPU idle and frequency scaling. To do so, mark the following options:

- **CPU Power Management → CPU idle → CPU idle PM support (OFF)**
- **CPU Power Management → CPU Frequency scaling → CPU Frequency scaling (OFF)**

3. Configure petalinux rootfs:

```
petalinux-config -p <petalinux_project> \
-c rootfs
```

Add stdc++ libs:

- Filesystem Packages → misc → gcc-runtime → libstdc++ (ON)

4. Add device tree fragment for APF driver. At the bottom of `<>/project-spec/meta-user/recipes-bsp/device-tree/files/system-user.dtsi`, add the following entry:

```
/{
 xlnk {
 compatible = "xlnx,xlnk-1.0";
 };
};
```

5. Build the PetaLinux image:

- `petalinux-build`

Preparing the Image for the SDSoC Platform Utility

In the directory `<petalinux_project>/images/linux/` there are a number of important files that are partitioned into two categories:

1. Files that end up compiled into `BOOT.BIN`, referred to collectively as 'boot files', that should be copied into a `boot` folder. Boot files include the following: `u-boot.elf`, `zynq-fsbl.elf` or `zynqmp-fsbl.elf`, along with `bl31.elf` and `pmufw.elf` for Zynq UltraScale+ devices.

2. Files that must reside on the SD card but are not compiled into `BOOT.BIN`, referred to as 'image files', that should be copied into an `image` folder. The only image file from a PetaLinux build is `image.ub`, but you can add other files to the `image` folder that you want to make available to users of the platform.

From within the `<petalinux_project>/images/linux/` folder run the following commands:

```
$ mkdir ./boot
$ mkdir ./image
$ cp u-boot.elf ./boot/u-boot.elf
$ cp *fsbl.elf ./boot/fsbl.elf
$ cp bl31.elf ./boot/bl31.elf
$ cp linux/pmufw.elf ./boot/pmufw.elf
$ cp image.ub ./image/image.ub
```

TIP: *The bl31.elf and pmufw.elf files are only required for for Zynq UltraScale+ devices.*

Finally, create a boot image format, or BIF file, that is used to compile the contents of the `boot` folder into a `BOOT.BIN` file. For more information on creating the BIF file for a target processor, refer to *Zynq-7000 SoC Software Developers Guide* (UG821) or *Zynq UltraScale+ MPSoC Software Developer Guide* (UG1137).

An SDSoC boot image format file looks similar to a standard BIF file, with tokens specified in angle brackets (< >) rather than direct paths to boot files. The BIF file tokens are replaced at SDSoC compile time with actual files and generated content. This is because the bitstream file for the programmable logic (PL) region will be procedurally generated, and some of the elements do not have known file names at the time the BIF file is created.

The following is an example `boot.bif` file for the Zynq-7000 SoC:

```
/* linux */
 the_ROM_image:
 {
   [bootloader]<fsbl.elf>
   <bitstream>
   <u-boot.elf>
 }
```

The following is an example BIF for a Zynq UltraScale+ MPSoC device:

```
the_ROM_image:
{
  [fsbl_config] a53_x64
  [bootloader]<fsbl.elf>
  [pmufw_image]<pmufw.elf>
  [destination_device=pl] <bitstream>
  [destination_cpu=a53-0, exception_level=el-3, trustzone] <bl31.elf>
  [destination_cpu=a53-0, exception_level=el-2] <u-boot.elf>
}
```

Taken together, the `boot` directory, the `image` directory, and the BIF file, constitute the software elements that the SDSoC platform project needs as input for the Linux OS. See Appendix B: SDx IDE Glossary for more information.

Standalone Boot Files

If no OS is required, you can create a standalone boot image (`boot.bin`) that runs the specified executable, along with any necessary boot loaders.

TIP: *If you have already configured the boot files for Linux OS then you can use those same files when creating the standalone boot image format file.*

First Stage Boot Loader (FSBL)

The first stage boot loader (FSBL) is responsible for loading the bitstream and configuring the Zynq and Zynq UltraScale + architecture Processing System (PS) at boot time.

When the platform hardware design is open in the Vivado Design Suite, click the **File → Export → Export Hardware** menu option.

Using the SDx IDE, or the Xilinx Software Development Kit (SDK), create a new Application project **File → New → Application Project** with the name `fsbl`.

Using the exported Hardware Platform, select the Zynq FSBL application from the list. This creates an FSBL executable. For more detailed information, see the SDK Help.

Once you generate the FSBL, you can copy it into a standard location for the SDx environment flow, or you can consume it as part of the process of building a platform project.

Example:

```
samples/platforms/zcu102_axis_io/sw/a53_standalone/boot/fsbl.elf
```

Board Image Format (BIF) File

For the SDx environment to use an executable (ELF) in the boot image, a BIF file must point to it. The following is an example standalone `boot.bif` file for the Zynq-7000 SoC:

```
/* standalone */
the_ROM_image:
 {
   [bootloader]<fsbl.elf>
   <bitstream>
   <elf>
 }
```

The SDx environment replaces the `<bitstream>` and `<elf>` tokens in the BIF file with actual bitstream and ELF file references generated during the SDSoC compilation process.

TIP: *The BIF file for the Zynq UltraScale+ MPSoC device is different from the BIF file for a Zynq-7000 SoC, and requires the addition of pmufw.elf. This file can be generated through SDK or the SDx IDE as a sample targeting the "psu_pmu_0" processor.*

FreeRTOS Configuration/Version Change

SDx support for FreeRTOS is based on the implementation found in the Xilinx Software Development Kit (SDK) tool. By default FreeRTOS v10 is supported and in SDK this corresponds to the most recent `freertos10_xilinx` BSP library.

IMPORTANT! *In the generated SDx platform file (`.spfm`), the processor group contains metadata that specifies the OS name (`sdx:os/sdx:osname`). If `osname` is specified as "freertos", that is mapped to the latest version of `freertos10_xilinx`. If the OS name is specified explicitly as "freertos10_xilinx", the specified version will be used.*

To change FreeRTOS configuration settings, you can use SDx, just as you would use Xilinx SDK, with the platform DSA to create and customize a supported FreeRTOS BSP.

1. Add the include files from the SDK BSP to your platform as library include files (you will define a library include path) when using the SDx IDE to create the platform project.
2. Add the `.mss` file from the SDK BSP to your platform as a BSP configuration file. A linker script can be generated when SDK creates a sample application using the BSP.
3. When you add the linker script to your SDx platform, you must increase the stack and heap sizes because the SDK default values are too small for a typical SDx application.
4. You may also need to increase the task heap size passed to `xTaskCreate` from `configMINIMAL_STACK_SIZE` when creating FreeRTOS applications.

TIP: *This is application dependent, but try 1000 and adjust up or down as appropriate.*

If you want to use a different FreeRTOS version or customize it in a manner that is different from the Xilinx BSP implementations, your can define a System Configuration for the standalone BSP, and add your FreeRTOS implementation as a library. You need to provide a FreeRTOS library, include files and a linker script.

Sample Applications

Optionally, a platform can include sample applications to demonstrate the usage of the platform. Sample applications can be provided in the `samples` directory of a platform. Each sample application is located in a sub-directory of the `samples` directory, as illustrated in Directory Structure for a Typical SDSoC Platform, and is described by a `description.json` file.

The following is an example `description.json` for the `array_copy` sample application in the Arty platform found in the `<SDX_Install_Dir>/samples/platforms/arty/samples/arty_arraycopy` folder.

```
{
        "example": "Arraycopy",
        "overview": "Implementation of an array copy core that simply reads 
from one array and
        writes to another. By default the function arraycopy() is marked 
for hardware and you
        can build the project."
        "board": [ "arty" ],
        "os": [  "Standalone" ],
        "runtime": [  "C/C++" ],
        "accelerators": [
        {
            "name": "arraycopy",
            "location": "arraycopy.cpp"
        }
    ],
    "contributors" : [
        {
            "group" : "Xilinx, Inc.",
            "url" : "http://www.xilinx.com"
        }
    ],
    "revision" : [
        {
            "date" : "2018",
            "version" : "1.0",
            "description" : "Initial revision"
        }
    ]
}
```

The `description.json` file is in `JSON` format and consists of comma-separated key-value pairs, and `JSON` values can be strings, arrays, or nested `JSON` objects. The example above includes the following information:

- Name and description: The example (name) is required, but the overview (description) is recommended.

- Platform: The board is optional, but the OS and runtime are required.
- Accelerators: Required to specify the hardware function name and the location of the source code file.
- Contributors: Optionally identifies the name and description of the source code provider.
- Revision: Optionally specifies version information.

The following table shows a complete list of the available attributes.

Table 1: **Description.JSON Attributes**

Attribute	Description
"example"	Name displayed in the GUI.
"overview"	Description displayed in the GUI.
"type"	Used to identify library projects vs application projects.
"board"	List of supported platforms; if specified, project platform must match an entry in the list.
"os"	List of supported operating systems.
"runtime"	List of supported runtime environments.
"accelerators"	List of hardware function names and source files.
"exclude"	List of files and directories that should not be copied into the example project.
"compiler"	Settings to specify compiler options and include directories.
"linker"	Settings to specify linker options, library paths, and libraries.
"system"	System-level settings.
"cmd_args"	Command line arguments used when launching the GUI.
"revision"	A list of revisions.
"contributors"	A list of contributors/authors for the example.

Example

The `"example"` attribute defines the name of the sample application to show in the SDx™ GUI. It should be descriptive, but not overly long.

```
"example" : "Array Copy"
```

Overview

The `"overview"` attribute is a longer text description of the sample application. The value can be either a single string, or an array of strings.

- Single string:

  ```
  "overview" : "Shows a __median filter__ function accelerated in hardware."
  ```

- Array of strings:

```
"overview" : [
    "This is the first line of the description.",
   "The description supports limited Markdown syntax,",
    "including __bold__, _italics_, and ~~strikethrough~~.",
    "- list item 1",
    "- list item 2",
]
```

TIP: *The description supports limited Markdown syntax, including bold, italic, and numbered or bulleted lists.*

Type

The optional `type` attribute identifies whether this example creates an application project or a library project. If the `type` value is set to `library`, this example creates a shared library project. Otherwise, it creates an application project.

```
"type" : "library"
```

Board

The `board` attribute lists all development board platforms this example is compatible with. If omitted, this example is available for any platform. The sample application will be available if the current platform matches any of the values in the list. If the example is included with a custom platform, the example is available for the custom platform only, and `board` need not be specified.

```
"board" : [
    "zc702",
    "zc706"
]
```

Operating System (OS)

The `os` attribute defines an operating system match for the selected SDSoC™ platform. The `OS` value is a list of supported operating systems including Linux, Standalone, and FreeRTOS. The sample application will be available if the current operating system matches any of the values in the list.

The following example defines an application that can be selected when any of Linux, Standalone, or FreeRTOS are selected as the operating system:

```
"os" : [
    "Linux",
    "Standalone",
    "FreeRTOS"
]
```

Runtime

The `runtime` attribute defines the runtime environments supported by this example is C/C++. The sample application will be available if the run time of the project matches any of the values in the list. For example, an example might work for the OpenCL runtime, and not work for the C/C++ runtime.

The following example defines an application that can be selected when the run time is `C/C++`:

```
"runtime" : [
    "C/C++"
]
```

Accelerators

The optional `Accelerators` attribute is a list of hardware functions that will be set up when creating a new project. The `Accelerators` attribute includes several required and optional sub-tags:

- `name`: A required value that specifies the name of the function.
- `location` A required value that specifies the path to the source file containing the function. The path is relative to the sample application folder in the platform.
- `clkid`: An optional values that specifies the accelerator clock to use instead of the default.
- `hlsfiles`: A optional value that specifies a list of additional files to compile along with the source file, when the accelerator calls code found in other files. The SDSoC environment invokes Vivado® HLS to compile the source file containing the function. If the source file depends on functions contained in additional source files, use `hlsfile` to specify those source files.

The following example specifies two functions to move to hardware `func1` and `func2` when creating the new project:

```
"accelerators" : [
    {
        "name" : "func1",
        "location" : "func1.cpp"
    },
    {
        "name" : "func2",
        "location" : "func2.cpp",
        "clkid” : "2",
        "hlsfiles" : [
            "func2_helper_a.cpp",
            "func2_helper_b.cpp"
        ]
    }
]
```

Compiler

The `compiler` attribute is optional, and is used to specify the following compiler options:

- `includepaths`: Defines a set of paths relative to the sample application folder that are passed to the `sds++` compiler using the `-I` flags.
- `options`: Defines application project settings for the compiler when creating a new project. The value defines compiler options required to build the application and appears in the SDx environment C/C++ Build Settings dialog box as compiler **Inferred Options** under the software platform.
- `symbols`: Defines a list of pre-processor symbols. This is an alternative to `options` when you want the symbols to show in the symbols list in the Eclipse project build settings.

The following example results in the SDSoC environment adding the flags `-I"../src/myinclude" -I"../src/dir/include"`, and the compiler option `-D MYAPPMACRO`, to the `compiler` command:

```
"compiler" : {
    "includepaths" : [
        "myinclude",
        "dir/include"
    ]
    "options" : "-D MYAPPMACRO"
}
```

Linker

The `linker` attribute is optional, and is used to add directories to the link path, and add libraries to be linked. When using multiple linker settings, they should be added to the same `linker` node.

- `librarypaths`: Specifies a list of paths relative to the application build directory (which is the location of the compiled application). The specified paths are passed to the linker using `-L` flags.
- `libraries`: Secifies additional libraries that are to be passed to the linker `-l` flags.
- `options`: Defines application project settings for the linker when creating a new project. The value defines linker options required to build the application and shows in the SDx environment C/C++ Build Settings dialog box as linker **Miscellaneous** options.

  For SDSoC projects, use the `sdcard` attribute c to specify the `sds++ -sdcard <path>` option. The SD card path is relative to the build directory:

The following settings add the flags `-lmylib1 -lmylib2`, and `-L"mylibrary"`, and add the linker option `-poll-mode 1` to the linker command line, and specifies the `-sdcard` path:

```
"linker": {
    "options" : "-poll-mode 1",
    "libraries" : [
        "mylib1"
        "mylib2"
    ],
    "librarypaths" : [
        "mylibrary"
    ]
    "sdcard" : "../sdcard"
}
```

Exclude

The `exclude` attribute defines a set of directories and files to be excluded from being copied when SDx creates the new project. This lets you have files or directories in the sample application folder that are not used to build the application.

The following example will result in SDSoC not making a copy of directories `MyDir` and `MyOtherDir` when creating the new project. It will also not make a copy of files `MyFile.txt` and `MyOtherFile.txt`:

```
"exclude" : [
    "MyDir",
    "MyOtherDir",
    "MyFile.txt",
    "MyOtherFile.txt"
]
```

System

The optional `system` attribute defines application project settings for the system when creating a new project. The `dmclkid` attribute defines the data motion clock ID. If the `system` attribute is not specified, the data motion clock uses the default clock ID.

The following example will result in SDx setting the data motion clock ID to 2 instead of the default clock ID when creating the new project:

```
"system" : {
    "dmclkid" : “2”
}
```

cmd_args

The optional `cmd_args` attribute defines a custom command line when launching the SDSoC application. There are two special variables that can be used in the `cmd_args`:

- `PROJECT` is replaced with the path to the project directory.

- `BUILD` is replaced with the path to the build directory.

These variables can be used to specify project-rooted data files, or build-output files.

Revision

The optional `revision` attribute provides the revision of the sample application, and with some limited revision history. The attribute includes several optional sub-tags:

- `date`: A user-meaningful string representing a date for the sample application.
- `version`: A string representing the version of the application as a period-delimited string of numbers, typically `major.minor[.update]`. For example: 1.0, or 2.5.1. If a sample application lists multiple versions, the highest number is considered the most recent version.
- `description`: Astring providing a brief description of the revision.

The following provides a brief revision history of the sample application:

```
"revision" : [
    {
        "date" : "2017",
        "version" : "1.2",
        "description" : "Updated the example for the 17.1 release"
    },
    {
        "date" : "2018",
        "version" : "1.0",
        "description" : "original release"
    }
]
```

Contributors

The `contributors` attribute is a list of contributors to the example, including the following sub-tags:

- `group`: A string specifying the name of the sample application developer.
- `url`: Specifies a URL for the contributor.

The following is an example:

```
"contributors" : [
    {
        "group" : "Xilinx, Inc.",
        "url" : "http://www.xilinx.com/"
    }
]
```

Platform Checklist

The overview of the platform creation process in this appendix touches on hardware and software platform requirements and components, platform validation, sample application support, directory structures and the platform metadata files that enable SDSoC™ to use your custom platform.

The SDSoC platform creation process requires familiarity with the Vivado® Design Suite and its use in creating Zynq®-7000 SoC or Zynq® UltraScale+™ MPSoC designs; familiarity with SDSoC from the perspective of a user of platforms; and familiarity with Xilinx® software development tools such as the Xilinx Software Development Kit (SDK), and embedded software environments (Linux or bare-metal).

If you are new to the SDSoC platform creation process, read the introductions in the chapters of this guide while lightly reading through the material for key concepts, and examine one or more of the examples discussed in Appendix E: SDSoC Platform Examples.

If you have previously created SDSoC platforms, you should still read though the chapters in this guide and the migration information in Appendix C: Migrating SDSoC Platforms to a New Release.

The checklist below summarizes tasks involved in SDSoC platform creation.

1. Using the Vivado Design Suite, create a Zynq-7000 or Zynq UltraScale+ MPSoC based design.
   - Refer to the Chapter 3: Creating the Platform Hardware Component for requirements and guidelines to follow when creating the Vivado hardware project. Test the hardware design using the Vivado Design Suite tools.
2. For supported target operating systems, provide software components for boot and user applications.
   - SDSoC creates an SD card image for booting the OS, if applicable, using boot files included in the platform.
     - A first stage boot loader (FSBL) is required, as well as a Boot Image File (BIF) that describes how to create a BOOT.BIN file for booting.
     - For Linux boot, provide U-boot and a Linux image (device tree, kernel image, and root file system as discrete files or FIT (Flattened Image Tree) boot image `.ub` file).
     - For bare-metal applications, create a linker script.

    - Zynq UltraScale+ MPSoC platforms also require ARM trusted firmware (ATF) and power management unit firmware (PMUFW).
    - Optionally create a `README` file and other files which need to reside on the SD card image.
  - If the platform provides libraries to link with the user's application, headers and libraries can be included as part of the platform for convenience.
  - See Chapter 5: Sample Applications for more information.
3. Optionally create one or more sample applications.
   - In the platform folder, you can create a `samples` folder with a single level of subfolders, with each subfolder containing the source code for an application. The samples folder also contains a `description.json` file used by the SDx™ IDE New Project wizard when creating an application.
   - See Chapter 5: Sample Applications.
4. Use the SDx IDE to create a Platform project to package the hardware and software components into an SDSoC platform.
   - As described in Creating an SDSoC Platform Project, the project combines the hardware platform DSA with the embedded processor information, operating system, and compiler settings to define the platform for use with SDSoC compilers.
5. Validate your platform supports the SDSoC environment.
   - The Chapter 2: Creating SDSoC Platforms chapter describes platform `Conformance` tests for data movers used by the SDSoC system compiler. Each test should build cleanly by running `make` from the command line in a shell available by launching an SDx Terminal or by running a settings64 script (`.bat`, `.sh` or `.csh`) found in the SDx installation. The tests should also run on your platform board.
6. Validate project creation with your platform in the SDx IDE.
   - Start the SDx IDE and create an SDSoC application project or system project using the New Project wizard. After specifying a project name, you are presented with a list of platforms. Click on your custom platform to select it. If your platform includes a samples folder, you can select one of your applications, or the empty application to which you can add source files. After your project is created, build it and run or debug the ELF file.

Appendix B

SDx IDE Glossary

SDx™ IDE

The SDx IDE descriptions in this appendix refer to platform projects (as opposed to application projects) and consists of multiple views that can be configured into different perspectives, or view configurations. The default perspective has the following features:

- **Explorer** view: On the left side of the display, this view lets you navigate through the project hierarchy, source files, and resources.
- **Editor** view: In the center of the display, the Editor displays the project and lets you modify features of the project and edit code, scripts, and configuration files. Files can be opened by double-clicking on them in the Explorer view.
- **Console** view: At the bottom, this view displays the output for the different processes and utilities that make up the SDx tool.

The views are configurable from the **Window→Show View** command. Views can also be shown or hidden, and arranged as needed, and saved into perspectives that can be loaded into the tool. For more information on working with the SDx IDE, refer to the *SDSoC Environment User Guide* (UG1027).

In the Editor view, the elements of the processor are contained in three levels of the platform: Platform, System Configurations, and Processor Domains. The bottom of the Editor view also displays a small workflow that defines the process for creating a platform. The following sections describe how you can define the elements of your platform, using the established workflow.

Platform

In the previous example, the platform project has been populated by the information you provided when you created the project:

- **Name**: Displays the name.
- **DSA**: Indicates the DSA file associated with the platform.

IMPORTANT! *The name and DSA file are specified when the platform is created and cannot be changed.*

- **Description**: This field is initially copied from the platform name. However, you can click the **Edit** command (✎) to modify the description to provide more details of the platform.

- **Samples**: This optional field specifies the path to a folder containing sample applications for use with the platform. See Chapter 5: Sample Applications for more information.

TIP: *The Samples folder can be specified through the **Browse** command, or you can add samples in the **resources** folder of the platform project and use the Search command ( ) to load it. The **resources** folder, as shown in the Explorer view is local to the project in the workspace, and can simplify the process of sharing the platform project with others.*

Defining the System Configuration

The System Configuration defines the software environment that is booted and runs on the hardware platform. It will specify Operating Systems and the run-time settings for the processors in the hardware platform, and will also have software-configurable hardware parameters.

With the platform project opened in the SDx IDE, you can add System Configurations to the platform by clicking the **Define System Configuration** command in the workflow at the bottom of the Editor. This will open the New System Configuration dialog box as shown below.

TIP: *You can also use the **Add** command ( ) in the Editor and select **System Configuration**.*

The fields of the System Configuration dialog box include:

- **Name**: Specifies an identifier for the System Configuration. The name should be alphanumeric, between 3 and 40 characters long, and include no special characters except underscore, '_', and dash, '-'. Because the System Configuration name is an identifier it cannot be modified after it is created.

TIP: *When using the Makefile flow, or command-line, you can specify this identifier using -sds-sys-config<name> option of `sds++` to specify the software platform used, which includes the target operating system and other settings.*

- **Display Name**: The name that will be displayed by the SDx IDE and in reports. This name can include spaces and special characters, and can be modified.
- **Description**: A brief description for the configuration.

IMPORTANT! *It is assumed that any files referenced in the BIF file will be in the directory specified by the Boot Directory field.*

Click **OK** to close the New System Configuration dialog box and add the configuration to the platform. You will see the configuration listed in the Editor view.

If you select the System Configuration in the tree view of the platform, on the left side of the Editor, you will see the information for the System Configuration as you have defined it. In addition, the **Readme** field is displayed to let you define a `readme` file associated with the configuration. The `readme` file should be available along with the SD Card. This file informs users of the platform how to boot an application for this configuration, and how the board should be physically set, for example Jumper Settings. It is a plain text file that contains instructions to the user.

Defining the Processor Domain

The Processor Domain will define the OS operating on one or more processors on the device, and the run-time. The domain defines different settings for the OS, whether Linux, FreeRTOS, or Standalone.

As shown in the figure below, SDSoC™ allows for applications targeting the standalone or FreeRTOS operating system to be built to target a specific **Processor** core (Cortex™-A9, Cortex-A53, or Cortex-R5), and for Linux applications to be built to run on all cores that are visible to the OS. When defining a domain for the Linux operating system, you do not need to target a specific core, but instead can assume that the Linux scheduler will schedule across processors as appropriate, and isn't restricted to any particular processor core.

You can add domains to a System Configuration by clicking the **Add Processor Group/Domain** command in the workflow at the bottom of the Editor. This opens the New Domain dialog box as shown below.

TIP: *You can also click the **Add** button (➕) in the Editor and select **Domain**.*

The fields of the System Configuration dialog box include:

- **Name**: Specifies the name of the domain.

TIP: *The name should be alphanumeric, between 3 and 40 characters long, and include no special characters except underscore, '_', and dash, '-'.*

- **Display Name**: The name that will be displayed by the SDx IDE and in reports. This name can include spaces and special characters.
- **OS**: Specify the OS you are configuring as either Linux, FreeRTOS, or Standalone from the drop-down menu. For each OS there are various files that must be included in order to compile, link, or generate the boot files for a given application. The fields of the New Domain dialog box vary depending on the OS selected.
- **Processor**: Defines the available types of processors for the platform. The types of processors available is determined by the IP in the hardware platform as defined by the DSA. Any processor IP present in the hardware design will show up in this list.

- **Supported Runtime**: Specify the kernel runtime for the platform from the drop-down menu. This determines the compiler used to process the kernel logic at runtime. **C/C++** is supported for all OSes.
- **Prebuilt Linux Image**: In case of Linux OS, you need to build the Linux image for the Hardware design, using PetaLinux for example, and provide the image directory. More details about how to build the correct image, refer to Chapter 4: Creating the Platform Software Component.
- **Linker Script**: This is required for FreeRTOS and Standalone Only. This is the linker script which will be used while linking the baremetal or FreeRTOS ELF in the SDx application build. This file allows the programmer to control how the sections are merged in the ELF, and at what locations they are placed in memory. It also allows the user to specify how much of DDR memory is allocated for the stack and heap.

TIP: *When you add the linker script to your SDx platform, you must increase the stack and heap sizes because the SDK default values are too small for a typical SDx application.*

- **Description**: A brief description for the domain.

Click **OK** to close the New Domain dialog box and add the Processor Domain to the platform. You will see the domain listed in the Editor view. After the domain has been defined, the SDx IDE shows additional details for the domain that can be edited when it is selected in the platform project, as shown below.

The new fields of the domain include the following:

- **Repositories**: Available only for Standalone or FreeRTOS. Used to keep the embedded software drivers and libraries that are required to create the BSP for the user hardware design. This is a directory and it will be copied into the `<platform>/sw/<sysconfig>/<domain>/bspRepo` location. And an entry will be added in the `spfm` file with the tag `sdx:bspRepo`.
- **Prebuilt data**: This specifies a directory containing a prebuilt bistream and pre-generated software artifacts, meant to be representative of a software-only project with no functions accelerated in hardware. See Prebuilt Hardware for more information on generating prebuilt data for custom platforms.

In addition, when the domain is selected in the Editor, you will see sub-headings of the domain as follows:

- **Prebuilt Image**: For Linux domains only, this indicates the **Prebuilt Linux Image** content that was specified when the domain was created.
- **Application Settings**: For Standalone or FreeRTOS domains, this indicates the **Linker Script** that was specified when the domain was created.

- **Board Support Package**: For Standalone or FreeRTOS domains, this specifies a Microprocessor Software Specification (MSS) file that defines the board support package (BSP) libraries and drivers for the base platform without any accelerators, and is the baseline for creating an MSS for the final design that includes the platform, data motion network and hardware accelerator functions. With the MSS, you will be able to define the libraries and the library options. This option will be added to the `spfm` file with the tag `sdx:bspConfig`. The MSS file provided will be copied in the `<platform>/sw/<sysconfig>/<domain>` location.

IMPORTANT! *You should provide an MSS file only if you do NOT want to use default settings for the BSP. If you specify a MSS file, you must also specify* ***Include Paths*** *pointing to a directory containing include files generated when using the MSS file.*

- **Libraries**: Displays two fields:
  - **Libraries**: Lets the platform user select the individual libraries to make available for an application.
  - **Include Paths**: Lets the platform user point to directories containing include files to make them available for an application.

TIP: *Libraries will be copied into `<platform>/sw/<sysconfig>/<domain>/lib`, and* ***Include Paths*** *will be copied into `<platform>/sw/<sysconfig>/<domain>/inc`.*

Generate Platform and Add to Repository

After configuring the settings for your SDSoC platform project, you can click the **Generate Platform** command in the workflow at the bottom of the Editor. This will start the process of copying and creating files, and generating metadata for your platform. The new platform files will be written into the workspace for the platform project. You can regenerate the platform files as needed.

IMPORTANT! *If you make any changes to any of the fields in the platform project, System Configuration, or domains, you must regenerate the platform to update the platform in the repository. You do not need to re-export the platform to the repository, but you will need to regenerate the output.*

When the SDx platform is generated, it is written to the platform project folder in the `workspace/<project_name>/export` folder.

Finally, with the platform generated, click the **Add to Custom Repositories** command in the workflow at the bottom of the Editor. This adds the exported platform to the SDx custom repository for use in SDx Application and System projects. The custom repository is a list of directories, where the SDx tool scans for additional platforms. When you click **Add to Custom Repositories**, the platform output directory (`./export`) is added to the repository.

To share this platform with other users you can copy the exported platform folder to a common location, and ask other users to add that location to their custom repository using the **Xilinx→ Add Custom Repository** menu command. The platform will be added to the list of available platforms as shown below.

TIP: *The SDx GUI journals the TCL commands that underlie the actions identified in this section so that you can script platform generation. For an example of using the command-line flow to build an SDSoC platform, refer to* Making the SDSoC Platform from the Command Line.

Migrating SDSoC Platforms to a New Release

Introduction

To support a newer release of the SDx™ development environment you must upgrade the Vivado® Design Suite project included in the hardware platform to the latest release, updating the IP used in the design, and regenerating output products for the project. As part of building an application project, SDx launches Vivado and attempts to auto-upgrade everything in the design. Updating the IP integrator design may be a simple matter of plugging in the latest IP revision for the current release. However, it can also be complicated in the case of major version changes from one release to another by the addition or removal of interface signals on the IP, or updated parameters. In this case, upgrading the Vivado Design Suite project can require more effort, and auto-upgrade will not be successful.

You may also need to update the software platform to be compatible with or take advantage of any new features of the hardware platform. Finally, you must regenerate your custom platform using the SDx platform project. This may simply be a matter of upgrading the Vivado® IP integrator block design to the latest release and rebuilding the software components with the latest SDSoC™ tools.

IMPORTANT!

*The Vivado tool requires you to **Upgrade IP** for every new version of the Vivado Design Suite. If you encounter IP Locked errors when the SDSoC platform project tries to generate the platform, or when the SDx IDE invokes the Vivado tools, it can be the result of failing to properly copy the Vivado project as described in Begin with a Vivado Project, or failing to upgrade the IP used in the Vivado project for a new release.*

To migrate an SDSoC hardware platform from a prior release, open the Vivado project in the new version of the Vivado tools, and upgrade the IP integrator block design and all IP, and regenerate the output products. Refer to this link in the Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator (UG994) for more information on updating block design projects.

You can build a new platform by modifying an existing platform, but be aware that the block diagram with in the DSA contains properties must be unset or they will interfere with your new platform. In particular, you must unset any property that you do not wish to be set. As an example, if you wish to use a different set of clock sources in your derived platform, make sure you unset any clock in the original platform or there will be clashes in your derived platform (recall: every platform clock ID must be unique).

Migrating Platforms to 2019.1 and Later

The platform structure and process for creating platforms has changed starting 2017.4, notably in the three following areas:

- SDSoC Tcl commands have been replaced with Vivado properties requiring you to update the project and rerun the `write_dsa` command, as described in Mapping SDSoC Tcl Commands to Vivado Properties.
- IP Caching is enabled for hardware platforms, which reduces the time to compile the hardware accelerated functions. You must set the `dsa.ip_cache_dir` property, re-synthesize the Vivado project to populate the cache, and regenerate the DSA file.
- The SDx IDE now supports directly creating an SDSoC platform project. Regenerating an SDSoC platform will now be done as described in Chapter 2: Creating SDSoC Platforms.

Mapping SDSoC Tcl Commands to Vivado Properties

The Tcl commands used in prior releases to set the platform and interface properties in the SDSoC platform have been replaced with a set of properties that can be defined directly in the Vivado Design Suite project. This requires reconfiguring the platform and interface properties for each platform to migrate it to the 2017.4 release before you can move to a more current version.

The following table shows the mapping of sdsoc:: Tcl commands to the PFM properties required in 2017.4 and beyond.

Table 2: **Mandatory Properties: Pre-2017.4 and Post 2017.4**

Purpose	Pre 2017.4	Post 2017.4
Declare the hardware platform	sdsoc::create_pfm	PFM_NAME
Define the DSA Vendor	Not required	PFM_NAME
Define the hardware platform name	sdsoc::pfm_name	PFM_NAME
Define a brief description of the platform	sdsoc::pfm_description	Defined in the SDSoC platform project.
Declare the platform clock ports	sdsoc::pfm_clock	PFM.CLOCK
Declare the platform AXI bus interfaces	sdsoc::pfm_axi_port	PFM.AXI_PORT
Declare the platform AXI4-Stream bus interfaces	sdsoc::pfm_axis_port	PFM.AXIS_PORT
Declare the available platform interrupts	sdsoc::pfm_irq	PFM.IRQ
Generate the hardware platform DSA.	sdsoc::generate_hw_pfm	Replaced by the `write_dsa` Tcl command.

Appendix D

Changing the SDSoC Platform Device

Introduction

This document describes how to migrate a base SDSoC™ platform, provided in the software release, to a part-based platform using one of the different devices available from Xilinx®.

When building a new custom platform for the SDSoC development environment, you will usually start with an existing base platform, such as the ZCU102 platform. Then, after you have performed some initial design work, you will want to migrate your design to a custom platform that more closely meets your needs. Your custom platform might require a different device than is used by the base platform, and use custom IP.

The methodology starts from a base platform, has you modify the platform device support archive (DSA) in the Vivado Design Suite, and write the modified DSA file, update the software components for the platform, and create a new SDSoC platform in the SDx™ IDE. The various steps are detailed in the following sections.

Edit the Platform DSA

Copy the Base Platform

The first step is to locate the platform specific files for one of the SDSoC base platforms. The files are provided as a part of the SDSoC software installation, and can generally be found at the following location: `<SDx_Install_Dir>/platforms`. For instance, the files for the ZCU102 platform can be found at `<SDx_Install_Dir>/platforms/zcu102`.

TIP: *It is a good idea to copy the platform folder to make sure you are editing the copy, and not the original platform from your software installation. Though perhaps your file permissions would prevent you from modifying the installation files directly.*

Open the Platform DSA

The DSA file for the platform can be can be found in the `./hw` folder of the platform. For example, the DSA file for the ZCU102 platform is found at `zcu102/hw/zcu102.dsa`.

In your copy of the platform, open the DSA in the Vivado Design Suite using the following process.

1. Launch the Vivado Design Suite:

   ```
   source <SDx_Install_Dir>/settings64.csh
   vivado &
   ```

2. From within the Vivado IDE, open the DSA by using the following command from the Tcl Console:

   ```
   open_dsa <path_to_dsa>/zcu102.dsa
   ```

A new project is created in your current working directory, and the block design is opened in the Vivado IP Integrator. The block design inside the DSA file has the name from the original platform you are editing. At this point, you should rename the block design to match the name of the new platform you are creating.

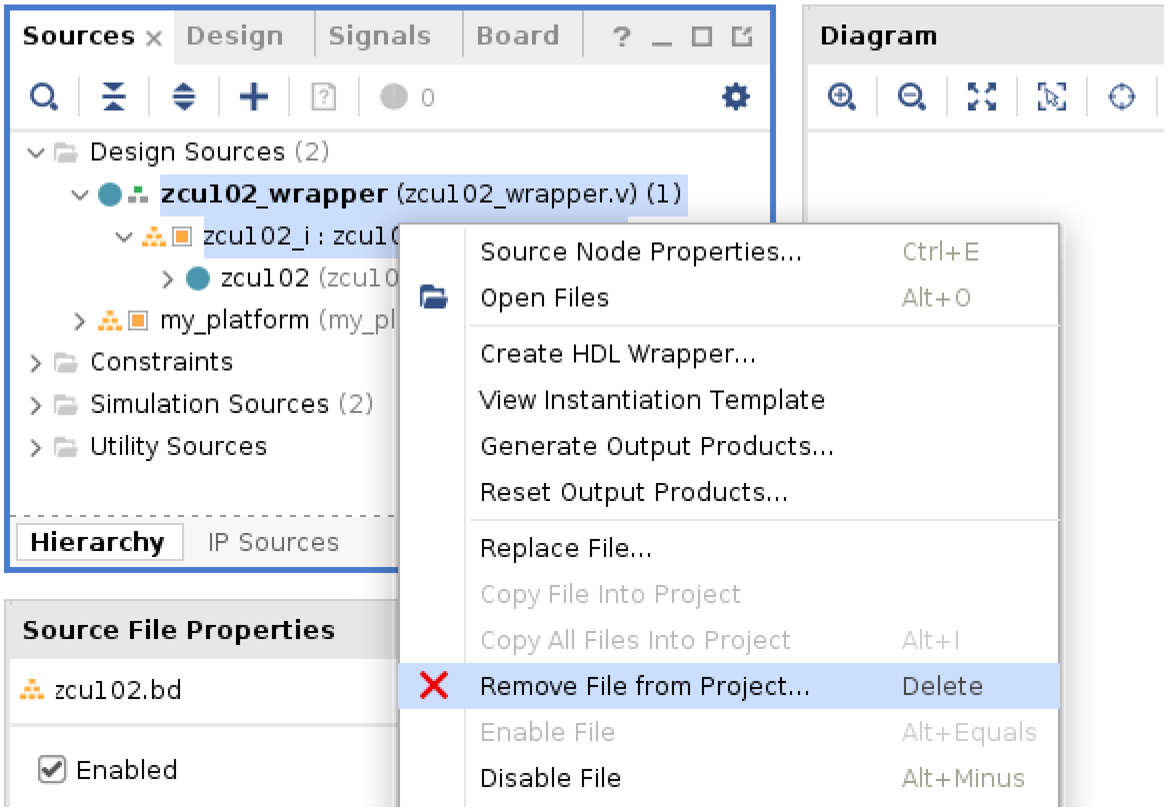
Use the **File→Save Block Design As** command from the main menu to rename the block design.

TIP: *When you use the* ***Save Block Design As*** *command, the new block design is added to the current project.*

Remove Original Block Design

To remove the original block design, and its wrapper from the project, open the Sources window, and expand the wrapper to show original block design inside. Select both the wrapper and the block design. Right-click and select the **Remove File from Project** command.

Figure 67: **Remove File from Project**



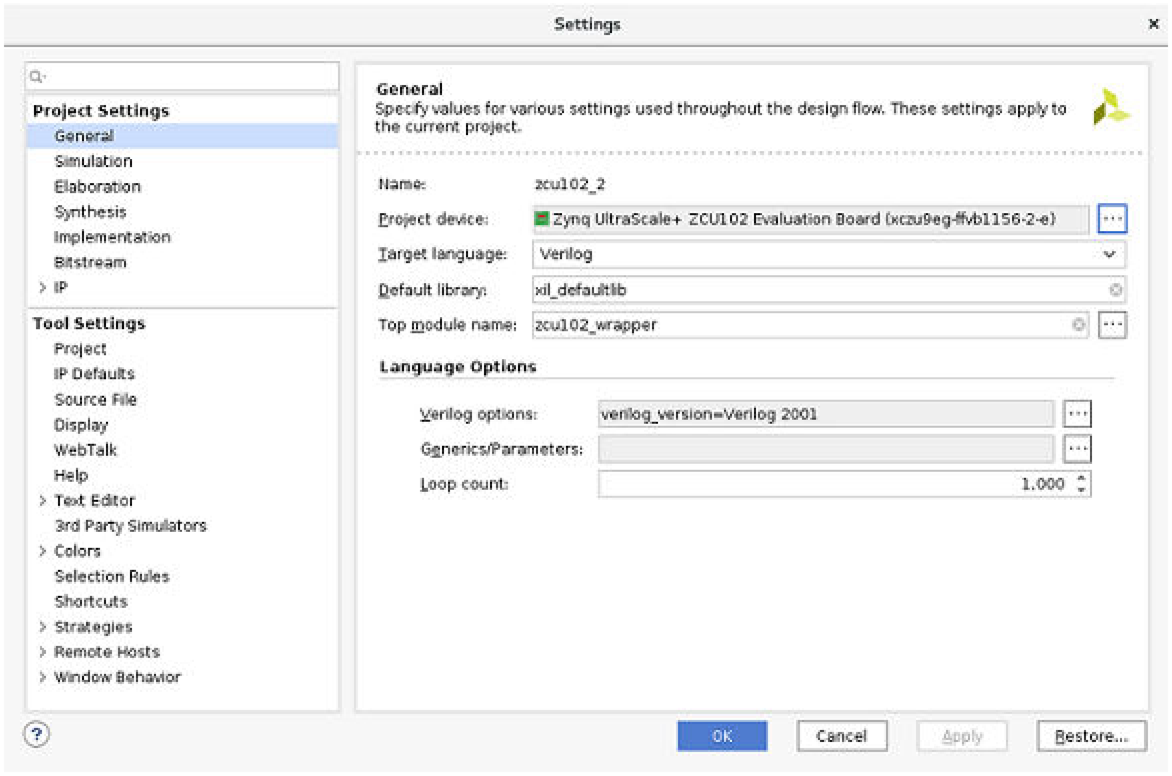
This removes the HDL wrapper and the original block design from the project. In the Remove Sources dialog box that opens, you can select the **Also delete the project local file/directory from disk** checkbox to completely delete the files. This should leave the new block design in the project, and you are ready to move to the next step of changing the target part for the project.

TIP: *Removing the HDL wrapper opens the Invalid Top Module dialog box as there is no top-level design module for the project. Select the* ***Ignore and continue with invalid top module*** *option, and click* ***OK****.*

Change the Project Part

At this point you can edit the project settings to change the target part for the project. Select the **Settings** command under the Project Manager menu in the Flow Navigator window. The Settings dialog box will open displaying the current Project device as shown below.

Figure 68: **General Page of the Settings Dialog Box**



General
Specify values for various settings used throughout the design flow. These settings apply to the current project.

Name: zcu102_2
Project device: Zynq UltraScale+ ZCU102 Evaluation Board (xczu9eg-ffvb1156-2-e)
Target language: Verilog
Default library: xil_defaultlib
Top module name: zcu102_wrapper

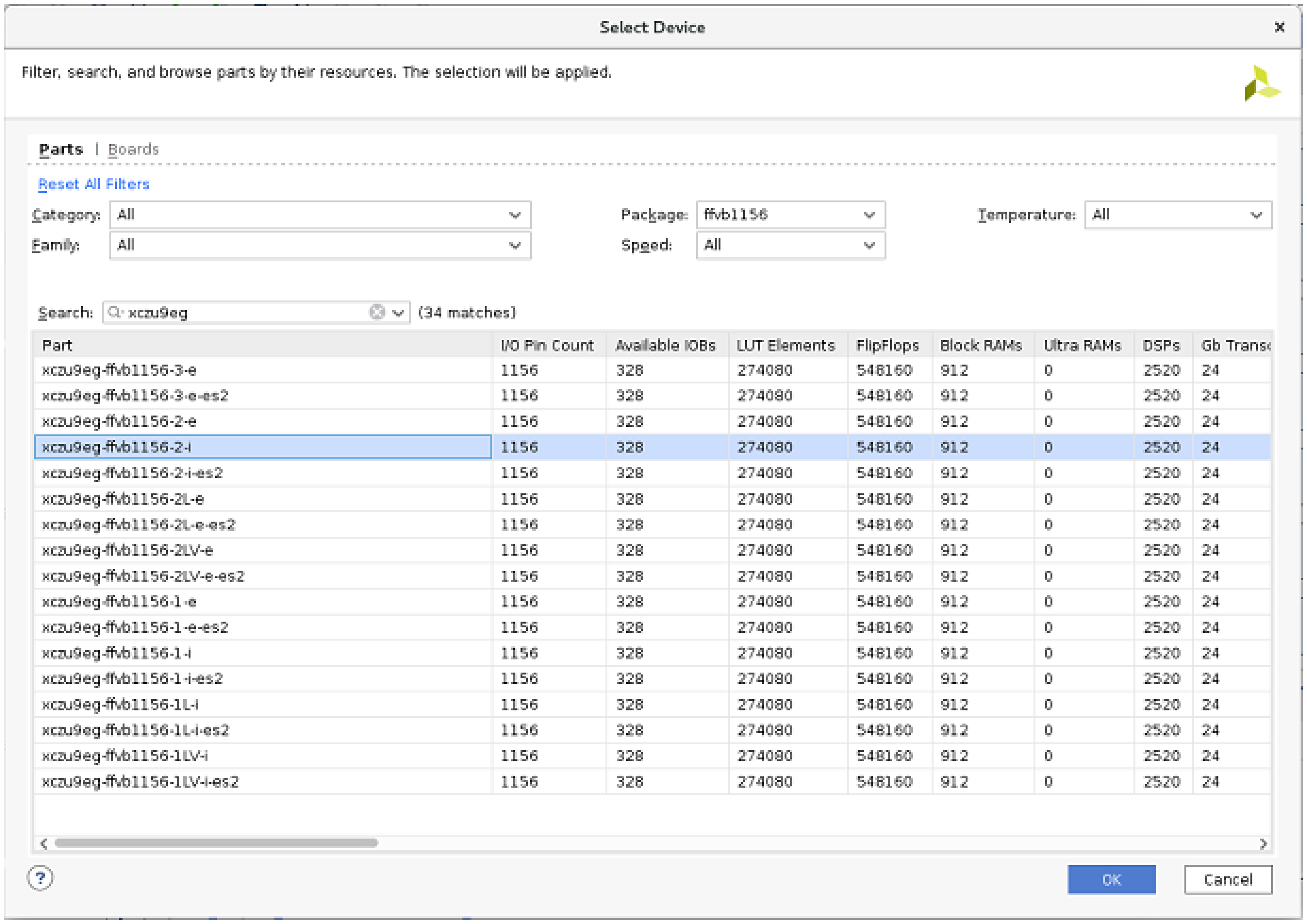
Language Options

Verilog options: verilog_version=Verilog 2001
Generics/Parameters:
Loop count: 1,000

Select the **Browse** button for the Project device field to open the Select Device dialog box. From this dialog box you can select a board or part for the project. Select the **Parts** tab at the top of the Select Device dialog box to list the available parts as shown below. The displayed parts can be filtered by Category, Family, Package, etc. You can also use the Search field to filter the displayed parts by a specific search string.

Select a new Project part for your design, for example the `xczu9eg-ffvb1156-2-i` device.

Figure 69: **Select Device**



Select Device

Filter, search, and browse parts by their resources. The selection will be applied.

Parts | Boards

Reset All Filters

Category: All
Family: All
Package: ffvb1156
Speed: All
Temperature: All

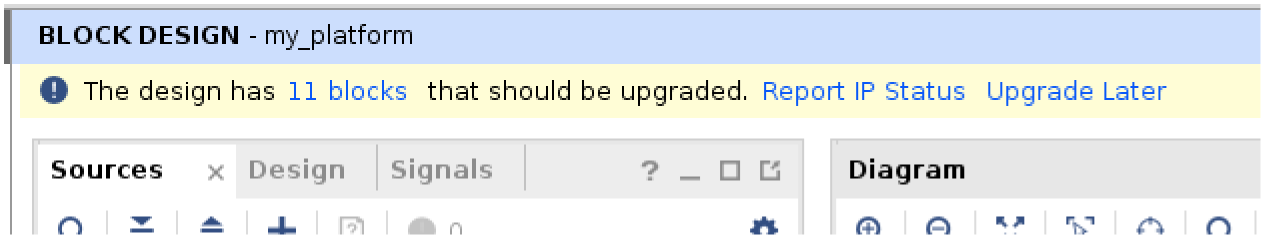
Search: xczu9eg (34 matches)

Part	I/O Pin Count	Available IOBs	LUT Elements	FlipFlops	Block RAMs	Ultra RAMs	DSPs	Gb Trans
xczu9eg-ffvb1156-3-e	1156	328	274080	548160	912	0	2520	24
xczu9eg-ffvb1156-3-e-es2	1156	328	274080	548160	912	0	2520	24
xczu9eg-ffvb1156-2-e	1156	328	274080	548160	912	0	2520	24
xczu9eg-ffvb1156-2-i	1156	328	274080	548160	912	0	2520	24
xczu9eg-ffvb1156-2-i-es2	1156	328	274080	548160	912	0	2520	24
xczu9eg-ffvb1156-2L-e	1156	328	274080	548160	912	0	2520	24
xczu9eg-ffvb1156-2L-e-es2	1156	328	274080	548160	912	0	2520	24
xczu9eg-ffvb1156-2LV-e	1156	328	274080	548160	912	0	2520	24
xczu9eg-ffvb1156-2LV-e-es2	1156	328	274080	548160	912	0	2520	24
xczu9eg-ffvb1156-1-e	1156	328	274080	548160	912	0	2520	24
xczu9eg-ffvb1156-1-e-es2	1156	328	274080	548160	912	0	2520	24
xczu9eg-ffvb1156-1-i	1156	328	274080	548160	912	0	2520	24
xczu9eg-ffvb1156-1-i-es2	1156	328	274080	548160	912	0	2520	24
xczu9eg-ffvb1156-1L-i	1156	328	274080	548160	912	0	2520	24
xczu9eg-ffvb1156-1L-i-es2	1156	328	274080	548160	912	0	2520	24
xczu9eg-ffvb1156-1LV-i	1156	328	274080	548160	912	0	2520	24
xczu9eg-ffvb1156-1LV-i-es2	1156	328	274080	548160	912	0	2520	24

As soon as you change the project settings from a board flow to a part-based flow, all of the IP in the design will go stale, be locked, and require an upgrade. This is indicated by the yellow banner at the top of the block design. At the very least, the change of the target part in the project has made the current customization of the IP in the design out-of-date. For this reason the IP is locked, and must be updated to re-target the new part in the design.

TIP: *There may be other reasons, such as major or minor revisions, that the IP may be locked and need to be updated.*

Figure 70: **Locked IP Banner**


BLOCK DESIGN - my_platform

The design has 11 blocks that should be upgraded. Report IP Status Upgrade Later

Click the **Report IP Status** link in the banner. This opens the **IP Status** report. Select all of the IP in the report and click the **Upgrade Selected** command.

After the IP in the design are upgraded, the Generate Output Products dialog box is displayed. Select **Generate**. This starts the generation of the necessary HDL files for the IP contained within the block design. Wait for the generation of the output products to finish. The **Design Runs** window displays the status of the Out-of-Context module runs.

Edit Platform Properties

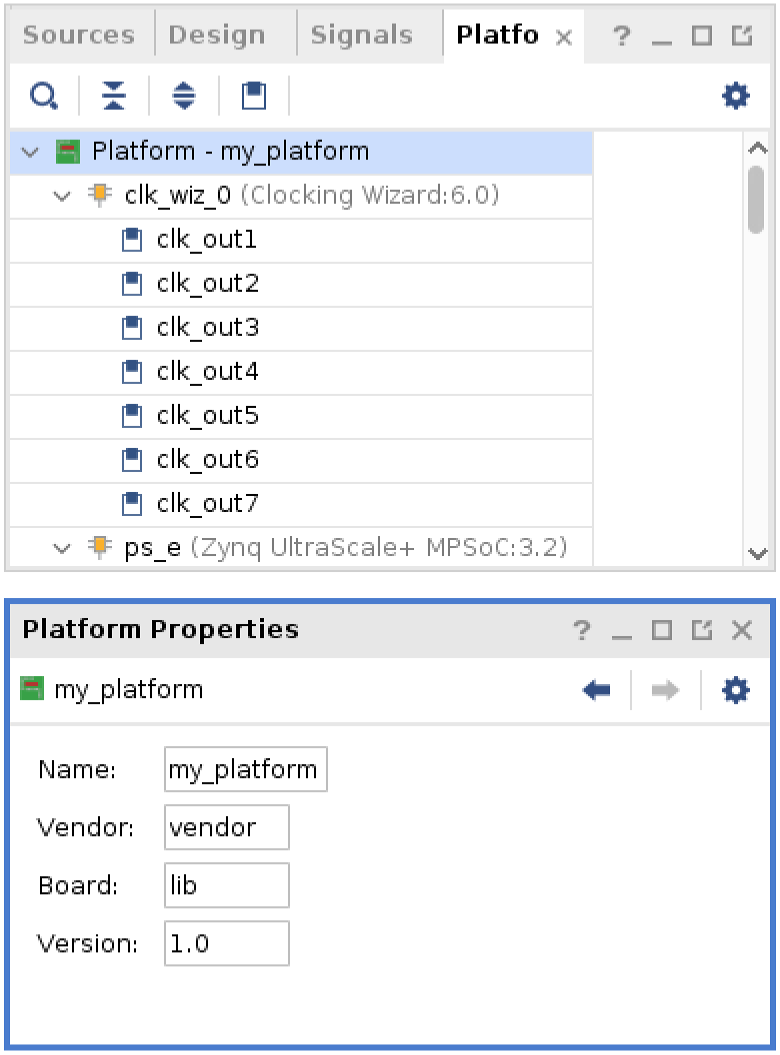
The block design needs platform properties to be defined, as explained in Declaring Platform (PFM) Interfaces and Properties. Review that topic for a complete discussion of what needs to be done.

Begin editing the platform properties by selecting **Window → Platform Interfaces** from the Vivado menu bar and clicking on the **Enable platform interfaces** link. This displays any existing platform properties that you may need to update.

TIP: *The platform properties defined in the original block design may be inherited in the new block design you created using the **Save Block Design As** command. You will need to set the platform properties again to insure they properly define your platform.*

In the Platform Interface window, select the top-level Platform, as shown in the following figure. Selecting the top-level Platform in the Platform Interfaces window, opens the Platform Properties window, also shown in the following figure.

Figure 71: **Select the Top-Level Platform**



Name: my_platform
Vendor: vendor
Board: lib
Version: 1.0

Notice the Platform Properties window displays the Name, Vendor, Board, and Version properties, which you can edit as needed to define your custom platform. Edit the **Vendor** field to specify your company name, or appropriate name as the Vendor of this platform.

IMPORTANT! *You must edit at least one field in the Platform Properties window to have the properties written to the current project, which is required for the DSA.*

After editing the Platform Properties, you should see commands similar to the following commands written to the Tcl Console window:

```
set_property pfm.name my_platform [get_files ...]
set_property dsa.name "my_platform" [current_project]
set_property dsa.vendor "xilinx.com" [current_project]
set_property dsa.board_id "lib" [current_project]
```

In the Platform Interfaces window, check the Clocking properties, the Interface properties, and the Interrupt properties as defined in Configuring Platform Interface Properties.

With the platform properties defined, you can validate the block design by running validation. This can be done by typing the following command in the Tcl console:

```
validate_bd_design -include_pfm -force
```

Write the Platform DSA

In this step, you will create an HDL wrapper for the block design, generate a bitstream for the design (after synthesis and implementation), export the hardware description file, and write the DSA file for use by SDSoC.

In the Sources window, create a new HDL wrapper by right-clicking on the block design and selecting **Create HDL Wrapper**. In the Create HDL Wrapper dialog box, select the **Let Vivado manage wrapper and auto-update** option, and click **OK**.

After the wrapper is created, right-click the wrapper in the Sources window, and select the **Set as Top** command to specify this as the top-level of your platform design.

In the Flow Navigator, select **Generate Bitstream**. The Vivado tool notifies you that there are no implementation results, and asks if you want to run synthesis and implementation. Click **Yes** to proceed. Click **OK** to launch the runs.

With the bitstream generated for the platform, use the **File → Export → Export Hardware** command to write the hardware description file for the project. Select **Include bitstream** when prompted in the Export Hardware dialog box.

You are now ready to generate the new platform DSA, to replace the original DSA file you opened at the start of this process. In the Tcl Console, use the following commands to write and validate the DSA:

```
write_dsa -force -include_bit <path_to_platform>/my_platform.dsa
validate_dsa my_platform.dsa
```

With the platform DSA file exported, you are now ready to import the DSA into a new platform project, and define the software platform elements as described in the next section.

Create the New Platform

TIP: *The process for creating a new platform project, and defining the software platform elements from the DSA file, is explained in detail in* Chapter 4: Creating the Platform Software Component*, and demonstrated in the* *SDSoC Platform Development Tutorial* *found on GitHub.*

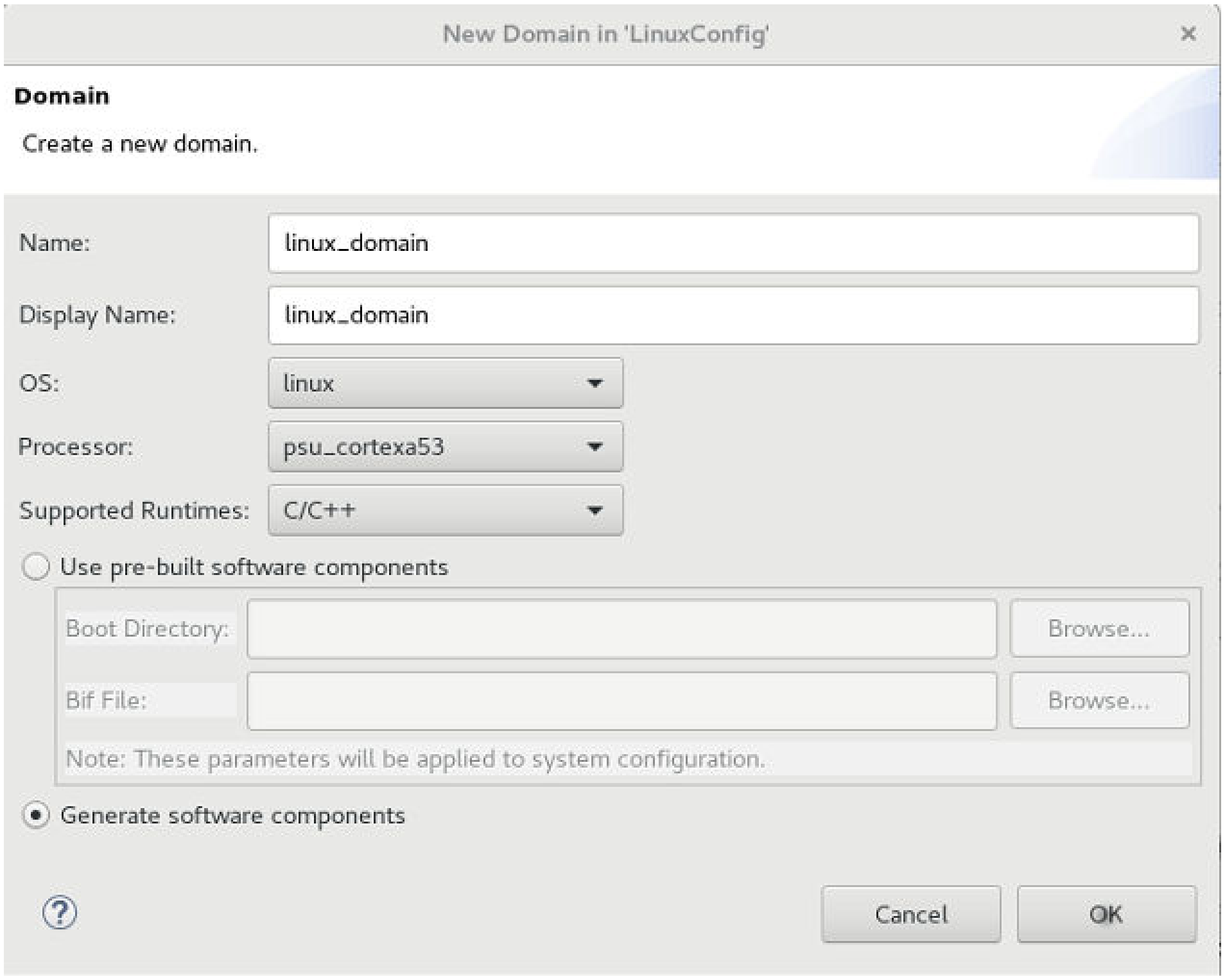
With the platform DSA file created, as described in the last section, you are now ready to define an SDSoC platform project, and the software platform elements needed to complete the updated platform. Start by opening the SDx IDE and creating a new platform project. Specify the platform name, taking care to use the same name you specified in the platform DSA file, the block design, and the PFM_NAME property.

Specify the **Create from hardware specification (DSA)** option in the New Platform Project dialog box, and click **Next**.

On the Platform Project Specification page, select the **Browse** button for the DSA file field, and navigate to select the DSA file for the new platform. The SDx IDE will populate the **Operating system** and **Processor** fields automatically, though you are free to change them as needed.

The platform project is created, and the Platform Configuration Settings opens in the Editor area of the SDx IDE. The SDx tool automatically creates a system configuration, called `sysconfig1`, and processor domain with a name based on the operating system and processor you selected. You can add additional System Configurations and Domains to add operating systems for specific processor cores.

Figure 72: **Add Domain - Linux**


New Domain in 'LinuxConfig'

Domain

Create a new domain.

Name: linux_domain

Display Name: linux_domain

OS: linux

Processor: psu_cortexa53

Supported Runtimes: C/C++

Use pre-built software components

Boot Directory:

Bif File:

Note: These parameters will be applied to system configuration.

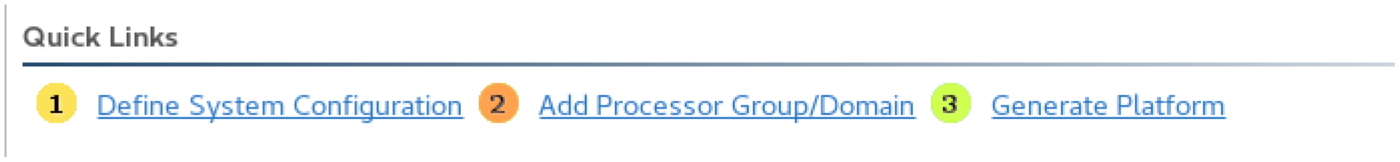
Generate software components [✓]

You can populate the software files needed for the platform, either by selecting the **Generate software components** option, or locating existing files by selecting the **Use pre-built software component** option.

IMPORTANT! *Generally, you should* ***Generate software components*** *to avoid difficulty trying to reuse existing software components, as existing software boot files, and software drivers may no longer work for the modified platform. As an example, a new driver for the custom IP in the block design may be needed when building the software files. In this case, you can let the SDx IDE generate the software components directly. Refer to the instructions in* Linux Boot Files *for information on setting up your PetaLinux installation to let the SDx IDE generate your Linux software components. Again, refer to the* *SDSoC Platform Development Tutorial* *for an example of this process.*

If there are no major hardware changes, you can use pre-built software components. You can locate these in an existing original platform, for example. For a standalone configuration, you will need to specify the **Boot Directory**, and the **Bif File**. For the Linux configuration, you will need to specify the **Boot Directory**, and the **Bif File**, as well as the **Image** folder in the domain, that contains the `image.ub` file.

Figure 73: **Generate Platform**



With the system configurations, and processor domains defined, you can click **Generate Platform** to create the SDSoC platform. Once the platform is created, you can add it to your board repository, and use it in SDSoC application projects.

Appendix E

SDSoC Platform Examples

Introduction

This appendix provides simple examples of SDSoC™ platforms created from a working hardware system built using the Vivado® Design Suite, with a software run-time environment, including operating system kernel, boot loaders, file system, and libraries that run on top of the hardware system. Each example demonstrates a commonly used platform feature, and is built upon the ZC702 board available from Xilinx.

- `zc702_axis_io` - Accessing a data stream that could represent direct I/O from FPGA pins in an SDSoC platform
- `zc702_acp` - Sharing a processing system AXI bus interface between the platform and the `sdscc` system compiler

Each example is structured with the following information:

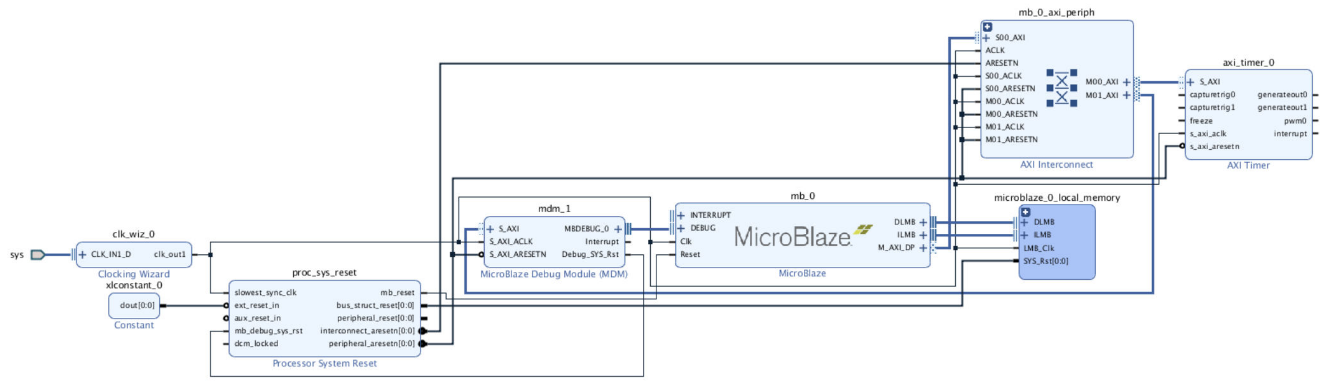
- Description of the platform and what it demonstrates.
- Instructions to generate the SDSoC hardware platform meta-data file.
- Instructions to create platform software libraries, if required.
- Description of the SDSoC software platform meta-data file.
- Basic platform testing.

In addition to these platform examples, it would be worthwhile to inspect the standard SDSoC platforms that are included in the SDx™ IDE in the `<sdx_root>/platforms` directory.

MicroBlaze Hardware Requirements

In addition to targeting Zynq and Zynq® UltraScale+™ MPSoC devices, you can build an SDSoC platform that targets any Xilinx® device by using the MicroBlaze processor as the target CPU (add reference to MicroBlaze™ documentation). A MicroBlaze platform in SDSoC must be a self-contained system containing an LMB memory, MicroBlaze Debug Module (MDM), UART, and AXI Timer built using the Vivado Design Suite and SDK. A sample MicroBlaze example is in `<sdx_root>/sample/platforms/arty` folder. The figure below shows a minimal system. Notice that the JTAG UART is enabled in the MDM and appears as an AXI-Lite slave. The system runs on a clock (sys) delivered from the board.

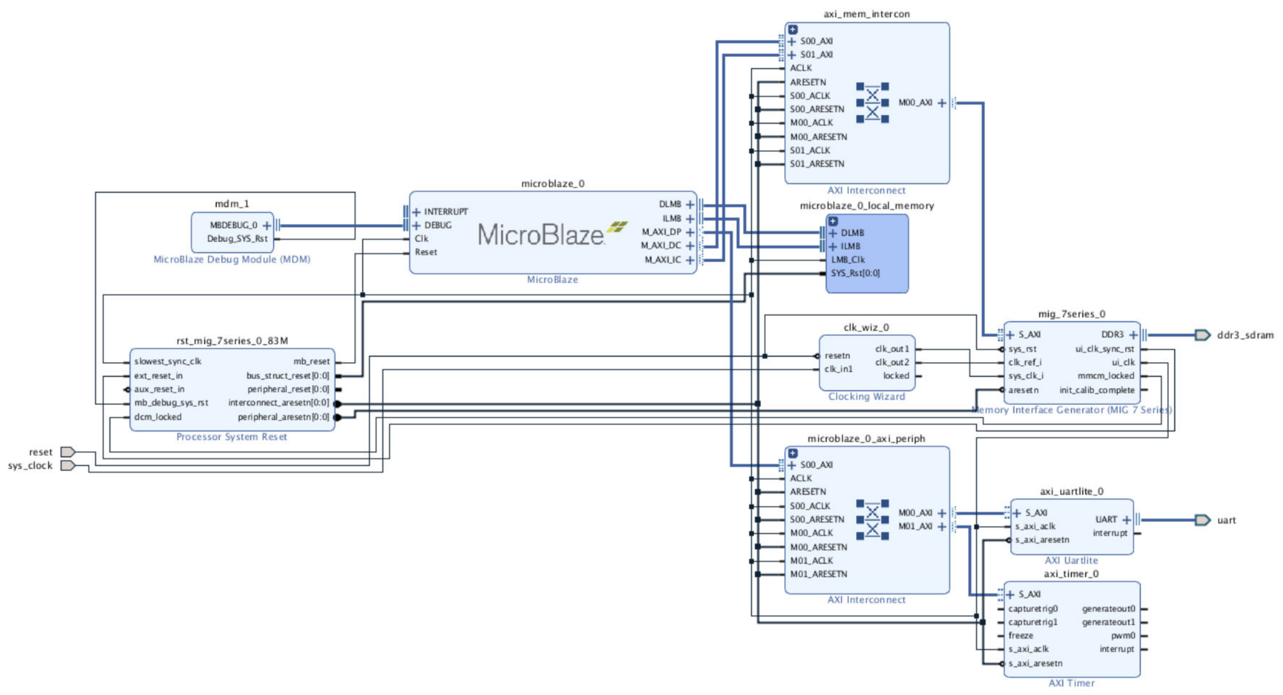
Figure 74: **Minimal MicroBlaze System**



The SDSoC runtime requires the platform hardware to include two IPs: a timer for the `sds_clock_counter()` API, and a UART to print runtime error messages. Since a MicroBlaze processor has a single AXI Master port (`M_AXI_DP`) to control AXI slaves, a MicroBlaze platform must include an AXI interconnect connected to this port as described in Example: Sharing a Platform IP AXI Port.

MicroBlaze systems that use DDR typically connect the processor to the DDR via MIG using the cache facilities built into the MicroBlaze. An example system is shown below.

Figure 75: **MicroBlaze System with MIG**



In the system above, the MicroBlaze is configured with an 8 KB cache. The instruction and data cache ports (`M_AXI_DC` and `M_AXI_IC`) are connected to the MIG. The UART is connected to an on-board USB-to-UART converter chip and does not use the JTAG UART from the previous design. The system runs on the user-interface (UI) clock produced by MIG. Other than these differences, the MicroBlaze systems are identical.

Ports on the MIG's AXI Interconnect IP can be registered using PFM properties as described in Declaring AXI Ports for the type `M_AXI_HP` which is exactly the same as Zynq HP Ports. SDSoC runtime will invalidate or flush buffers in exactly the same way as it does for Zynq devices.

Example: Direct I/O in an SDSoC Platform

An SDSoC platform can include input and output subsystems; for example, analog-to-digital and digital-to-analog converters, or video I/O, by converting raw physical data streams into AXI4-Stream interfaces that are exported as part of the platform interface specification. For information on the `zc702_axis_io` sample platform, see "Using External I/O" in the *SDSoC Environment User Guide* (UG1027). This example includes sample applications that demonstrate how an input data stream can be written directly into memory buffers without data loss, and how an application can "packetize" the data stream at the AXI transport level to communicate with other functions (including, but not limited to DMAs) that require packet framing.

RECOMMENDED: *The source code for this platform can be found in* `<sdx_root>/samples/platforms/zc702_axis_io/src`.

Run the following command from the command shell using `zc702_axis_io_dsa.tcl`, a Vivado Tcl script to build the hardware platform in a batch mode:

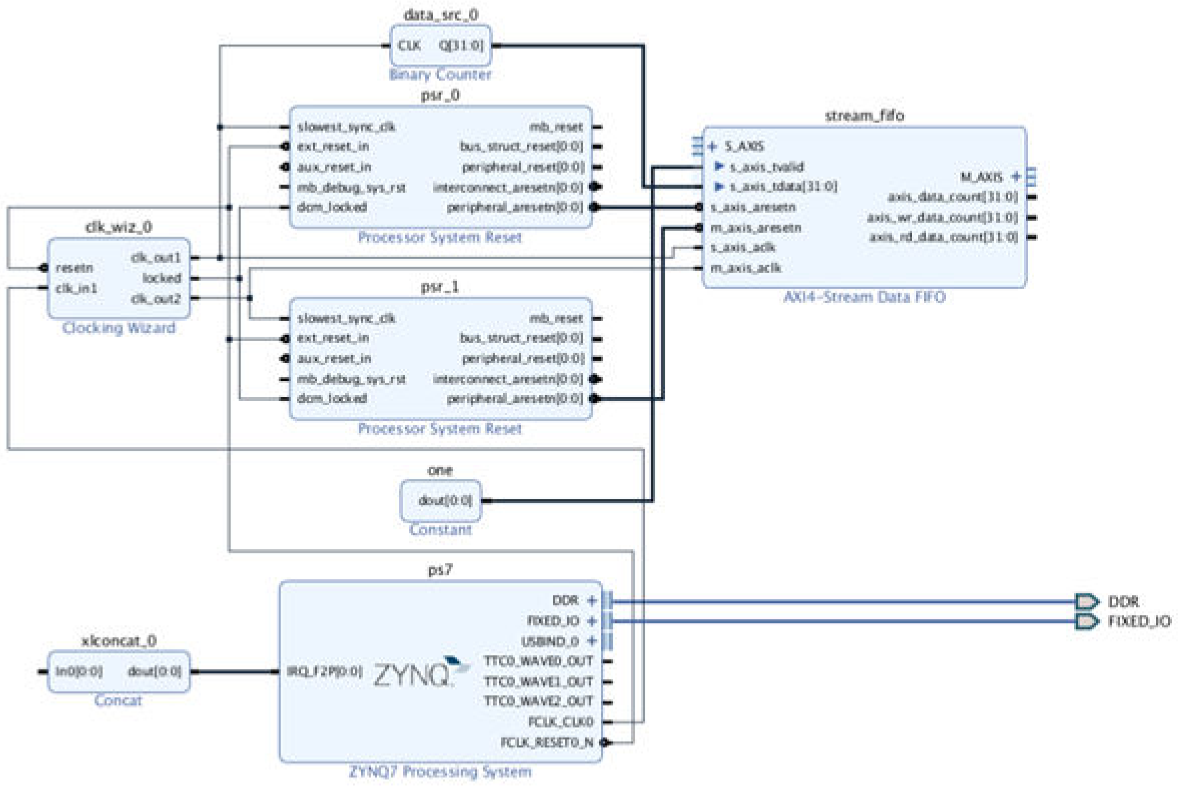
```
vivado -mode batch -source zc702_axis_io_dsa.tcl
```

Run the following command to build the platform in GUI mode to inspect the hardware system in the Vivado IP integrator:

```
vivado -mode gui -source zc702_axis_io_dsa.tcl
```

The command opens the Vivado IDE and builds the platform. The resulting hardware system will look similar to the following block diagram.

Figure 76: **zc702_axis_io Block Diagram**



To make this design portable, this platform includes a free-running binary counter that generates a continuous stream of data samples at 50 MHz, which acts as a proxy for data streaming directly from FPGA pins. To convert this input data stream into an AXI4-Stream for SDSoC applications, the platform connects the counter output to the `s_axis_tdata` slave port of an AXI4-Stream data FIFO, with a constant block providing the required `s_axis_tvalid` signal, always one. The data FIFO IP is configured to store up to 1024 samples with an output clock of 100 MHz to provide system elasticity so that the consumer of the stream can process the stream "bubble-free" (i.e., without dropping data samples). In a real platform, the means for converting to an AXI4-Stream, relative clocking and amount of hardware buffering will vary according to system requirements.

Similar to input streaming off of an analog-to-digital converter, this data stream is not packetized, which means the AXI4-Stream has no `TLAST` signal. Consequently, any SDSoC application that consumes the data stream must be capable of handling unpacketized streams. Within the SDSoC environment, every data mover IP core (e.g., the Vivado AXI4 Direct Memory Access IP (AXI DMA)) requires packetized AXI4-Streams that include the `TLAST` signal. To consume the streaming input from this platform, an application must employ direct hardware connections to the AXI4-Stream port.

TIP: *A platform can also export an AXI4-Stream port that includes the `TLAST` signal, in which case SDSoC applications do not require direct connections to the port.*

Declaring the SDSoC Hardware Platform Interface

As described in Chapter 3: Creating the Platform Hardware Component, the hardware platform port interface is defined by setting PFM properties on cells and ports within a Vivado IP integrator block diagram.

In the `zc702_axis_io_dsa.tcl` script, this occurs in lines 45-67. Use the following steps to set the properties on cells and ports within a Vivado IP integrator block diagram.

1. Declare the platform name as an IP-XACT VLNV (vendor:library:name:version) string using the following commands:

```
set_property PFM_NAME \
"xilinx.com:zc702_axis_io:zc702_axis_io:1.0" \
[get_files ./zc702_axis_io_vivado/zc702_axis_io.srcs/\
sources_1/bd/zc702_axis_io/zc702_axis_io.bd]
```

2. Declare a platform clock with id 1 using the following commands:

```
set_property PFM.CLOCK { \
clk_out2 {id "1" is_default "true" proc_sys_reset "psr_1" } \
    } [get_bd_cells /clk_wiz_0]
```

   Note that every clock must have an associated `proc_sys_reset` that provides synchronized reset signals for blocks using this clock.

3. At least, one general purpose AXI master and one AXI slave port must be declared. Use the following command to declare the platform AXI interfaces, each with an associative list containing several attributes.

```
set_property PFM.AXI_PORT { \
M_AXI_GP0 {memport "M_AXI_GP"} \
M_AXI_GP1 {memport "M_AXI_GP"} \
S_AXI_ACP {memport "S_AXI_ACP" sptag "ACP"
memory "ps7 ACP_DDR_LOWOCM"} \
S_AXI_HP0 {memport "S_AXI_HP" sptag "HP0"
memory ps7 HP0_DDR_LOWOCM"} \
S_AXI_HP1 {memport "S_AXI_HP" sptag "HP1"
memory ps7 HP1_DDR_LOWOCM"} \
S_AXI_HP2 {memport "S_AXI_HP" sptag "HP2"
memory ps7 HP2_DDR_LOWOCM"} \
S_AXI_HP3 {memport "S_AXI_HP" sptag "HP3"
memory ps7 HP3_DDR_LOWOCM"} \
} [get_bd_cells /ps7]
```

Each AXI port requires a memport memory type declaration, which must be one of the following:

a. `M_AXI_GP` – a general purpose master

b. `S_AXI_ACP` – a cache coherent slave

c. `S_AXI_HP` – a high performance, non-cache coherent slave

d. `S_AXI_HPC` – a high performance slave (Zynq UltraScale+ MPSoC only)

e. `MIG` – a slave on an external DDR (MIG) memory controller IP

An AXI slave port requires an sptag that provides a symbolic tag to represent the port, and two additional memory attributes.

f. Memory instance: The cell name of the block in the IP integrator address editor

g. Address segment: The 'Base Name' associated with the port as seen in the Vivado IP integrator address editor

4. Use the following command to declare the `stream_fifo` master AXI4-Stream port.

```
set_property PFM.AXIS_PORT { \
    M_AXIS {type "M_AXIS"} \
    } [get_bd_cells /stream_fifo]
```

5. Use the following command to declare the interrupt inputs by constructing a list of port names on a `Concat` block that is connected to the interrupt port on the `processing_system7` block:

```
set intVar []
for {set i 0} {$i < 16} {incr i} {
    lappend intVar In$i {}
}
set_property PFM.IRQ $intVar [get_bd_cells /xlconcat_0]
```

6. After declaring the port interface, use the following command to generate the output products required to create the DSA from the block diagram:

   ```
   generate_target all \
   [get_files ./zc702_axis_io_vivado/zc702_axis_io.srcs/\
   sources_1/bd/zc702_axis_io/zc702_axis_io.bd]
   ```

7. Use the following command to generate the DSA:

   ```
   write_dsa -force ./zc702_axis_io.dsa
   ```

Making the SDSoC Platform from the Command Line

As described in Software Platform Data Creation, the following platform components provide the application run time context:

- Bootloaders
- Operating system
- File system

The `<sdx_root>/samples/platforms/zc702_axis_io/src/zc702_axis_io_pfm.tcl` file is a a tcl script that builds the SDSoC platform by incorporating the DSA hardware and software components that were built using PetaLinux and SDx (SDK style first-stage boot loader project).

Run the following command from an SDSoC command shell to build the platform in batch mode using `zc702_axis_io_pfm.tcl` SDx tcl script which is executed by the `xsct` utility provided as part of SDx.:

```
xsct -sdx  ./zc702_axis_io_dsa.tcl
```

1. Use the following command to create a platform object in the `xsct` command line interpreter:

   ```
   platform -name zc702_axis_io \
   -desc "Zynq ZC702 Board with direct I/O" \
   -hw ./zc702_axis_io.dsa -out ./output \
   -prebuilt  -samples samples
   ```

2. Use the following command to create a new system configuration for Linux applications:

   ```
   system -name linux -display-name "Linux"  \
   -boot ./boot  -readme ./generic.readme
   ```

3. Use the following command to define a processor group or domain for this system configuration:

   ```
   domain -name linux -proc ps7_cortexa9_0 \
   -os linux -image ./linux/image
   ```

4. Use the following command to register boot files for the Linux system configuration:

   ```
   boot -bif ./linux/linux.bif
   ```

5. Use the following command to register QEMU arguments and a directory containing boot files to support SDSoC emulation:

   ```
   domain -qemu-args ./qemu/lnx/qemu_args.txt
   domain -qemu-data ./boot
   ```

6. Use the following command to create and populate a standalone ('bare metal') system configuration:

   ```
   system -name standalone  -display-name "Standalone" -boot ./boot  -
   readme ./generic.readme
   domain -name standalone -proc ps7_cortexa9_0 -os standalone
   app -lscript ./standalone/lscript.ld
   boot -bif ./standalone/standalone.bif
   domain -qemu-args ./qemu/std/qemu_args.txt
   domain -qemu-data ./boot
   ```

7. Use the following command to create the SDSoC platform:

   ```
   platform -generate
   ```

   This script creates an SDK Platform project called `zc702_axis_io` in the following directory:

   ```
   output/zc702_axis_io/export/
   ```

Platform Sample Designs

An SDSoC platform can include sample applications that demonstrate its use, as described in Chapter 5: Sample Applications. The SDx IDE looks for a file called `samples/<example>/description.json` (`template.xml`) for information on the sample application within a platform. The `template.xml` file for the `zc702_axis_io` platform lists several test applications, each of which is of specific interest.

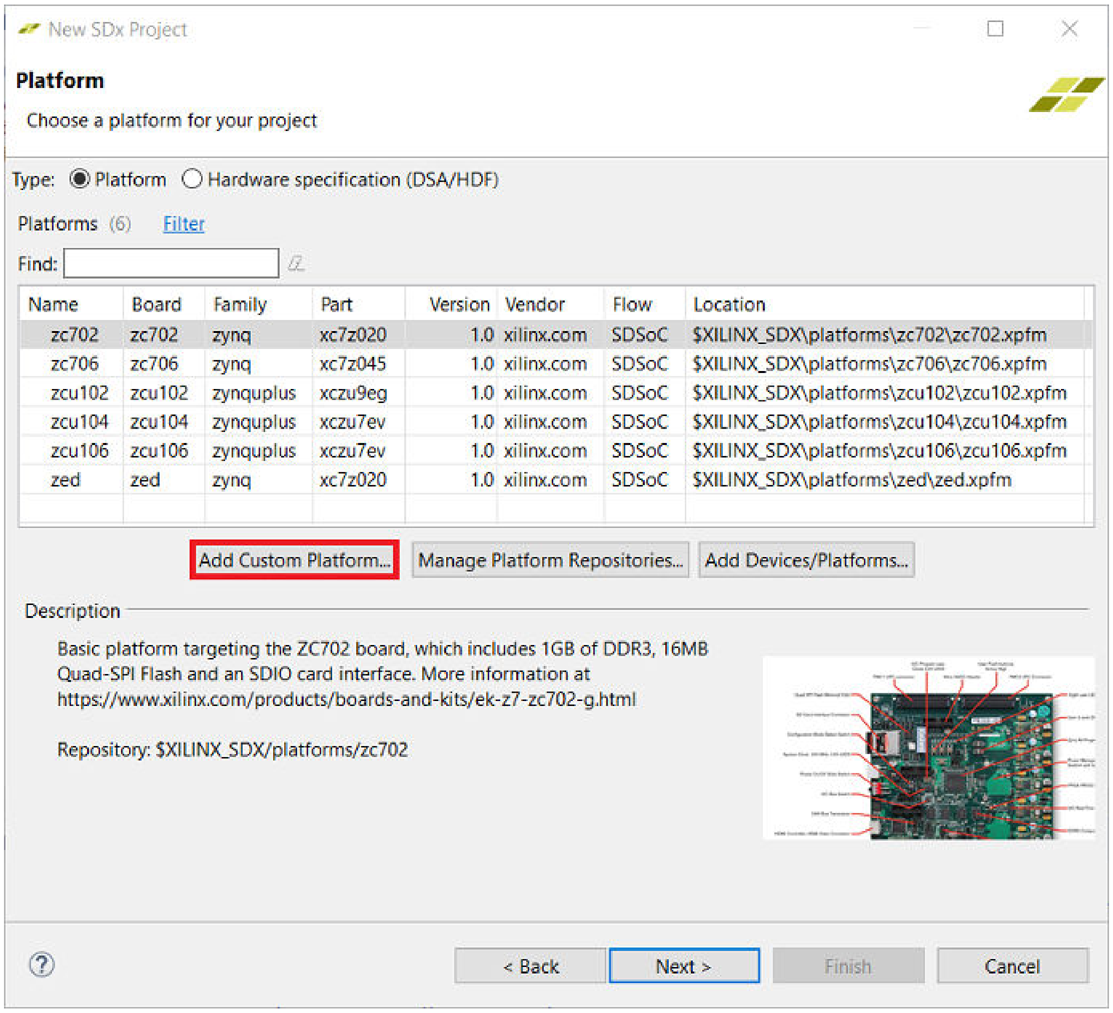
```
<template location="aximm" name="Unpacketized AXI4-Stream to DDR"
             description="Shows how to copy unpacketized AXI4-Stream data
directly to DDR.">
        <supports>
            <and>
                <or>
                    <os name="Linux"/>
                    <os name="Standalone"/>
                </or>
            </and>
        </supports>
        <accelerator name="s2mm_data_copy" location="main.cpp"/>
    </template>
    <template location="stream" name="Packetize an AXI4-Stream"
                 description="Shows how to packetize an unpacketized AXI4-
Stream.">
        <supports>
            <and>
```

```
                <or>
                    <os name="Linux"/>
            <os name="Standalone"/>
                </or>
            </and>
        </supports>
        <accelerator name="packetize" location="packetize.cpp"/>
        <accelerator name="minmax" location="minmax.cpp"/>
    </template>
    <template location="pull_packet" name="Lossless data capture from AXI4-
Stream to DDR"
              description="Illustrates a technique to enable lossless data
capture from a free-running input source.">
        <supports>
            <and>
                <or>
                    <os name="Linux"/>
                    <os name="Standalone"/>
                </or>
            </and>
        </supports>
        <accelerator name="PullPacket" location="main.cpp"/>
    </template>
```

To use a platform in the SDx IDE, you must add it to the platform repository for the Eclipse workspace as described in the following steps.

1. Launch Xilinx SDx and provide a path to your workspace such as `<path_to_tutorial>/myplatforms/`.
2. Create a new project by selecting **File → New → Xilinx SDx Project**.
3. Specify the type of project as an **Application Project**, and click **Next**.
4. Specify a project name in the Create New SDx Project page such as `my_zc702_axis_io`, and click **Next**.
5. In the Choose Hardware Platform page click **Add Custom Platform**.

Figure 77: **Add Custom Platform**



New SDx Project

Platform

Choose a platform for your project

Type: [● Platform] ○ Hardware specification (DSA/HDF)

Platforms (6) Filter

Find:

Name	Board	Family	Part	Version	Vendor	Flow	Location
zc702	zc702	zynq	xc7z020	1.0	xilinx.com	SDSoC	$XILINX_SDX\platforms\zc702\zc702.xpfm
zc706	zc706	zynq	xc7z045	1.0	xilinx.com	SDSoC	$XILINX_SDX\platforms\zc706\zc706.xpfm
zcu102	zcu102	zynquplus	xczu9eg	1.0	xilinx.com	SDSoC	$XILINX_SDX\platforms\zcu102\zcu102.xpfm
zcu104	zcu104	zynquplus	xczu7ev	1.0	xilinx.com	SDSoC	$XILINX_SDX\platforms\zcu104\zcu104.xpfm
zcu106	zcu106	zynquplus	xczu7ev	1.0	xilinx.com	SDSoC	$XILINX_SDX\platforms\zcu106\zcu106.xpfm
zed	zed	zynq	xc7z020	1.0	xilinx.com	SDSoC	$XILINX_SDX\platforms\zed\zed.xpfm

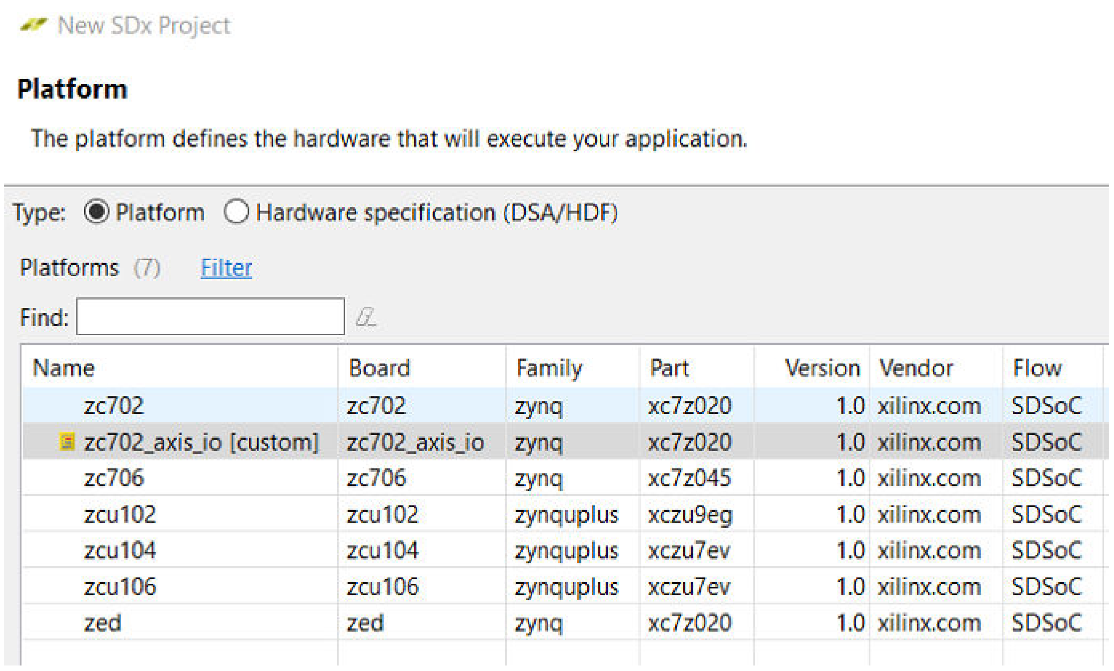
Description

Basic platform targeting the ZC702 board, which includes 1GB of DDR3, 16MB Quad-SPI Flash and an SDIO card interface. More information at https://www.xilinx.com/products/boards-and-kits/ek-z7-zc702-g.html

Repository: $XILINX_SDX/platforms/zc702

6. Navigate to the folder containing the platform `<sdx_root>/samples/platforms/zc702_axis_io`.
7. The platform will show up in the Choose Hardware Platform Page. Select `zc702_axis_io (custom)` and click **Next**.

Figure 78: **Choose Hardware Platform**


New SDx Project

Platform

The platform defines the hardware that will execute your application.

Type: [✓ Platform] Hardware specification (DSA/HDF)

Platforms (7) Filter

Find:

Name	Board	Family	Part	Version	Vendor	Flow
zc702	zc702	zynq	xc7z020	1.0	xilinx.com	SDSoC
zc702_axis_io [custom]	zc702_axis_io	zynq	xc7z020	1.0	xilinx.com	SDSoC
zc706	zc706	zynq	xc7z045	1.0	xilinx.com	SDSoC
zcu102	zcu102	zynquplus	xczu9eg	1.0	xilinx.com	SDSoC
zcu104	zcu104	zynquplus	xczu7ev	1.0	xilinx.com	SDSoC
zcu106	zcu106	zynquplus	xczu7ev	1.0	xilinx.com	SDSoC
zed	zed	zynq	xc7z020	1.0	xilinx.com	SDSoC

8. On the System Configuration page, keep the default **Linux** for System Configuration and click **Next**.

9. On the Templates page, select `Unpacketized AXI4-Stream to DDR` to test the platform with one of the sample applications, and click **Finish**.

   The `s2mm_data_copy` function is pre-selected for hardware acceleration. The program data flow within `s2mm_data_copy_wrapper` creates a direct signal path from the platform input to a hardware function called `s2mm_data_copy` that then pushes the data to memory as a `zero_copy` datamover. That is, the `s2mm_data_copy` function acts as a custom DMA. The main program allocates four buffers, invokes `s2mm_data_copy_wrapper`, and then checks the written buffers to ensure that data values are sequential, i.e., the data is written bubble-free. For simplicity, this program does not reset the counter, so the initial value depends upon how much time elapses between board power-up and invoking the program.

10. Open up `main.cpp`. Key points to observe are:

    - Buffers are allocated using `sds_alloc` to guarantee physically contiguous allocation required for the zero_copy datamover.

```
unsigned *bufs[NUM_BUFFERS];
unsigned* rbuf0;
for(int i=0; i<NUM_BUFFERS; i++) {
    bufs[i] = (unsigned*) sds_alloc(BUF_SIZE *
 sizeof(unsigned));
}
// Flush the platform FIFO of start-up garbage
s2mm_data_copy(rbuf0, bufs[0]);
```

```
s2mm_data_copy(rbuf0, bufs[0]);
s2mm_data_copy(rbuf0, bufs[0]);
for(int i=0; i<NUM_BUFFERS; i++) {
  s2mm_data_copy(rbuf0, bufs[i]);
}
```

- Specify the connectivity between hardware function and the platform using the `sys_port` pragma.

```
// s2mm "DMA" accelerator
#pragma SDS data sys_port (fifo:stream_fifo_M_AXIS)
#pragma SDS data zero_copy(buf)
int s2mm_data_copy(unsigned *fifo, unsigned buf[BUF_SIZE])
{
#pragma HLS interface axis port=fifo
     for(int i=0; i<BUF_SIZE; i++) {
#pragma HLS pipeline
          buf[i] = *fifo;
     }
     return 0;
}
```

11. Build the application by clicking on the Build icon in the toolbar. When the build completes, the `Debug` folder contains an `sd_card` folder with the boot image and application ELF.
12. After the build finishes, copy the contents of the `sd_card` directory onto an SD card, boot, and run `my_zc702_axis_io.elf`.

```
sh-4.3# cd /mnt
sh-4.3# ./my_zc702_axis_io.elf
TEST PASSED!
sh-4.3#
```

Example: Sharing a Platform IP AXI Port

To share an AXI master (slave) interface between a platform IP and the accelerator and data motion IPs generated by the SDSoC compilers, employ the SDSoC Tcl API to declare the first unused AXI master (slave) port (in index order) on the AXI interconnect IP block connected to the shared interface. Your platform must use each of the lower indexed masters (slaves) on this AXI interconnect.

SDSoC Platform Hardware Interface

Use the following steps to build the SDSoC hardware platform interface within a Vivado IDE:

Note: The source code for this platform is available in `<sdx_root>/samples/platforms/zc702_acp/src` file.

1. Run the following command from the command shell using `zc702_acp_dsa.tcl`, a Vivado tcl script to build the hardware platform in a batch mode:

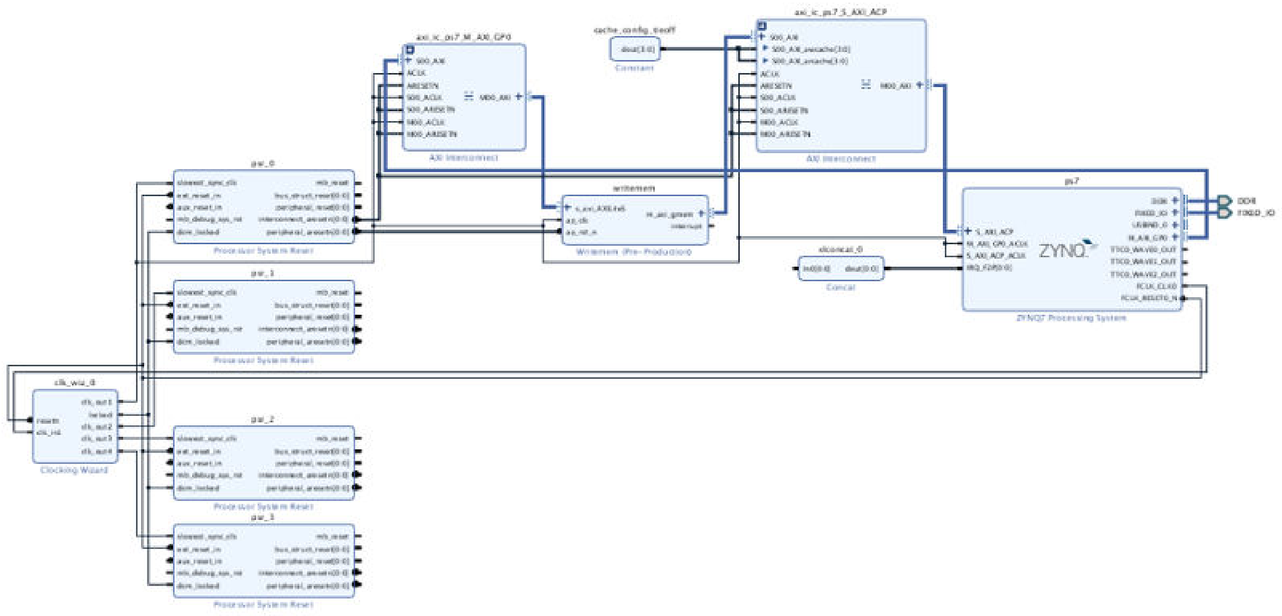
   ```
   vivado -mode batch -source zc702_acp_dsa.tcl
   ```

   You can also build the platform in GUI mode to inspect the hardware system in Vivado IP integrator. Run the following command to build the platform in GUI mode:

   ```
   vivado -mode gui -source zc702_acp_dsa.tcl
   ```

   This command will open the Vivado IDE and build the platform. The resulting hardware system will look similar to the following block diagram.

   Figure 79: **zc702_acp Block Design**



2. Use the following commands to declare the platform name as an IP-XACT VLNV (vendor:library:name:version) string:

   ```
   set_property PFM_NAME \
   "xilinx.com:zc702_acp:zc702_acp:1.0" \
   [get_files ./zc702_acp_vivado/zc702_acp.srcs/\
   sources_1/bd/zc702_acp/zc702_acp.bd]
   ```

3. Use the following command to declare a platform clock with id 1:

   ```
   set_property PFM.CLOCK { \
     clk_out1 {id "2" is_default "true" proc_sys_reset "psr_0" } \
     clk_out2 {id "1" is_default "false" proc_sys_reset "psr_1" } \
     clk_out3 {id "0" is_default "false" proc_sys_reset "psr_2" } \
     clk_out4 {id "3" is_default "false" proc_sys_reset "psr_3" } \
   } [get_bd_cells /clk_wiz_0]
   ```

Note that every clock must have an associated `proc_sys_reset` that provides synchronized reset signals for blocks using this clock.

4. At least one general purpose AXI master and one AXI slave port must be declared. Use the following command to declare the platform AXI interfaces from the processing system IP, each with an associative list containing several attributes:

```
set_property PFM.AXI_PORT { \
M_AXI_GP1 {memport "M_AXI_GP"} \
S_AXI_HP0 {memport "S_AXI_HP" sptag "HP0"
memory ps7 HP0_DDR_LOWOCM"} \
S_AXI_HP1 {memport "S_AXI_HP" sptag "HP1"
memory ps7 HP1_DDR_LOWOCM"} \
S_AXI_HP2 {memport "S_AXI_HP" sptag "HP2"
memory ps7 HP2_DDR_LOWOCM"} \
S_AXI_HP3 {memport "S_AXI_HP" sptag "HP3"
memory ps7 HP3_DDR_LOWOCM"} \
} [get_bd_cells /ps7]
```

   Each AXI port requires a memport memory type declaration, which must be one of the following:

   - `M_AXI_GP` – a general purpose master
   - `S_AXI_ACP` – a cache coherent slave
   - `S_AXI_HP` – a high performance, non-cache coherent slave
   - `S_AXI_HPC` – a high performance slave (Zynq UltraScale+ MPSoC only)
   - `MIG` – a slave on an external DDR (MIG) memory controller IP

     An AXI slave port requires an sptag that provides a symbolic tag to represent the port, and two additional memory attributes.

   - Memory instance – the cell name of the block in the IP integrator address editor
   - Address segment – the 'Base Name' associated with the port as seen in the Vivado IP integrator address editor

5. The platform uses both the `S_AXI_ACP` and `M_AXI_GP0` ports on the processing system. Use the following Tcl code to declare additional ports on the axi_interconnect IPs within the platform:

```
set gpMasters []
for {set i 1} {$i < 64} {incr i} {
  lappend gpMasters M[format %02d $i]_AXI {memport "M_AXI_GP"}
}
set_property PFM.AXI_PORT $gpMasters \
[get_bd_cells /axi_ic_ps7_M_AXI_GP0]

set acpSlaves []
for {set i 1} {$i < 8} {incr i} {
  lappend acpSlaves S[format %02d $i]_AXI {memport "S_AXI_ACP" \
sptag "ACP" memory "ps_ACP_DDR_LOWOCM"}
}
set_property PFM.AXI_PORT $acpSlaves \
[get_bd_cells /axi_ic_ps7_S_AXI_ACP]
```

Note that the memport attribute is inherited from the processing system port connected to the axi_interconnect.

6. Use the following command declares the interrupt inputs by constructing a list of port names on a Concat block that is connected to the interrupt port on the processing_system7 block:

```
set intVar []
for {set i 0} {$i < 16} {incr i} {
    lappend intVar In$i {}
}
set_property PFM.IRQ $intVar [get_bd_cells /xlconcat_0]
```

7. After declaring the port interface, use the following command to generate the output products required to create the DSA from the block diagram:

```
generate_target all \
[get_files ./zc702_acp_vivado/zc702_acp.srcs/\
sources_1/bd/zc702_acp/zc702_acp.bd]
```

8. Finally, use the following command to generate the DSA:

```
write_dsa -force ./zc702_acp.dsa
```

Appendix F

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator (DocNav) provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. DocNav is installed with the SDSoC™ and SDAccel™ development environments. To open it:

- On Windows, select **Start → All Programs → Xilinx Design Tools → DocNav**.
- At the Linux command prompt, enter `docnav`.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In DocNav, click the **Design Hubs View** tab.
- On the Xilinx website, see the Design Hubs page.

Note: For more information on DocNav, see the Documentation Navigator page on the Xilinx website.

References

These documents provide supplemental material useful with this guide:

1. *SDSoC Environments Release Notes, Installation, and Licensing Guide* (UG1294)
2. *SDSoC Environment User Guide* (UG1027)
3. *SDSoC Environment Getting Started Tutorial* (UG1028)
4. *SDSoC Environment Tutorial: Platform Creation* (UG1236)
5. *SDSoC Environment Platform Development Guide* (UG1146)
6. *SDSoC Environment Profiling and Optimization Guide* (UG1235)
7. *SDx Command and Utility Reference Guide* (UG1279)
8. *SDSoC Environment Programmers Guide* (UG1278)
9. *SDSoC Environment Debugging Guide* (UG1282)
10. *SDx Pragma Reference Guide* (UG1253)
11. *UltraFast Embedded Design Methodology Guide* (UG1046)
12. *Zynq-7000 SoC Software Developers Guide* (UG821)
13. *Zynq UltraScale+ MPSoC: Software Developers Guide* (UG1137)
14. *ZC702 Evaluation Board for the Zynq-7000 XC7Z020 SoC User Guide* (UG850)
15. *ZCU102 Evaluation Board User Guide* (UG1182)
16. *PetaLinux Tools Documentation: Reference Guide* (UG1144)
17. *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)
18. *Vivado Design Suite User Guide: Creating and Packaging Custom IP* (UG1118)
19. SDSoC Development Environment web page
20. Vivado® Design Suite Documentation

Training Resources

1. SDSoC Development Environment and Methodology
2. Advanced SDSoC Development Environment and Methodology

Please Read: Important Legal Notices

AUTOMOTIVE APPLICATIONS DISCLAIMER

Copyright