SDSoC Environment Platform Development Guide

UG1146 (v2017.4) January 26, 2018

XILINX ALL PROGRAMMABLE™

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
1/26/2018	2017.4	• Minor edits throughout.
12/20/2017	2017.4	• Replaced SDSoC Platform Utility with SDx GUI Platform Project as described in Chapter 2: Creating SDSoC Platforms. • Complete rewrite of Chapter 3: SDSoC Hardware Platform Creation. • Updated Appendix B: SDSoC Platform Migration. • Updated Appendix C: SDSoC Platform Examples. • Minor edits throughout.
08/16/2017	2017.2	• Updated document to reflect new SDSoC Platform Utility GUI. • Added "MicroBlaze Hardware Requirements." • Removed "Software Requirements" in Chapter 4: Software Platform Data Creation. • Updated Linux Boot Files.

Date	Version	Revision
06/20/2017	2017.1	• Significant edits to Chapter 2: Creating SDSoC Platforms. • Documented the use of the SDSoC Platform Utility. • Expanded information on creating the Vivado Design Suite project and supporting Tcl script in "Hardware Platform Creation." • Eliminated details of the platform metadata files. • Updated process in "Using PetaLinux to Create Linux Boot Files". • Revised content for updating platform files in Appendix B: SDSoC Platform Migration. • Added "Tutorial: Using the SDSoC Platform Utility."

Table of Contents

Chapter 1

Introduction

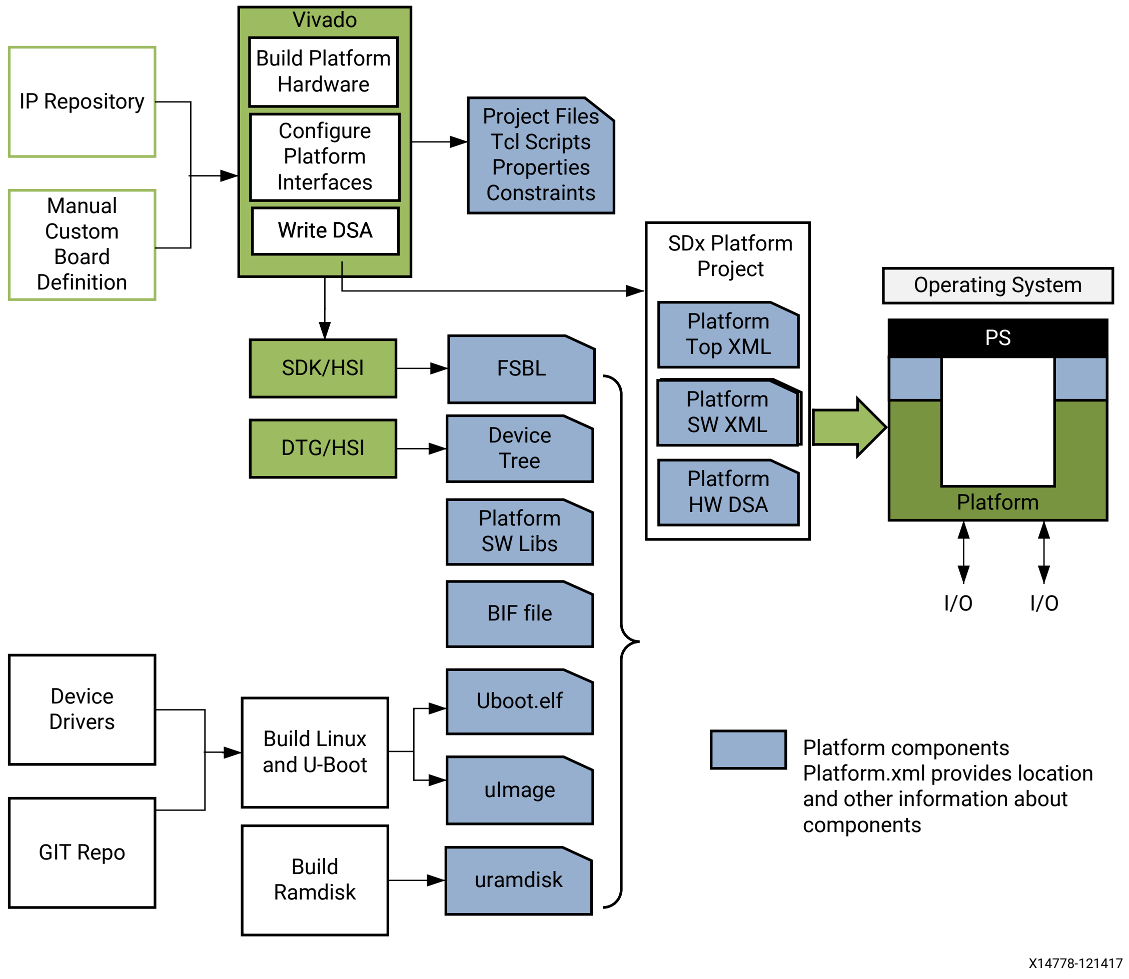
The SDx™ environment is an Eclipse-based integrated development environment (IDE) for implementing heterogeneous embedded systems using Zynq®-7000 All Programmable SoCs and Zynq UltraScale+™ MPSoCs. The SDx IDE supports both the SDSoC (Software-Defined System On Chip) and SDAccel design flows on Linux and only SDSoC flows on Windows. The SDSoC system compiler (sdscc or sds++) generates an application-specific system-on-chip by compiling application code written in C or C++ into hardware and software that extends a target platform. The SDx IDE includes platforms for application development; other platforms are provided by Xilinx partners. Refer to *SDSoC Environment User Guide* (UG1027) for more information.

This document describes how to create a custom SDSoC platform using the SDx IDE to define a platform project and add your custom platform to a repository for use in SDSoC application projects. An SDSoC platform defines a base hardware and software architecture and application context, including processing system, external memory interfaces, custom input/output, and software run time - including operating system (possibly "bare metal"), boot loaders, drivers for platform peripherals and root file system. Every project you create within the SDx IDE targets a specific hardware platform, and you employ the SDSoC tools to customize the platform with application-specific hardware accelerators and data motion networks. In this way, you can easily create highly tailored application-specific systems-on-chip for different base platforms, and can reuse base platforms for many different application-specific systems-on-chip.

A platform developer designs the hardware platform required for use with the SDSoC tools using the Vivado® Design Suite and IP integrator as described in Chapter 3: SDSoC Hardware Platform Creation. This hardware design component is referred to as the Device Support Archive (DSA). The hardware platform design used to create a DSA consists of a Vivado IP integrator subsystem design with the required processor and memory system configuration, and all board interfaces configured and connected to the device I/Os. Platform properties are also defined for platform identification and platform interface configuration.

The platform developer must also provide boot loaders and target operating systems required to boot the platform. A platform can optionally include software libraries to be linked into applications targeting the platform using the SDSoC compilers. If a platform supports a target Linux operating system, you can build the kernel and U-boot bootloader using the PetaLinux tool suite. You can use the PetaLinux tools, SDx IDE or the Xilinx SDK to build platform libraries. Refer to Chapter 4: Software Platform Data Creation for more information on defining the software platform libraries.

Figure 1: **Primary Components of an SDSoC Platform**



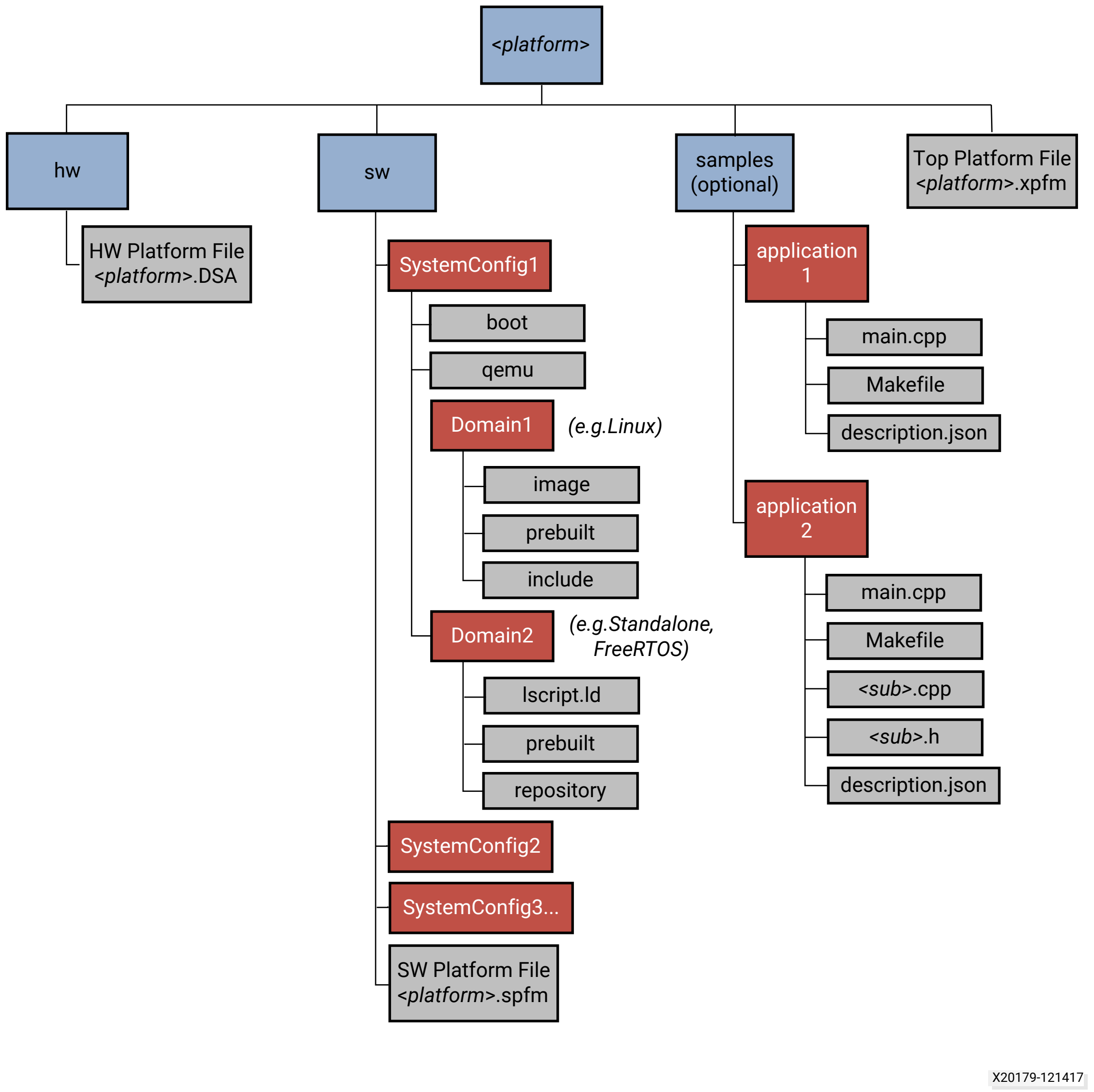
Chapter 2

Creating SDSoC Platforms

An SDSoC platform consists of the following directory structure as shown in the figure below:

- Hardware Folder
  - Device Support Archive file
- Software Folder
  - System Configurations and Processor Domains defining the boot process and OS assignment per processor.
  - Common boot objects (first stage boot loader, for Zynq UltraScale+ MPSoC ARM trusted firmware and power management unit firmware)
  - Linux related objects (u-boot and Linux device tree, kernel and ramdisk as discrete objects or an `image.ub` FIT (Flattened Image Tree) boot image)
  - Pre-built hardware files (optional)
    - Bitstream
    - Exported hardware files for SDK
    - Pre-generated port information software files
    - Pre-generated software interface files
  - Library header files (optional)
  - Static libraries (optional)
- Metadata files generated as part of the platform project.
- Platform sample applications (optional)

Figure 2: **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_path>/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.

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 for one or more applications and a `template.xml` metadata file, users can use the SDx IDE New Project wizard to select and build any of the provided applications. For additional information on application template creation, see Chapter 5: Platform 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 hardware platform DSA as described in* Chapter 3: SDSoC Hardware Platform Creation*, and the software platform files as described in* Chapter 4: Software Platform Data Creation*, available for use in defining the SDSoC platform.*

Launching SDx

***Note*:** Although SDSoC™ supports Linux application development on Windows hosts, a Linux host is strongly recommended for SDSoC platform development, and required for creating a platform supporting a target Linux OS.

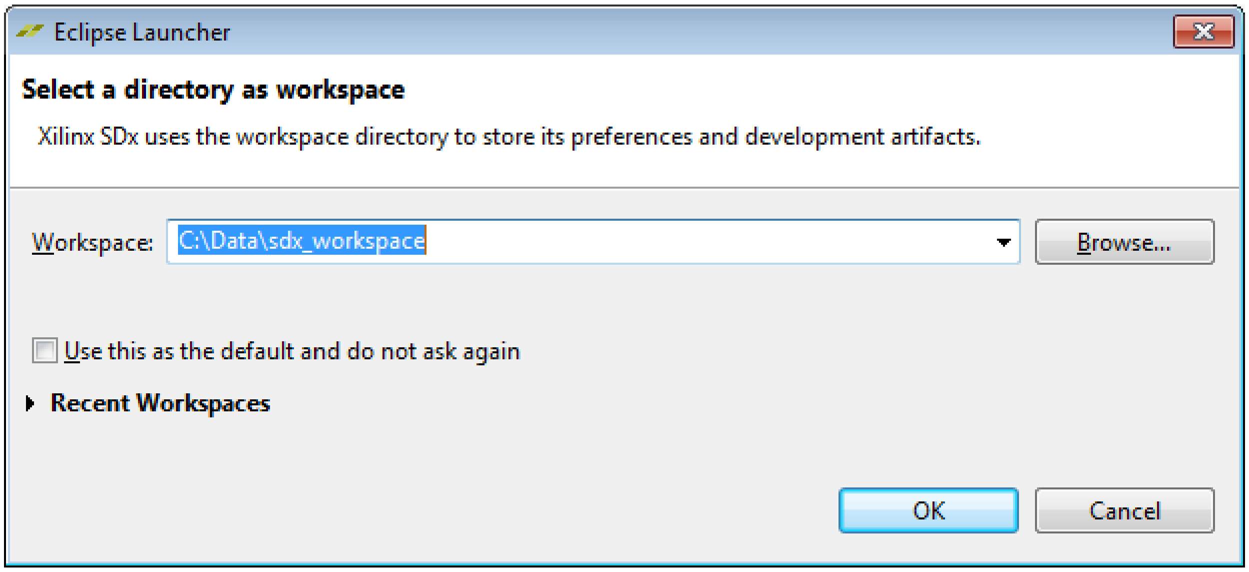
You can launch the SDx IDE directly from the desktop icon, or from the command line by one of the following methods:

- Double-clicking the SDx icon to start the application
- Launching it from the Start menu in the Windows operating system
- Using the following command from the command prompt: sdx

TIP: *Launching the SDx tool from the command line requires that the command shell is configured to run the application, or is an SDx Terminal window launched from the Start menu. To configure a command shell, run the `settings64.bat` file on Windows, or source the `settings64.sh` or `settings64.csh` file on Linux, from the `<install_dir>/SDx/2017.4` directory. Where <install_dir> is the installation folder of the SDx software.*

The SDx IDE opens, and prompts you to select a workspace when you first open the tool.

Figure 3: **Specify SDSoC Workspace**



Eclipse Launcher

Select a directory as workspace

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

Workspace: C:\Data\sdx_workspace

The SDx workspace is the SDx 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, such as the SDSoC project you can create with the following instructions.

1. Use 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** checkbox to set the specified workspace as your default choice and eliminate this dialog box in subsequent uses of SDx.

TIP: *You can always change the current workspace from within the SDx IDE using the* ***File→Switch Workspace*** *command.*

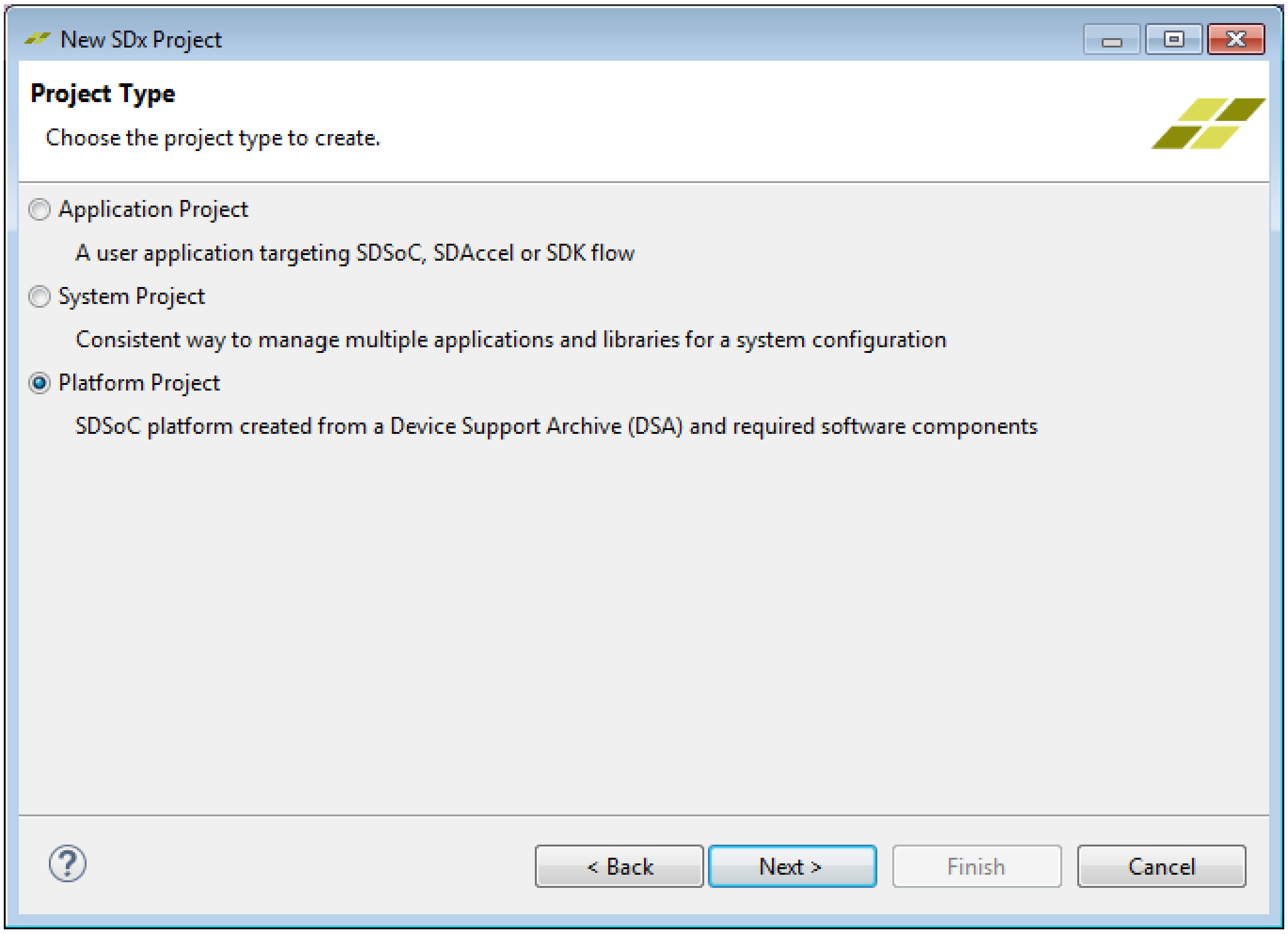
3. When you click **OK**, if this is the first time the SDx IDE has been launched, the **SDx Welcome** screen opens to let you specify an action. Select either **Create SDx Project**, or **File→New→SDx Project**.

Creating an SDSoC Platform Project

TIP: *There are sample platform files including Vivado projects and boot files provided in the SDSoC software installation area at* `<install>/samples/platforms`*. Refer to the SDSoC Environment Tutorial: Platform Creation (* UG1236*) for an example of creating an SDSoC platform.*

After launching the SDx IDE, you can create a new project to define an SDSoC project. The **New SDx Project** wizard opens with the Project Type dialog box to let you specify the type of project to create.

Figure 4: **Specify Project Type**


New SDx Project

Project Type

Choose the project type to create.

( ) Application Project
A user application targeting SDSoC, SDAccel or SDK flow

( ) System Project
Consistent way to manage multiple applications and libraries for a system configuration

(●) Platform Project
SDSoC platform created from a Device Support Archive (DSA) and required software components

SDx supports three project types:

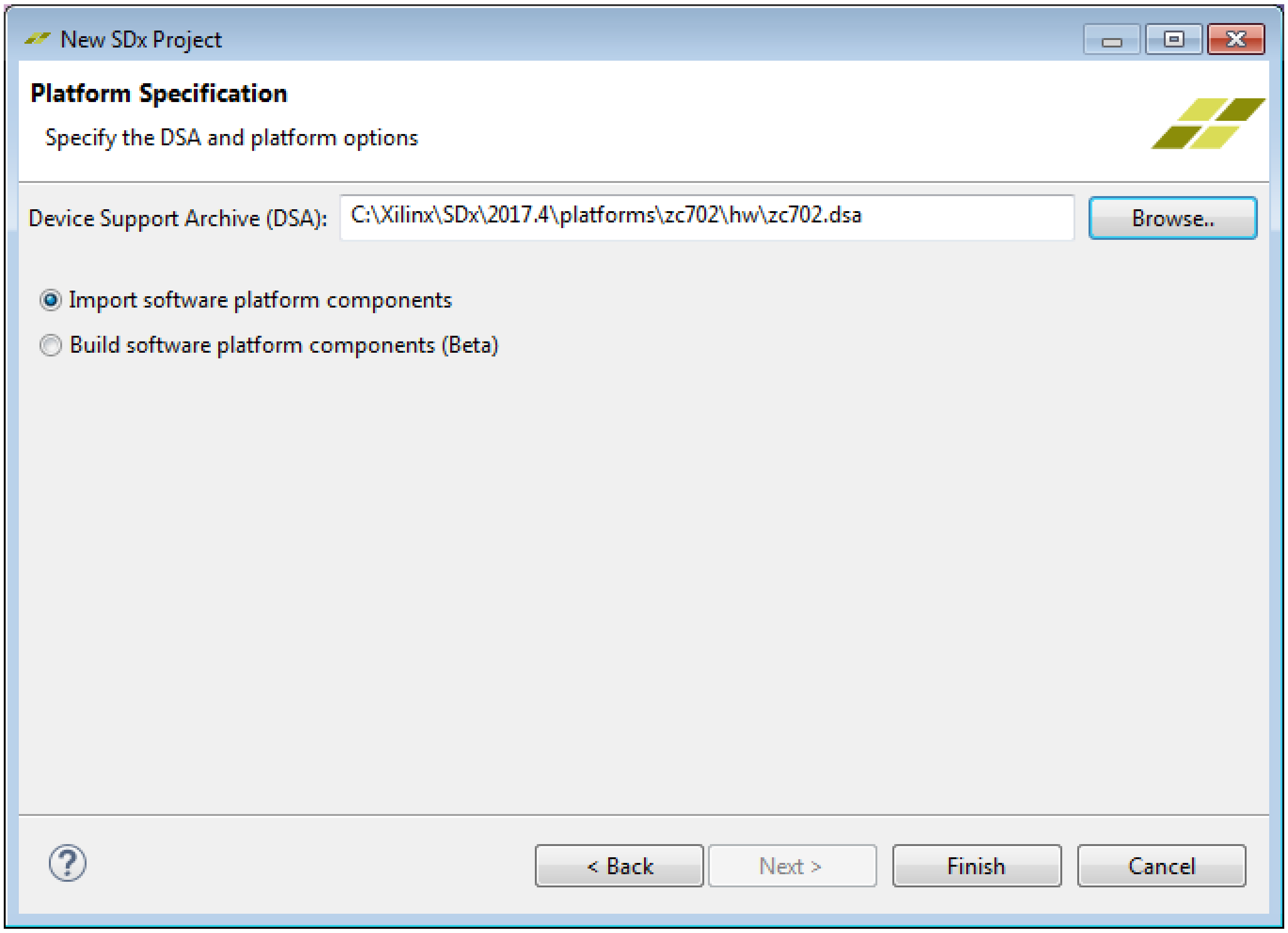
- **Application Project**: A software application with portions of the application accelerated onto one or more hardware kernels on the SDAccel or SDSoC platform.

- **System Project**: Contains multiple application projects to support different applications and hardware kernels running on the platform. While only one project can specify hardware kernels, other projects within the system project can be a "shared library" which can contain HW material used by multiple linking applications.
- **Platform Project**: Defines the hardware and software platform for the SDSoC project.

To define an SDSoC platform project, select **Platform Project** and click **Next**.

The **New SDx Project** wizard opens the Platform Specification page to specify the hardware platform DSA as shown below.

Figure 5: **Specify Hardware Platform DSA**


New SDx Project

Platform Specification

Specify the DSA and platform options

Device Support Archive (DSA): C:\Xilinx\SDx\2017.4\platforms\zc702\hw\zc702.dsa

[✓ Import software platform components]

Build software platform components (Beta)

Specify the **Device Support Archive (DSA)**. 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 hardware platform DSA for a new platform. Refer to Chapter 3: SDSoC Hardware Platform Creation for more information on creating the DSA.

Leave **Import software platform components** enabled at this time. This will let you define the software components of your platform from within the newly created SDSoC platform project.

Click **Finish** to create the project.

The platform project will open in the SDx IDE and will be named based on the DSA file you specified.

Defining the Software Platform

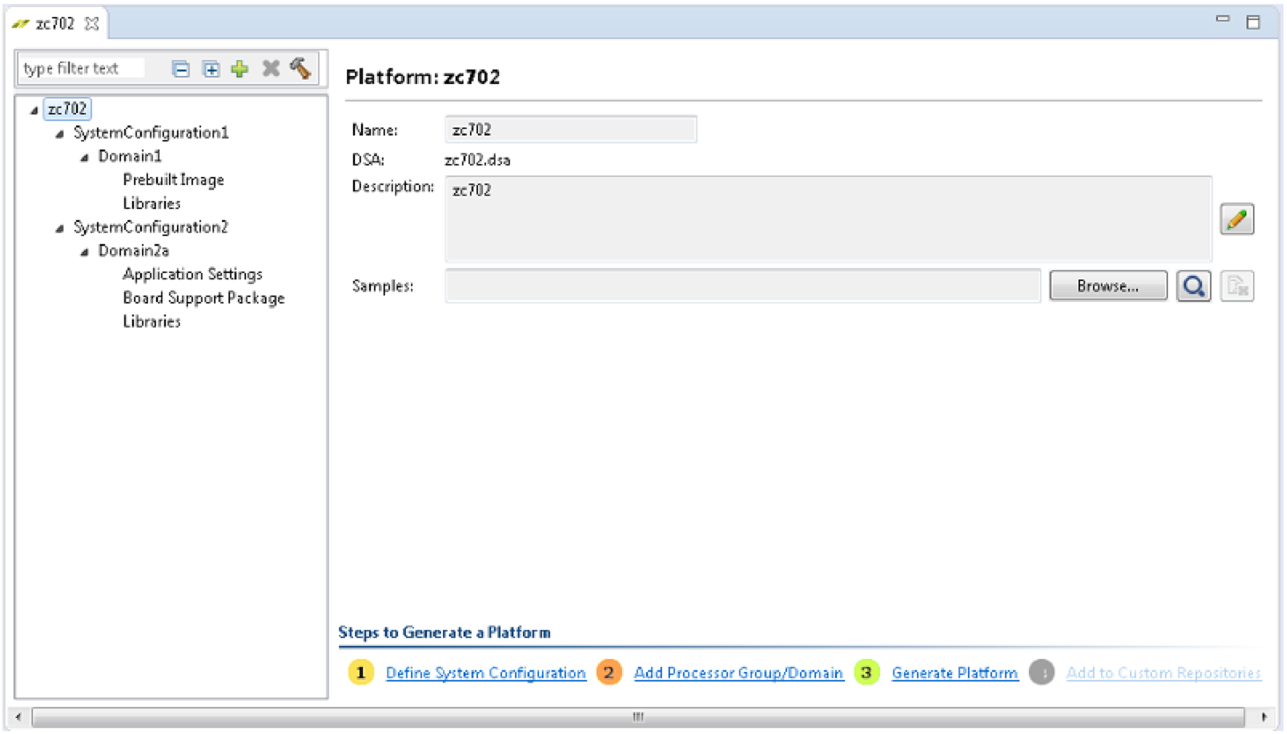
SDx IDE

The SDx IDE 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).

Figure 6: **Editor View**



Platform: zc702

Name: zc702
DSA: zc702.dsa
Description: zc702
Samples:

In the Editor view, as shown above, 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 figure above 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: Platform 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.*

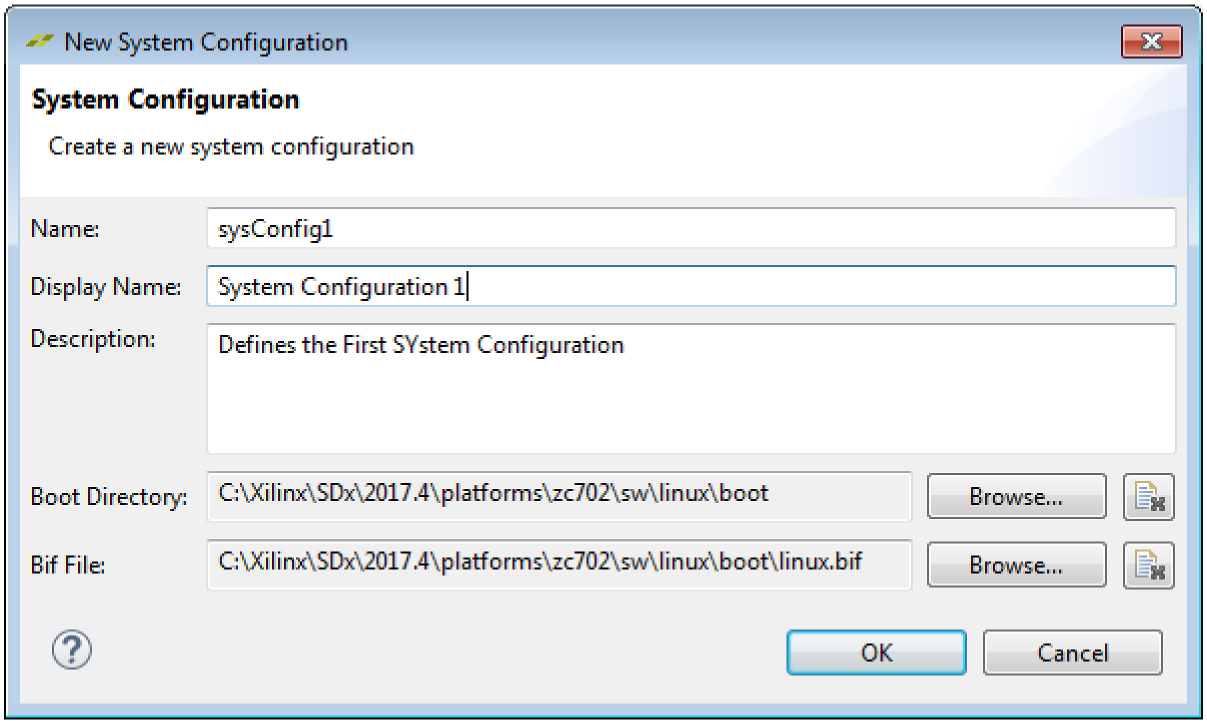
Defining the System Configuration

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.*

Figure 7: **System Configuration**


New System Configuration

System Configuration

Create a new system configuration

Name: sysConfig1

Display Name: System Configuration 1

Description: Defines the First SYstem Configuration

Boot Directory: C:\Xilinx\SDx\2017.4\platforms\zc702\sw\linux\boot

Bif File: C:\Xilinx\SDx\2017.4\platforms\zc702\sw\linux\boot\linux.bif

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 `sdscc` or `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.
- **Boot Directory**: A directory containing any files referenced in the BIF file such as the FSBL, U-Boot, ARM Trusted Firmware. The Boot directory should have all the software components referred by the BIF file. If any of the components referred to in the BIF file are not present in Boot directory the platform generation step will return an error about the missing component.
- **Bif File**: The Boot Information File (BIF) contains information such as which ELF file to load on which processor (ie. FSBL runs on core 0), how the bitstream should be loaded, and any other sub-programs which must run to configure the processor (such as the ARM Trust Zone bl31.elf, or u-boot.elf, etc.). Refer to Chapter 4: Software Platform Data Creation for more information.

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

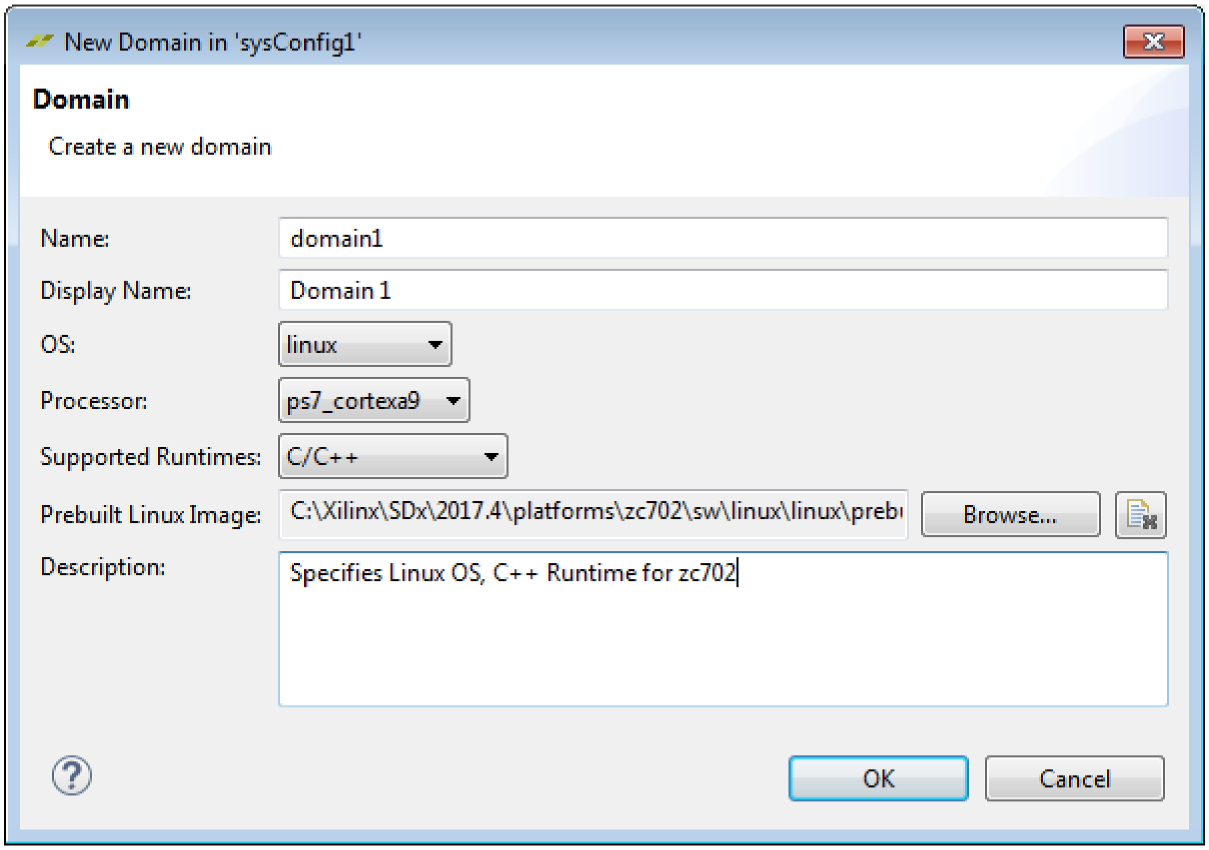
The Processor Domain will define the OS operating on one or more processors on the device, and the run-time. The domain defines different settings for the OS, whether Linux, FreeRTOS, or Standalone.

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 will open the New Domain dialog box as shown below.

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

Figure 8: **Domain**



New Domain in 'sysConfig1'

Domain

Create a new domain

Name:	domain1
Display Name:	Domain 1
OS:	linux
Processor:	ps7_cortexa9
Supported Runtimes:	C/C++
Prebuilt Linux Image:	C:\Xilinx\SDx\2017.4\platforms\zc702\sw\linux\linux\prebı
Description:	Specifies Linux OS, C++ Runtime for zc702

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, '-'.*

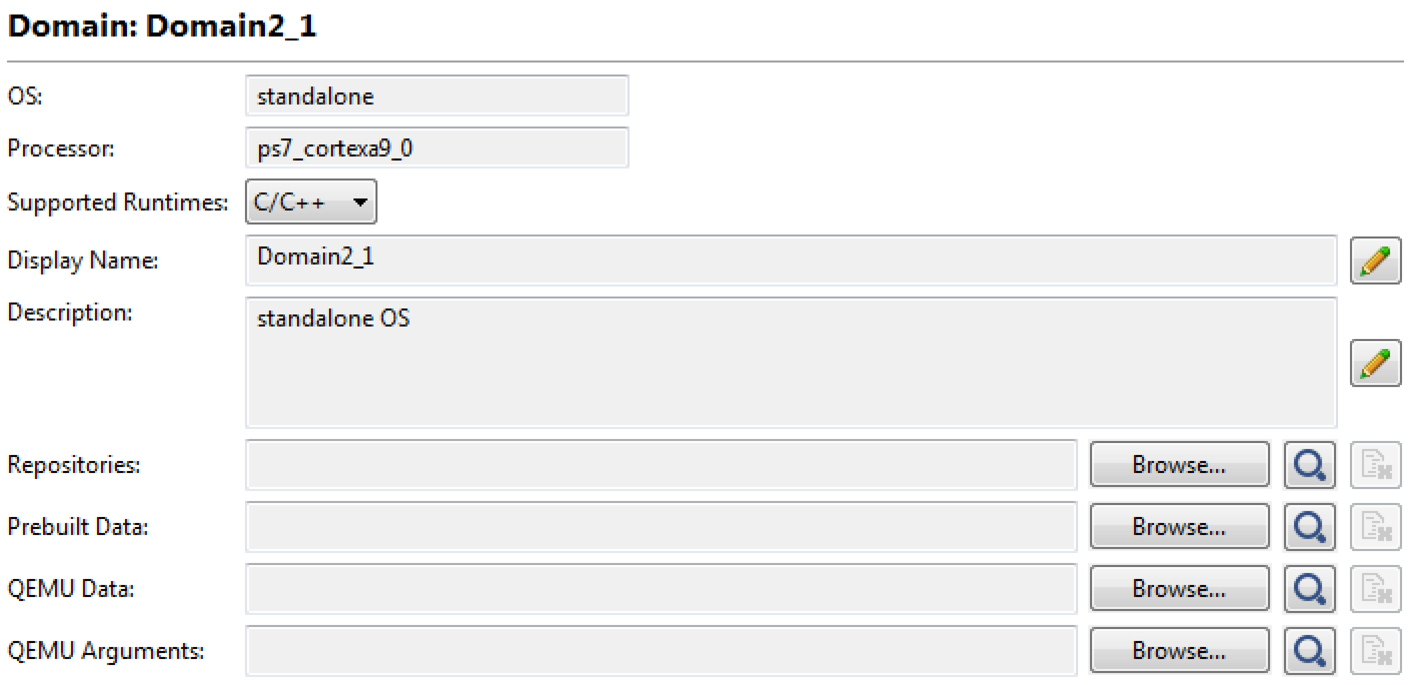
- **Display Name**: The name that will be displayed by the SDx IDE and in reports. This name can include spaces and special characters.
- **OS**: Specify the OS you are configuring as either Linux, FreeRTOS, or Standalone from the drop-down menu. For each OS there are various files that must be included 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, while **OpenCL** is only supported if the target OS is Linux.
- **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, please refer to Chapter 4: Software Platform Data Creation.
- **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.

Figure 9: **Domain Added to Platform**



Domain: Domain2_1

OS:	standalone
Processor:	ps7_cortexa9_0
Supported Runtimes:	C/C++
Display Name:	Domain2_1
Description:	standalone OS
Repositories:	
Prebuilt Data:	
QEMU Data:	
QEMU Arguments:	

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 Pre-built Hardware for more information on generating prebuilt data for custom platforms.
- **QEMU Data**: Specify QEMU data file to enable emulation for the platform. All the SW components required by the **QEMU Arguments** and **PMU QEMU Arguments** files should be present in the QEMU Data directory.
- **QEMU Arguments file**: This is required for enabling emulation for the platforms.

- **PMU QEMU Arguments file**: This file contains arguments for PMU QEMU to start. This is only displayed for Zynq UltraScale+ MPSoC based designs.

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.

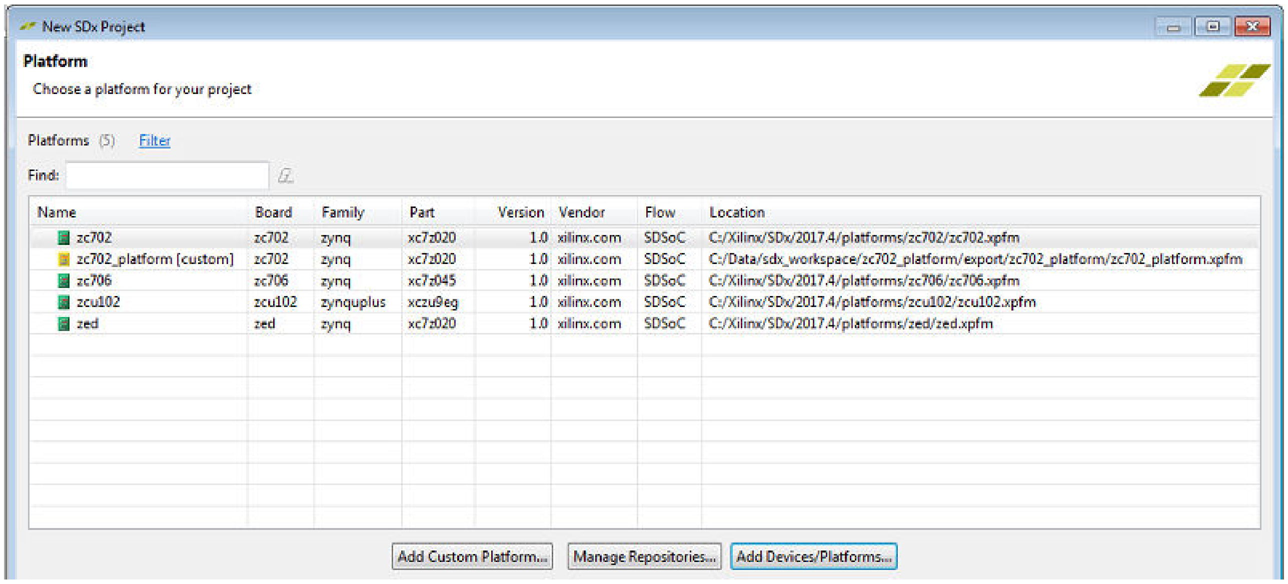
IMPORTANT!: *If you make any changes to any of the fields in the platform project, System Configuration, or domains, you must regenerate the platform to update the platform in the repository. You do not need to re-export the platform to the repository, but you will need to regenerate the output.*

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 will add the exported platform to the SDx custom repository for use in SDx Application and System projects as shown in the following figure. 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.

Figure 10: **New SDx Project - Select Platform**



Name	Board	Family	Part	Version	Vendor	Flow	Location
zc702	zc702	zynq	xc7z020	1.0	xilinx.com	SDSoC	C:/Xilinx/SDx/2017.4/platforms/zc702/zc702.xpfm
zc702_platform [custom]	zc702	zynq	xc7z020	1.0	xilinx.com	SDSoC	C:/Data/sdx_workspace/zc702_platform/export/zc702_platform/zc702_platform.xpfm
zc706	zc706	zynq	xc7z045	1.0	xilinx.com	SDSoC	C:/Xilinx/SDx/2017.4/platforms/zc706/zc706.xpfm
zcu102	zcu102	zynquplus	xczu9eg	1.0	xilinx.com	SDSoC	C:/Xilinx/SDx/2017.4/platforms/zcu102/zcu102.xpfm
zed	zed	zynq	xc7z020	1.0	xilinx.com	SDSoC	C:/Xilinx/SDx/2017.4/platforms/zed/zed.xpfm

TIP: *For an example of using the command-line flow to build an SDSoC platform, refer to* Making the SDSoC Platform from the Command Line.

Metadata Files

The SDSoC platform includes the following metadata files automatically created in the SDSoC platform project that describe the hardware and software interfaces.

- Top-Level Platform XML file (.xpfm): The top-level platform file written by the SDx platform project as `<platform>/<platform>.xpfm`, with references to the hardware and software files and the folders that contain them.

- Software Metadata file (.spfm): The software platform file is written by the SDx platform project as `<platform>/sw/<platform>.spfm`. The `.spfm` file describes the software environments, or system configurations available for use by the platform. Each configuration has an operating system (OS) associated with it, and the user selects the system configuration when creating an application project on the hardware platform.

Testing and Using 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.xpfm` file, not the `xpfm` file itself.*

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>
```

Refer to *SDSoC Environment User Guide* (UG1027) for more information.

Chapter 3

SDSoC Hardware Platform Creation

The Hardware Platform captures the logical and physical interfaces to the hardware functions coming from the SDx environment. The processor, memory, and all external board interfaces are configured using a combination of Xilinx 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 DSA file for use in an SDx 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 needed to support the platform into a Device Support Archive (DSA) file. 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.

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.

- The Vivado Design Suite project name must match the hardware platform name.

TIP: *If the Vivado 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 Vivado 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 IPs.
- 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 IPs.
- 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.

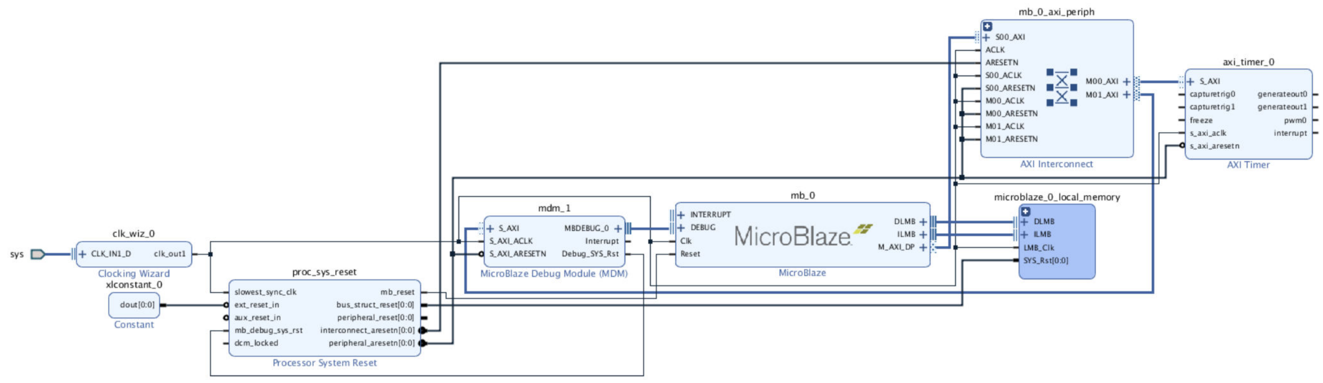
TIP: *Accelerator functions generated by the SDSoC compilers might run on a different clock that is provided by the platform.*

- 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.

MicroBlaze Hardware Requirements

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. 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.

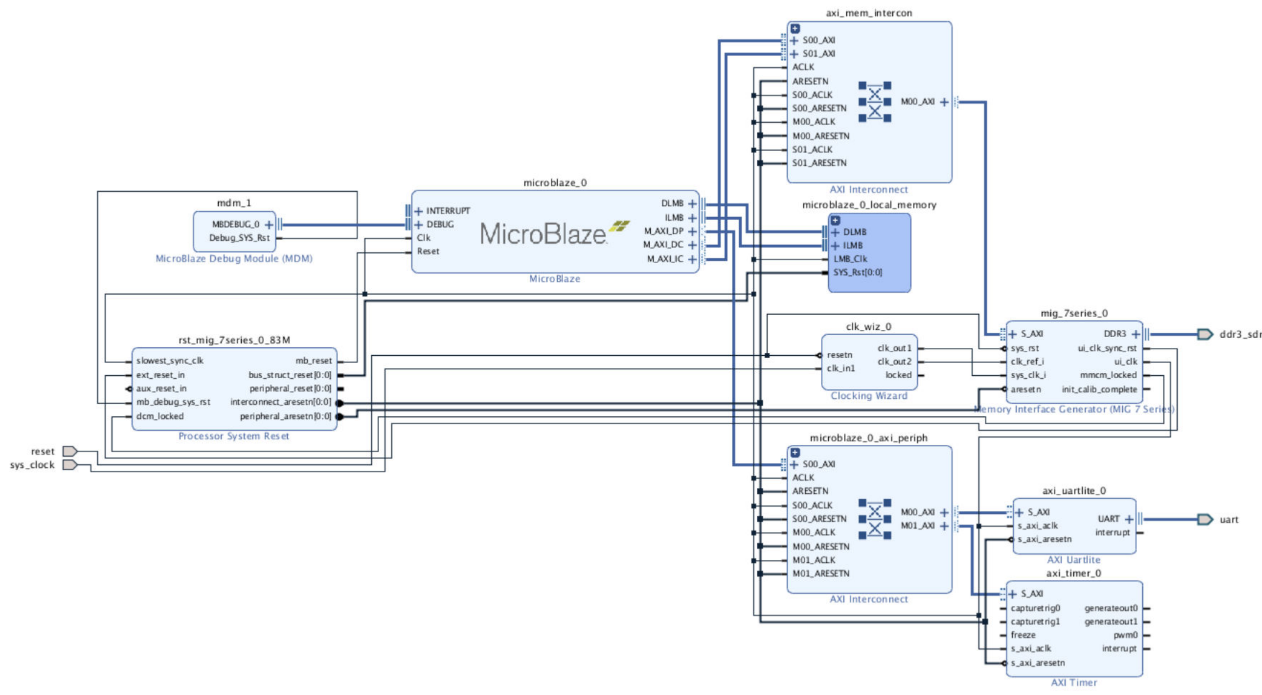
Figure 11: **Minimal MicroBlaze System**



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 12: **MicroBlaze System with MIG**



In the system above, the MicroBlaze is configured with an 8KB 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.

Creating the Vivado Platform 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.

***Note*:** The Vivado Design Suite project must have the same name as the target platform.

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 SoC, 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 Vivado 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.*

Creating a Vivado Project and DSA

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

1. Launch the Vivado IDE.
2. From the Vivado Design Suite use the **Create New Project** command to create the platform project called <*my_platform*>.

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

3. Select the Xilinx device or a supported board, such as the ZC702 or ZCU102 board, to use for your SDSoC platform. 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 top-level Vivado project opens in the Vivado IDE, use the **Create Block Design** command to create a block design also named after the platform, <*my_platform*>.
5. With the block design open in the IP integrator feature, instantiate the embedded processor IP from the IP catalog, as well as other Xilinx IP or custom IP needed to complete the design.

   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. Define the Platform Interfaces by setting the PFM properties to configure the interface ports.
7. Validate the block design to ensure everything is correct, and save the design.
8. Optionally enable IP caching to reduce synthesis and compilation times.
9. **Generate Output Products** of the IP in the block design.
10. Use the **Create HDL Wrapper** command to create the top-level RTL design.
11. **Export Hardware** to SDK to provide the boot loaders and target operating system required for the software elements of the platform. Refer to Chapter 4: Software Platform Data Creation for more information on defining the software platform libraries.
12. If you are using programmable logic device I/Os, assign I/O port constraints.
13. Optionally, simulate and implement the design to validate functionality and performance
14. Archive the project for use as a backup Project.
15. Write and validate the DSA using `write_dsa` and `validate_dsa`.

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 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. I large assortment of AXI4 compliant Xilinx IP is available for most system design needs.

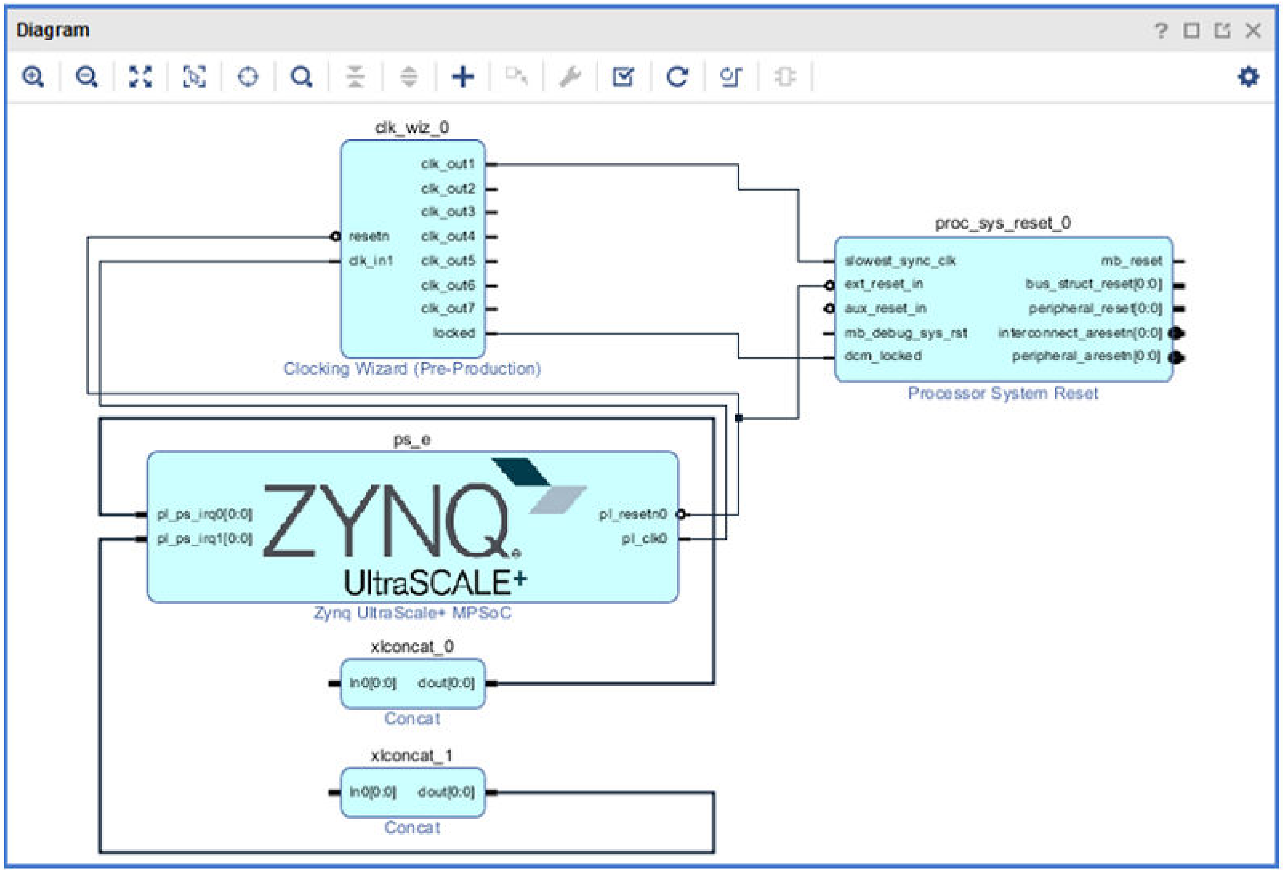
The logic design can be captured using IP integrator or with RTL sources. A top-level wrapper is used to instantiate the IP integrator design as well as any top-level RTL modules. RTL modules can also be added directly to the IP integrator block design (BD).

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

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

The following figure shows an example Zynq-based hardware platform design.

Figure 13: **Example zcu102 Block Design in IP Integrator**



Defining Platform (PFM) Properties

After you complete the hardware platform design project in the Vivado Design Suite, you must add platform properties (PFM) to define the platform name and configure platform interfaces such as clocks, interrupts, and bus interfaces. These properties are set once and stored in the project.

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 zc702 [get_files zc702.bd]
```

This results in the PFM_NAME: `vendor:library:zc702:1.0`.

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 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 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.

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 AXI-4 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's 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

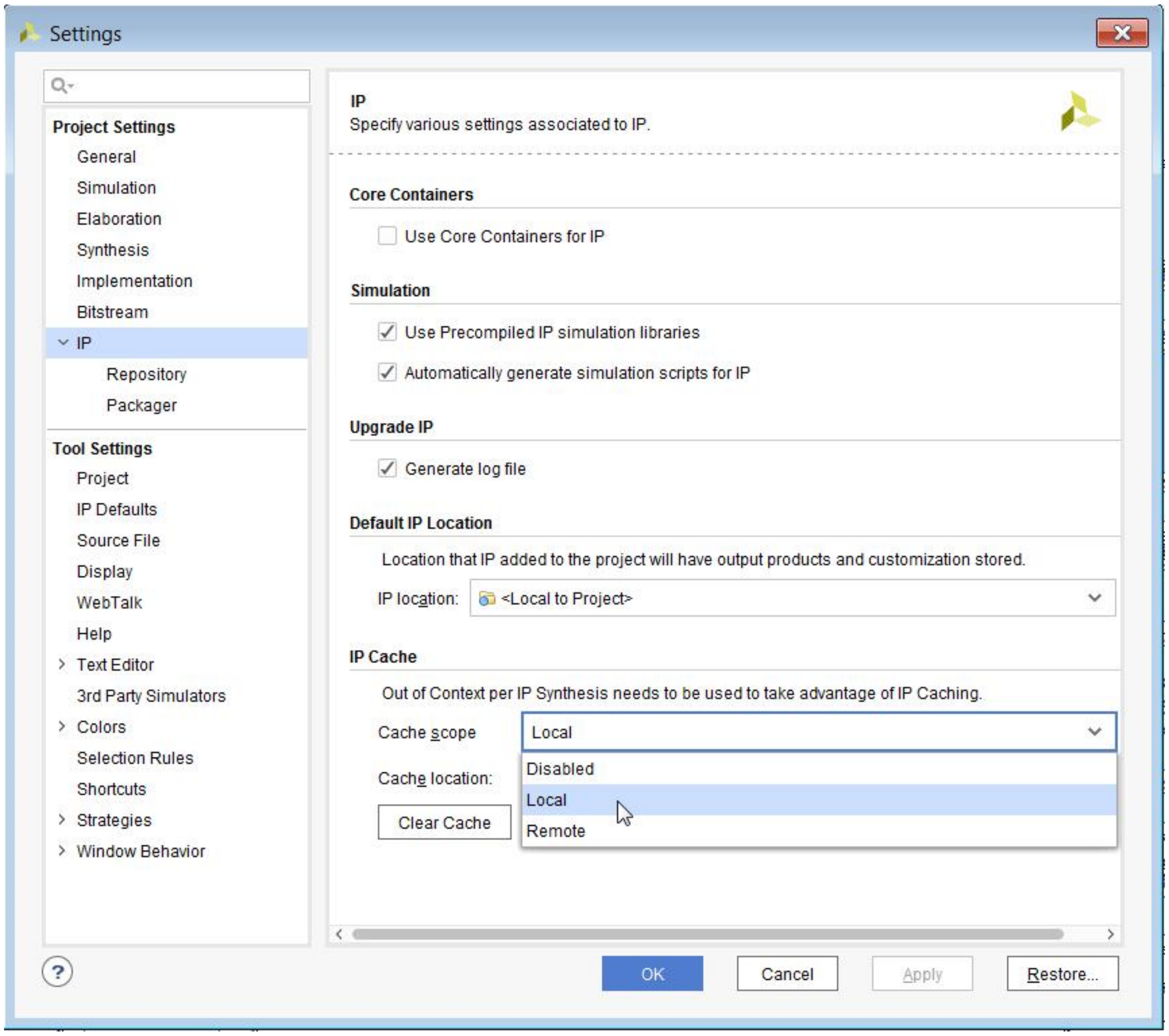
The hardware platform design should be implemented and validated to ensure it works as expected in the Xilinx® SDAccel™ flow. The first step in that validation process should be to ensure the hardware platform design itself is performing as expected. This can be done using test kernel logic to populate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

Using the IP Cache

Significant synthesis runtime savings can be achieved by taking advantage of the IP caching capabilities in Vivado. 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 14: **Vivado Settings - IP Cache**



IP
Specify various settings associated to IP.

Core Containers
☐ Use Core Containers for IP

Simulation
✓ Use Precompiled IP simulation libraries
✓ Automatically generate simulation scripts for IP

Upgrade IP
✓ 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
Cache location:

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.

The AXI Verification IP, described in Configuring Simulation IP, should help to drastically reduce logic simulation runtime and effectiveness for PCIe based platforms.

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® SDAccel™ environment. Use the following command to validate a DSA file:

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

Chapter 4

Software Platform Data Creation

Introduction

In general, the software components are created using the Xilinx SDK for standalone and FreeRTOS applications whereas the PetaLinux tools are used to create Linux images and applications. An HDF file, generated by the Vivado tools **Export Hardware** command, is used by the software creation tools as an input that describes the hardware design.

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`.

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.

Pre-built Hardware

A platform can optionally include pre-built configurations to be used directly when the user does not specify any hardware functions in an application. In this case, the platform user does not need to wait for a hardware compile of the platform itself to create a bitstream and other required files.

The pre-built 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. When defining the SDSoC platform project, you can specify the path to a folder containing pre-built hardware information to be included in the platform. As shown in Directory Structure for a Typical SDSoC Platform, the path to pre-built hardware data is:

```
path_to_platform/sw/os/os/prebuilt
```

The path is relative to the software platform folder and the Operating System for the processor. For example, for Linux the folder name `prebuilt` indicates the prebuilt hardware bitstream and generated files are found in `<path_to_platform>/sw/Linux/Linux/prebuilt`. The `prebuilt` folder for the zc702 platform contains `bitstream.bit`, `zc702.hdf`, `partitions.xml`, `apsys_0.xml`, `portinfo.c` and `portinfo.h` files.

Pre-built hardware files are automatically used by the SDx environment when an application has no hardware functions using the usual flag:

```
-sds-pf zc702
```

To force a full Vivado tools bitstream and SD card image compile, instead of using the pre-built files, use the following `sdscc` option:

```
-rebuild-hardware
```

Files used to populate the `platforms/<platform>/sw/os/os/prebuilt` folder are found in the `_sds` folder after creating the application ELF and bitstream.

- `system.bit`, and `system.hdf`
  - Files found in `_sds/p0/vpl`

IMPORTANT!: *The `system.hdf` file should be renamed to match the name of the platform (`<platform>.hdf`) when copying it into the `<platform>/sw/os/os/prebuilt` folder or the tool will return an error when generating the platform.*

- `partitions.xml`, `apsys_0.xml`
    - Files found in `_sds/.llvm`
- `portinfo.c`, `portinfo.h`
    - Files found in `_sds/swstubs`

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 All Programmable (AP) SoC designs.

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: SDSoC Hardware Platform Creation *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 MPSoC: Device Drivers→ Generic Driver Options → Size in Mega Bytes(1024)
   - for Zynq-7000: 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 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 All Programmable 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 All Programmable (AP) 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 Defining the Software Platform 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 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/zc702_axis_io/sw/standalone/boot/fsbl.elf
```

Executable

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 All Programmable (AP) 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 AP 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 v9.0.1 is supported and in SDK this corresponds to the most recent `freertos901_xilinx` BSP library. SDx can also support `freertos823_xilinx`.

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 `freertos901_xilinx`. If the OS name is specified explicitly as "freertos901_xilinx" or "freertos823_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.

Chapter 5

Platform Sample Applications

Overview

A platform can optionally include sample applications to demonstrate the usage of the platform. The sample applications must be defined in a file named `template.xml` found in the `samples` directory of a platform. Here is an example for the `zc702_axis_io` sample platform found in the `<SDx_install>/SDx/2017.4/samples/sdspfm/zc702_axis_io/src/samples` folder.

```
<?xml version="1.0" encoding="UTF-8"?>
<manifest:Manifest xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
                   xmlns:manifest="https://www.xilinx.com/manifest">
    <template location="aximm" name="Unpacketized AXI4-Stream to DDR"
              description="Shows how to copy unpacketized AXI-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"/>
                </or>
            </and>
        </supports>
        <accelerator name="PullPacket" location="main.cpp"/>
    </template>
</manifest:Manifest>
```

The first line defines the template file format as XML, and is mandatory:

```
<?xml version="1.0" encoding="UTF-8"?>
```

The `<manifest:Manifest>` XML element is required as a container for all application templates defined in the template file:

```
<manifest:Manifest xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
                   xmlns:manifest="https://www.xilinx.com/manifest">
   <!-- ONE OR MORE TEMPLATE XML ELEMENTS GO HERE -->
</manifest:Manifest>
```

<template> element

Each application template is defined inside a `<template>` element, and there can be multiple `<template>` tags defined within the `<manifest>` element. The `<template>` element can have multiple attributes as shown in the following table:

Table 1: **Attributes of the Template Element**

Attribute	Description
location	Relative path to the template application
name	Application name displayed in the SDSoC environment
description	Application description displayed in the SDSoC environment

Example:

```
<template location="aximm" name="Unpacketized AXI4-Stream to DDR"
              description="Shows how to copy unpacketized AXI-Stream data
directly to DDR.">
```

The `<template>` element can also contain multiple nested elements that define different aspects of the application.

Table 2: **Template Sub-Elements**

Element	Description
<supports>	Defines an Operating System match for the current template.
<includepaths>	Paths relative to the application to be added to the compiler as `-I` flags.
<librarypaths>	Paths relative to the application to be added to the linker as `-L` flags.
<libraries>	Platform libraries to be linked against using linker `-l` flags.
<exclude>	Directories or files to exclude from copying into the SDSoC project.

Table 2: **Template Sub-Elements** *(cont'd)*

Element	Description
<system>	Application project settings for the system, for example the data motion clock.
<accelerator>	Application project settings for specifying a function target for hardware.
<compiler>	Application project settings defining compiler options.
<linker>	Application project settings defining linker options.

<supports> element

The `<supports>` element defines an Operating System match for the selected SDx platform. The `<os>` elements must be enclosed in `<and>` and `<or>` elements to define a boolean function.

The following example defines an application that can be selected when either of Linux, Standalone or FreeRTOS are selected as an Operating System:

```
<supports>
  <and>
    <or>
      <os name="Linux"/>
      <os name="Standalone"/>
      <os name="FreeRTOS"/>
    </or>
  </and>
</supports>
```

<includepaths> element

The `<includepaths>` element defines the set of paths relative to the application that are to be passed to the compiler using -I flags. Each `<path>` element has a `location` attribute.

The following example results in SDx adding the flags `-I"../src/myinclude" -I"../src/dir/include"` to the compiler:

```
<includepaths>
  <path location="myinclude"/>
  <path location="dir/include"/>
</includepaths>
```

<librarypaths> element

The `<librarypaths>` element defines the set of paths relative to the application that are to be passed to the linker using -L flags. Each `<path>` element has a `location` attribute.

The following example results in the SDSoC tool adding the flags `-L"../src/mylibrary" -L"../src/dir/lib"` to the linker:

```
<librarypaths>
  <path location="mylibrary"/>
  <path location="dir/lib"/>
</librarypaths>
```

<libraries> element

The `<libraries>` element defines the set of libraries that are to be passed to the linker -l flags. Each `<lib>` element has a `name` attribute.

The following example results in the SDSoC tool adding the flags `-lmylib2 -lmylib2` to the linker:

```
<libraries>
  <lib name="mylib1"/>
  <lib name="mylib2"/>
</libraries>
```

<exclude> element

The `<exclude>` element defines a set of directories and files to be excluded from being copied when SDx creates the new project.

The following example will result in the SDSoC tool 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`. This allows you to have files or directories in the application directory that are not needed to build the application.

```
<exclude>
  <directory name="MyDir"/>
  <directory name="MyOtherDir"/>
  <file name="MyFile.txt"/>
  <file name="MyOtherFile.txt:/>
</exclude>
```

<system> element

The optional `<system>` element defines application project settings for the system when creating a new project. The `dmclkid` attribute defines the data motion clock ID. If the `<system>` element 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"/>
```

<accelerator> element

The optional `<accelerator>` element defines application project settings for a function targeted for hardware when creating a new project. The `name` attribute defines the name of the function and the `location` attribute defines the path to the source file containing the function (the path is relative to the folder in the platform containing the application source files). The `name` and `location` are required attributes of the `<accelerator>` element. The optional attribute `clkid` specifies the accelerator clock to use instead of the default. The optional sub-element `<hlsfiles>` specifies the `name` of a source file (path relative to the folder in the platform containing application source files) containing code called by the accelerator and the accelerator is found in a different file. The SDx environment normally infers `<hlsfiles>` information for an application and this sub-element does not need to be specified unless the default behavior needs to be overridden.

The following example will result in the SDSoC tool specifying two functions to move to hardware `func1` and `func2` when creating the new project.

```
<accelerator name="func1" location="func1.cpp"/>
<accelerator name="func2" location="func2.cpp" clkid="2">
  <hlsfiles name="func2_helper_a.cpp/>
  <hlsfiles name="func2_helper_b.cpp/>
</accelerator>
```

<compiler> element

The optional `<compiler>` element defines application project settings for the compiler when creating a new project. The `inferredOptions` attribute defines compiler options required to build the application and appears in the SDx environment C/C++ Build Settings dialog as compiler Inferred Options under Software Platform.

The following example will result in the SDSoC tool adding the compiler option `-D MYAPPMACRO` when creating the new project.

```
<compiler inferredOptions="-D MYAPPMACRO"/>
```

<linker> element

The optional `<linker>` element defines application project settings for the linker when creating a new project. The `inferredOptions` attribute defines linker options required to build the application and appears in the SDx environment C/C++ Build Settings dialog as linker Miscellaneous options.

The following example will result in SDx adding the linker option `-poll-mode 1` when creating the new project.

```
<linker inferredOptions="-poll-mode 1"/>
```

Appendix A

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 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 C: 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 B: SDSoC Platform Migration.

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: SDSoC Hardware Platform Creation 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 Software Platform Data Creation 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 Platform 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

SDSoC Platform Migration

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* Creating the Vivado Platform 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.*

Migrating Platforms to 2017.4

The platform structure and process for creating platforms has changed in 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 regenetrate 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.

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

Table 3: **Mandatory Properties for 2017.4**

Purpose	Pre 2017.4	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 C

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.

Example: Direct I/O in an SDSoC Platform

An SDSoC platform can include input and output subsystems, e.g., 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:

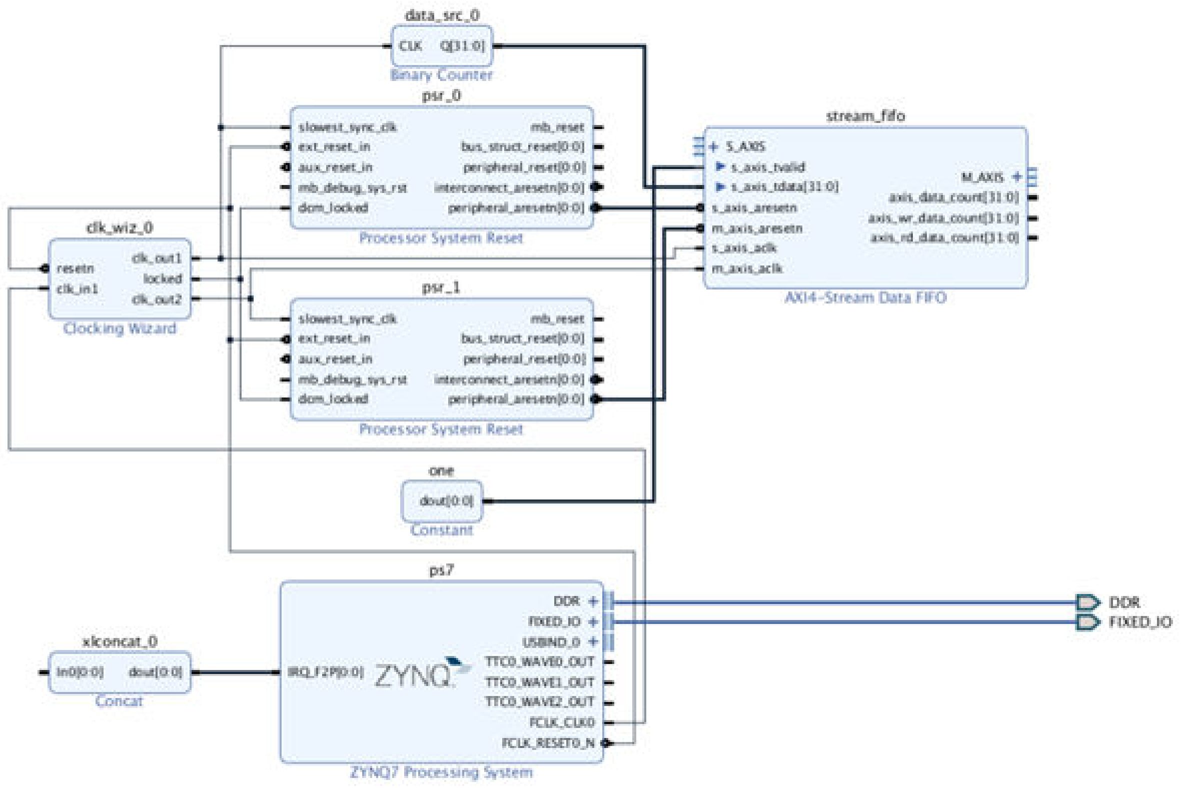
```
vivado -mode batch -source zc702_axis_io_dsa.tcl
```

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

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

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

Figure 15: **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. Hence, 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: SDSoC Hardware Platform Creation, 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+ 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 SDSoC 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 SDx 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. The SDx IDE looks for a file called `samples/template.xml` for information on where the sample application source files reside within the platform. The template file for the `zc702_axis_io` platform lists several test applications, each of which is of specific interest.

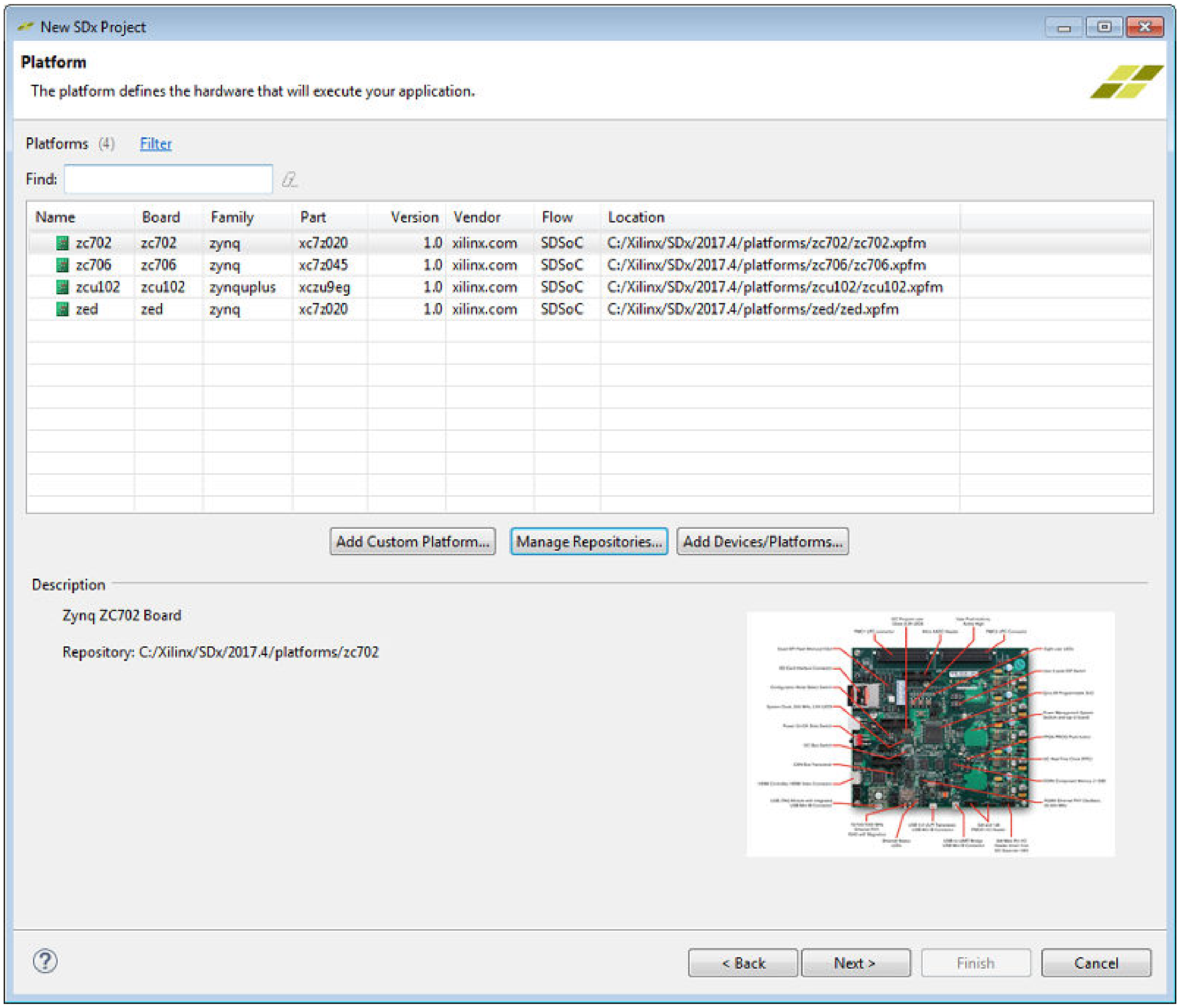
```
<template location="aximm" name="Unpacketized AXI4-Stream to DDR"
              description="Shows how to copy unpacketized AXI4-Stream data
directly to DDR.">
        <supports>
            <and>
                <or>
                    <os name="Linux"/>
                    <os name="Standalone"/>
                </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 16: **Add Custom Platform**



New SDx Project

Platform

The platform defines the hardware that will execute your application.

Platforms (4) Filter

Find:

Name	Board	Family	Part	Version	Vendor	Flow	Location
zc702	zc702	zynq	xc7z020	1.0	xilinx.com	SDSoC	C:/Xilinx/SDx/2017.4/platforms/zc702/zc702.xpfm
zc706	zc706	zynq	xc7z045	1.0	xilinx.com	SDSoC	C:/Xilinx/SDx/2017.4/platforms/zc706/zc706.xpfm
zcu102	zcu102	zynquplus	xczu9eg	1.0	xilinx.com	SDSoC	C:/Xilinx/SDx/2017.4/platforms/zcu102/zcu102.xpfm
zed	zed	zynq	xc7z020	1.0	xilinx.com	SDSoC	C:/Xilinx/SDx/2017.4/platforms/zed/zed.xpfm

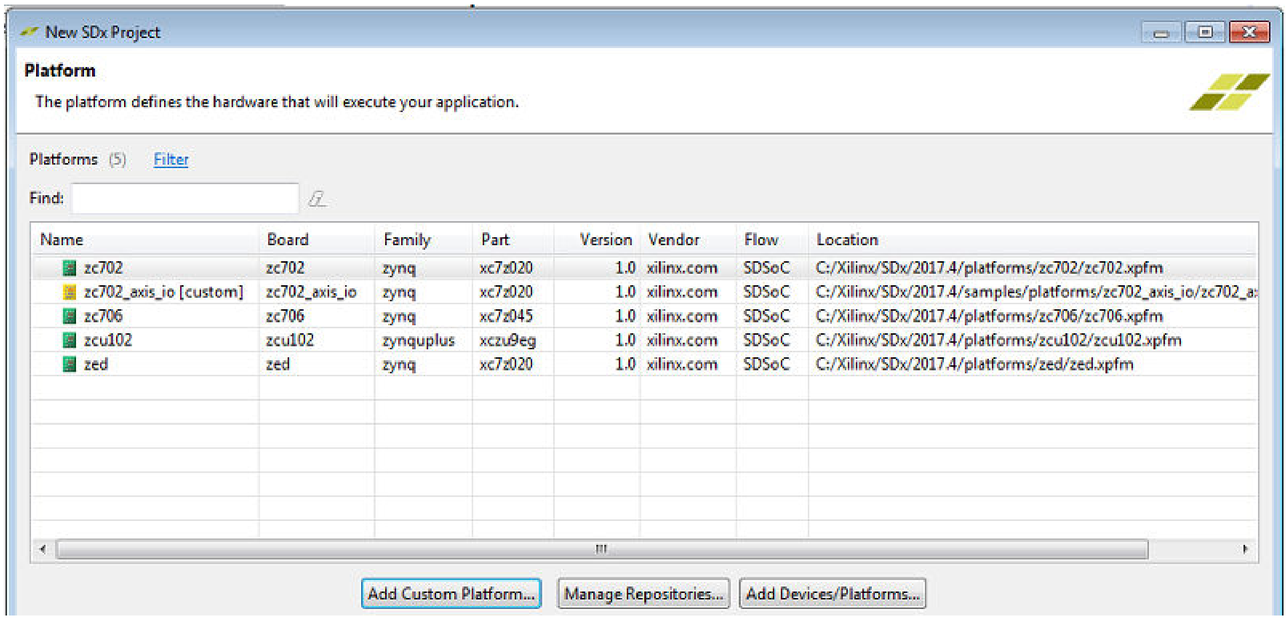
Description

Zynq ZC702 Board

Repository: C:/Xilinx/SDx/2017.4/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 17: **Choose Hardware Platform**


New SDx Project

Platform

The platform defines the hardware that will execute your application.

Platforms (5) Filter

Find:

Name	Board	Family	Part	Version	Vendor	Flow	Location
zc702	zc702	zynq	xc7z020	1.0	xilinx.com	SDSoC	C:/Xilinx/SDx/2017.4/platforms/zc702/zc702.xpfm
zc702_axis_io [custom]	zc702_axis_io	zynq	xc7z020	1.0	xilinx.com	SDSoC	C:/Xilinx/SDx/2017.4/samples/platforms/zc702_axis_io/zc702_a:
zc706	zc706	zynq	xc7z045	1.0	xilinx.com	SDSoC	C:/Xilinx/SDx/2017.4/platforms/zc706/zc706.xpfm
zcu102	zcu102	zynquplus	xczu9eg	1.0	xilinx.com	SDSoC	C:/Xilinx/SDx/2017.4/platforms/zcu102/zcu102.xpfm
zed	zed	zynq	xc7z020	1.0	xilinx.com	SDSoC	C:/Xilinx/SDx/2017.4/platforms/zed/zed.xpfm

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:

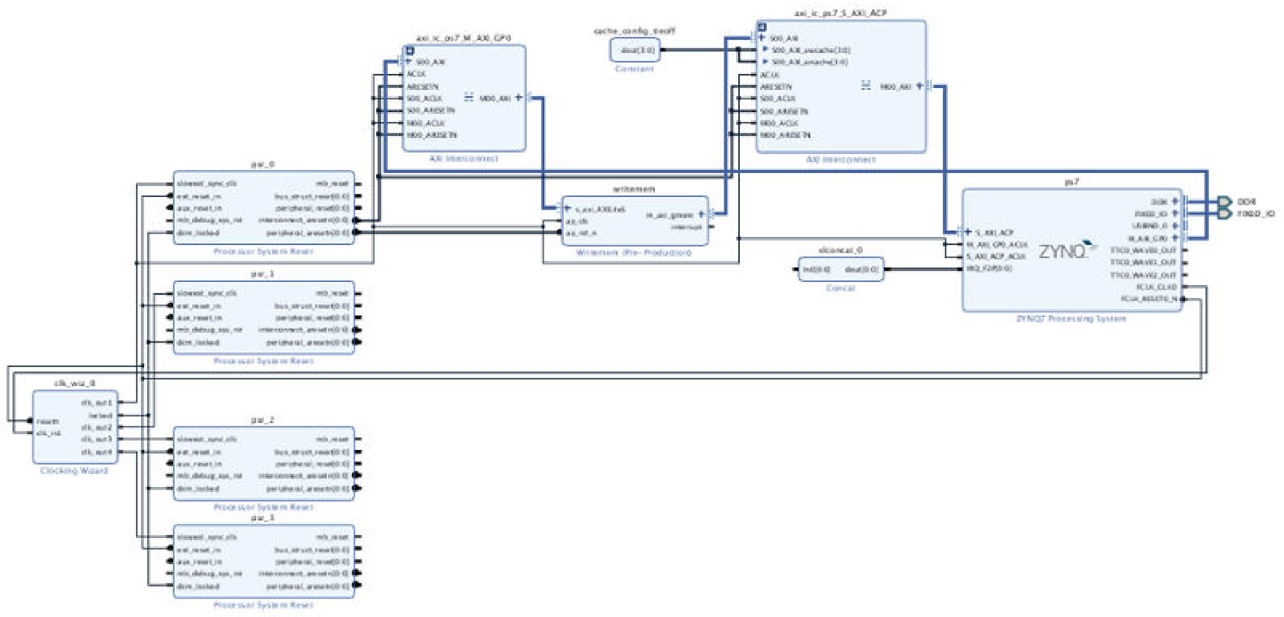
   ```
   vivado -mode batch -source zc702_acp_dsa.tcl
   ```

   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 18: **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+ 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 D

Additional Resources and Legal Notices

Xilinx Resources

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

Solution Centers

See the Xilinx Solution Centers for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips

References

These documents provide supplemental material useful with this webhelp:

1. *SDx Environments Release Notes, Installation, and Licensing Guide* (UG1238)
2. *SDSoC Environment User Guide* (UG1027)
3. *SDSoC Environment Optimization Guide* (UG1235)
4. *SDSoC Environment Tutorial: Introduction* (UG1028)
5. *SDSoC Environment Platform Development Guide* (UG1146)
6. SDSoC Development Environment web page
7. *UltraFast Embedded Design Methodology Guide* (UG1046)
8. *Zynq-7000 All Programmable SoC Software Developers Guide* (UG821)
9. *Zynq UltraScale+ MPSoC Software Developer Guide* (UG1137)
10. *ZC702 Evaluation Board for the Zynq-7000 XC7Z020 All Programmable SoC User Guide* (UG850)
11. *ZCU102 Evaluation Board User Guide* (UG1182)
12. *PetaLinux Tools Documentation: Workflow Tutorial* (UG1156)
13. *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)

14. *Vivado Design Suite User Guide: Creating and Packaging Custom IP* (UG1118)

15. Vivado® Design Suite Documentation

Please Read: Important Legal Notices

AUTOMOTIVE APPLICATIONS DISCLAIMER

Copyright