SDSoC Environment Platform Development Guide

UG1146 (v2017.1) June 20, 2017

XILINX
ALL PROGRAMMABLE™

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
06/20/2017	2017.1	• Significant edits to 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 SDSoC Platform Migration. • Added Tutorial: Using the SDSoC Platform Utility.

Table of Contents

Additional Resources and Legal Notices

XILINX
ALL PROGRAMMABLE™

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. This document describes how to create a custom SDSoC platform starting from a hardware system built using the Vivado® Design Suite, and a software run-time environment, including operating system kernel, boot loaders, file system, and libraries.

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 environment IDE targets a specific hardware platform, and you employ the tools within the SDx environment IDE 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.

IMPORTANT: *For additional information on using the SDSoC environment, see the SDSoC Environment User Guide (UG1027).*

XILINX
ALL PROGRAMMABLE™

Chapter 2

SDSoC Platforms

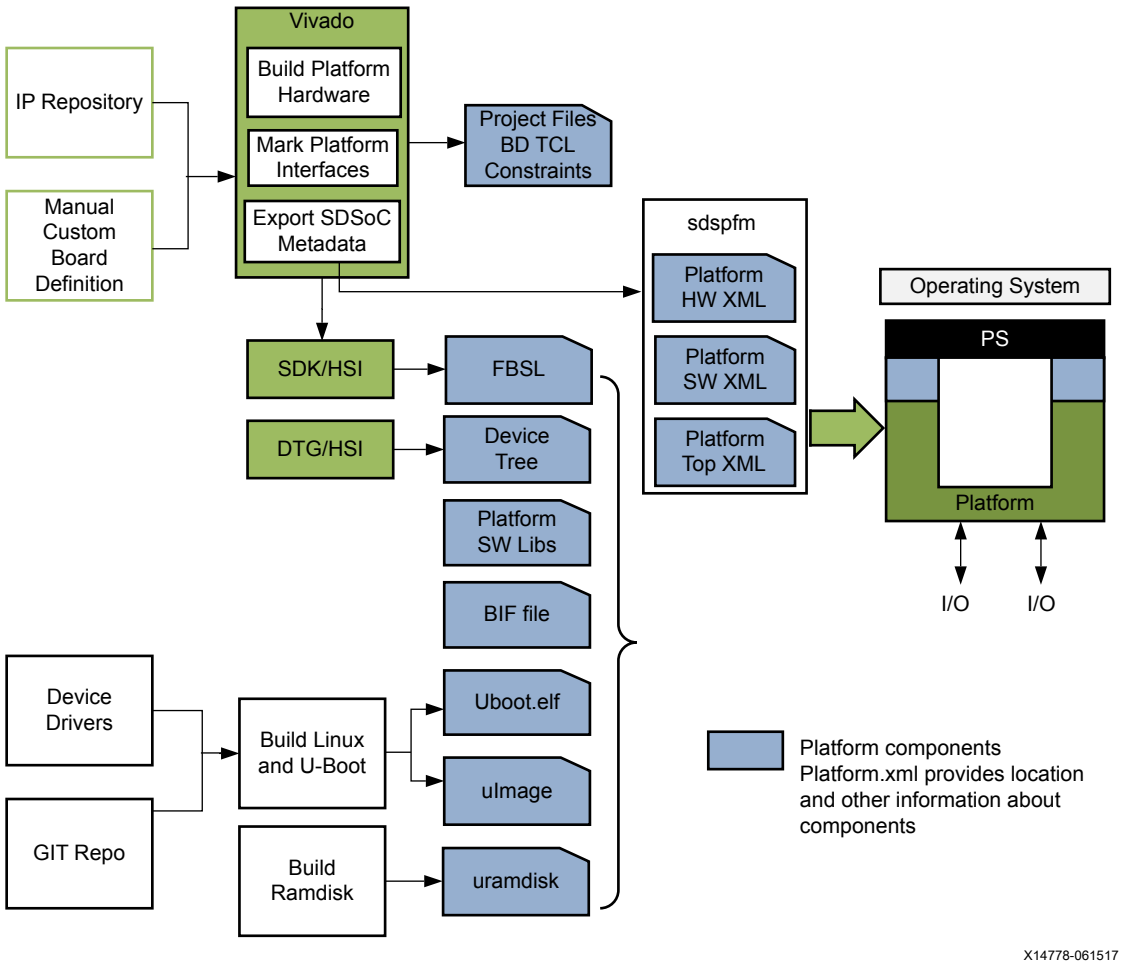
Introduction

An SDSoC platform consists of a Vivado® Design Suite hardware project, a target operating system, boot files, and optionally, software libraries that can be linked with user applications that target the platform. An SDSoC platform also includes supporting XML files that describe the hardware and software interfaces used by the SDSoC compilers to target the platform. These files are generated by the SDSoC Platform Utility.

A platform developer designs the platform hardware using the Vivado Design Suite and IP Integrator. After the hardware has been built and verified, the platform developer also provides a Tcl script to run within the Vivado tools to specify SDSoC platform hardware interfaces and generate the SDSoC platform hardware metadata file. See Hardware Platform Creation for more information on creating the hardware platform and required Tcl script.

The platform developer must also provide boot loaders and target operating system 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 at the command line or using the PetaLinux tool suite. You can use the PetaLinux tools, SDx IDE or the Xilinx SDK to build platform libraries. Refer to Software Platform Data Creation for more information on defining the software platform libraries.

Figure 1: Primary Components of an SDSoC Platform

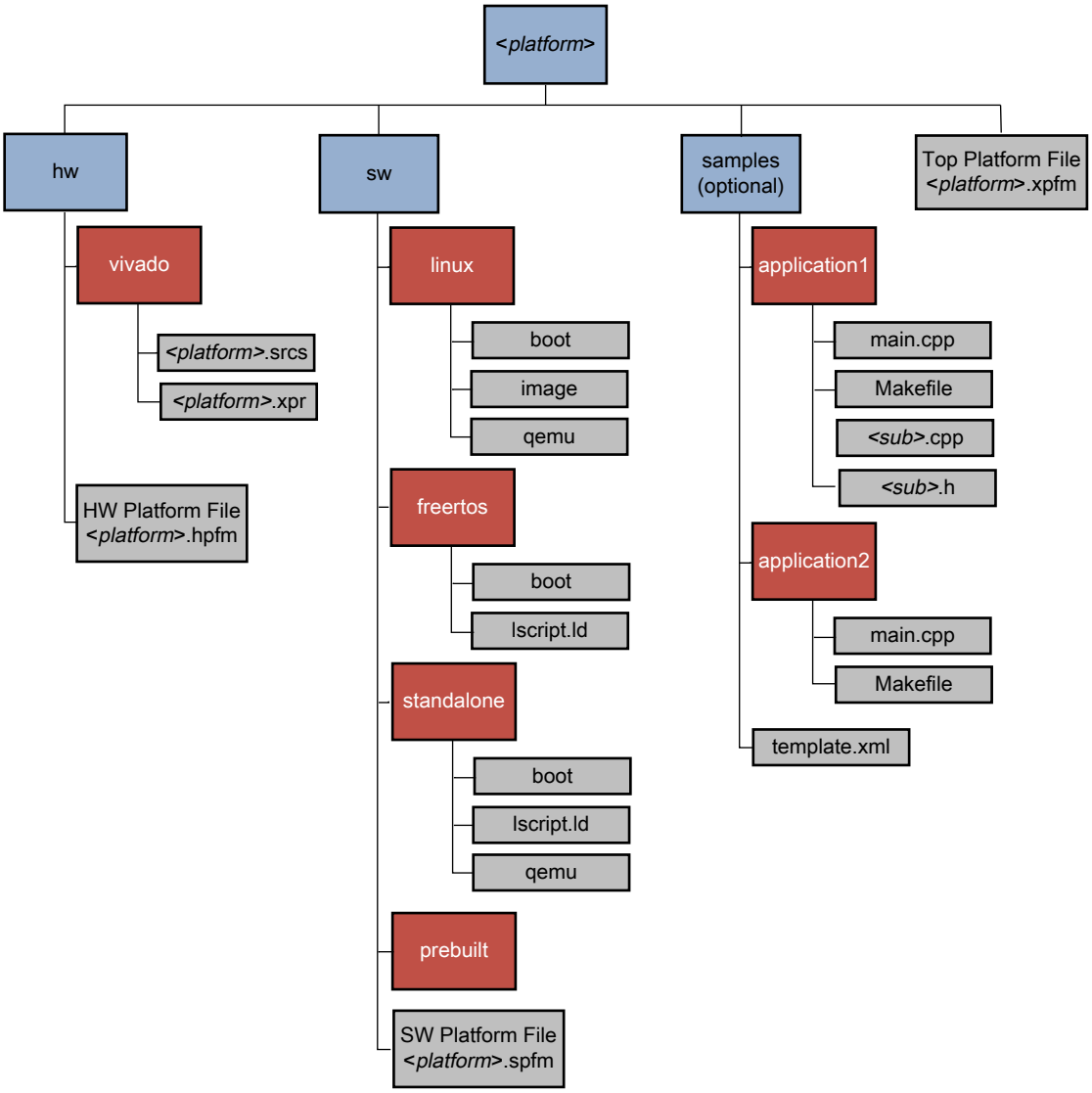


An SDSoC platform consists of the following elements, hierarchically structured as shown in Figure 2:

- Metadata files generated by SDSoC Platform Utility:
- Vivado Design Suite project
  - Design Source Files
  - Timing and Physical Constraint Files
  - IP blocks
- Software files
  - Library header files (optional)
  - Static libraries (optional)
  - 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` unified boot image)

- Pre-built hardware files (optional)
  - Bitstream
  - Exported hardware files for SDK
  - Pre-generated port information software files
  - Pre-generated hardware and software interface files
- 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 liveness and sanity 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 Environment IDE New Project Wizard to select and build any of the provided applications. For additional information on application template creation, see Platform Sample Applications.

The SDSoC environment provides the SDSoC Platform Utility, `sdspfm`, to assist with the creation of an SDSoC platform. You can use this utility after you have created the required Vivado hardware files and Tcl script, and built the components of your software platform.

Creating an SDSoC Platform

Introduction

TIP: *There are sample platform files including Vivado projects and boot files provided in the SDSoC software installation area at `<install>/samples/sdspfm`. Refer to Tutorial: Using the SDSoC Platform Utility for a demonstration of creating an SDSoC platform.*

The SDSoC Platform Utility simplifies the creation of SDSoC platforms. The platform utility exists in two parts: a simple form-like GUI for entering platform data and locating platform files, and a command-line utility that produces the platform as defined by the options.

SDSoC Platform Utility

The SDSoC Platform Utility simplifies the creation of SDSoC platforms, and can be invoked via the GUI as follows:

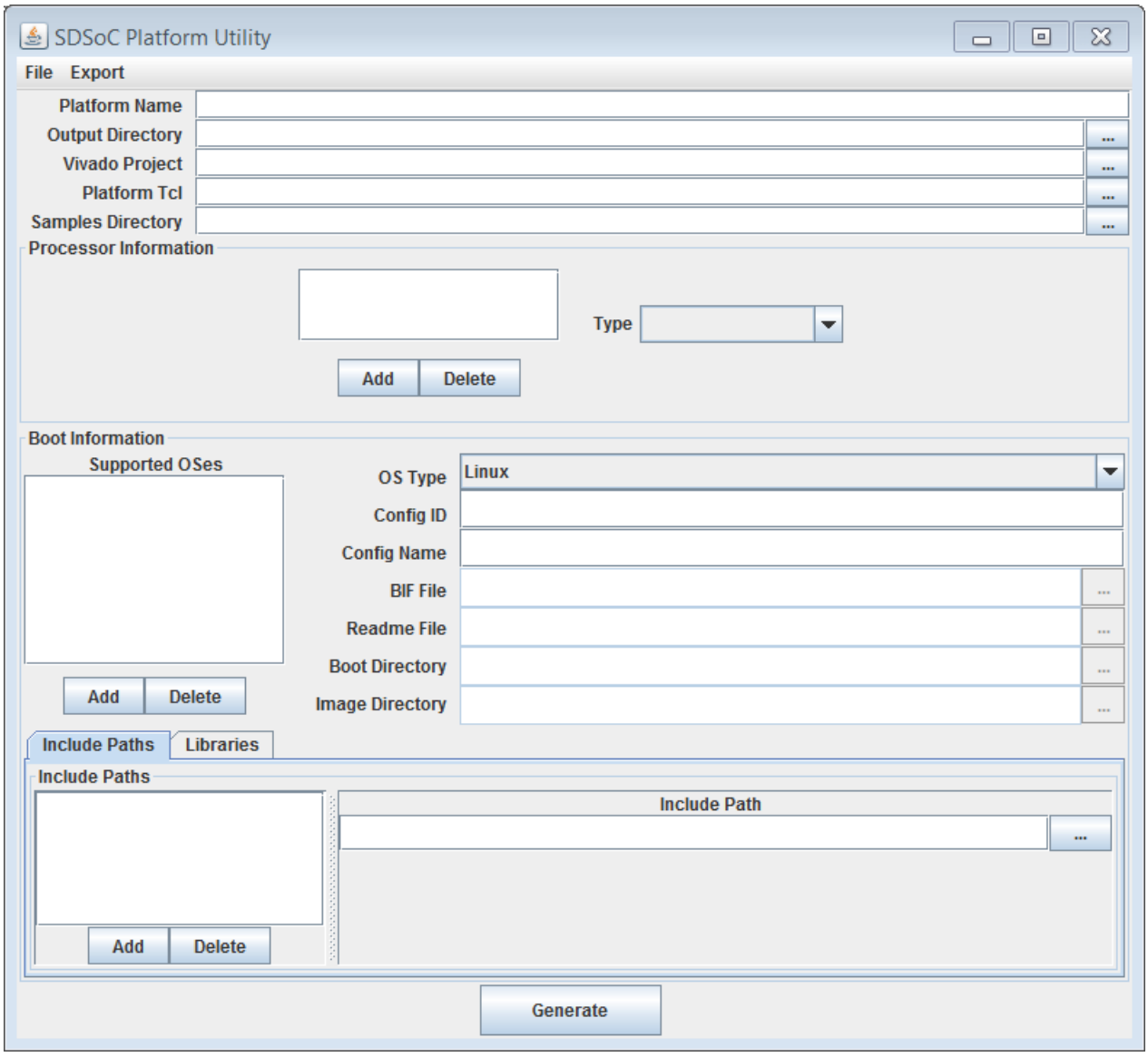
```
sdspfm -gui
```

IMPORTANT: *The SDSoC Platform Utility should be launched from the C:/ directory on the Windows OS, or an equivalent root directory, to avoid path name length limitations that can cause problems when generating the platform files on Windows.*

This GUI lets you save the platform configuration as an XML file that can be reloaded so that the platform configuration can be modified and reused. The GUI Platform Utility has three areas for platform data entry, as seen in the figure below:

- **Base Info**: The base information related to the platform name and the Vivado hardware project.
- **Processor Information**: Specifies the type of processor found in the platform as defined in the Vivado hardware design.
- **Boot Info**: As it relates to the Operating System data and the Compiler Settings for include paths and libraries to link against.

Figure 3: SDSoC Platform Utility



The **Base Info** includes:

- **Platform Name**: Must match the name of the Vivado project and the IP Integrator block diagram. (REQUIRED)
- **Output Directory**: Specifies a top-level platform repository, where the newly created platform directory will be written. (REQUIRED)
- **Vivado Project**: Specifies the Vivado project file (`.xpr`) containing the hardware platform as defined in Hardware Platform Creation. (REQUIRED)
- **Platform Tcl**: Script file that contains the platform clocks and ports exposed to SDSoC as defined in SDSoC Tcl Commands in Vivado (REQUIRED)
- **Samples Directory**: A directory containing any sample applications associated with the platform and the user-created `template.xml` file as defined in Platform Sample Applications. (OPTIONAL)

The **Processor Information** defines the available types of processors for the platform. The types of processors available is configured based on the device chosen in the Vivado project. For example, if a Zynq-7000 device such as the XC7Z020 part is chosen in the Vivado project then the GUI will only show an ARM Cortex-A9 processor type. However, if a Zynq MPSoC device such as the XCZU9 part is chosen then the GUI will show both ARM Cortex-A53 and ARM Cortex-R5 processor types. This allows the user to create a platform targeting just the A53 or R5 cores, or both as desired.

TIP: *You must select a Vivado project *.xpr file before the processors can be added to the configuration. Otherwise, the processor types will be empty.*

SDSoC only allows projects to be built targeting core #0 of any processor (Cortex-A9, Cortex-A53, or Cortex-R5). This means that you can have a standalone project on a9_0, but not a standalone project on a9_1. In another case, you can have Linux on both cores a9_0 and a9_1 but SDSoC doesn't need to know that Linux also uses a9_1. This limitation is mainly due to the fact that the FSBL must run on core #0 to initialize all cores in the processor, regardless of which core your application/OS runs on.

The **Boot Info** consists of the Operating System and the Compiler Settings. SDSoC currently supports 3 Operating Systems: Linux, FreeRTOS, and Standalone (ie. bare-metal). 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. Refer to Software Platform Data Creation for information on creating these files:

- **Config ID**: The ID of the OS/Boot configuration. The SDSoC Platform Utility allows platform developers to produce platforms with multiple build configurations. Each build configuration specifies: an OS, a processor type, and a set of boot files/settings. The Config ID identifies the build configuration, and must be unique for each build configuration in the platform. The Config ID must be made up of an alphanumeric string with underscores or dashes only (no spaces or special characters). When creating an SDSoC Makefile project, this ID is passed into the option *-sds-sys-config<config_ID>*. (All OSes)
- **Config Name**: The name of the OS/Boot configuration. The user-defined Config Name must be unique for each build configuration in the platform. The Config Name must be made up of an alphanumeric string with spaces, parentheses, underscores or dashes only. When creating an SDSoC project in the SDSoC Development Environment IDE this name is displayed in the project creation wizard.
- **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.). It is assumed that any files referenced in the BIF file will be in the directory specified by the **Boot Directory** field.
- **Readme File**: This file is required by SDSoC to inform users of the platform how to boot an application for this boot configuration on the platform. It is a plain text file that contains instructions to the user, for setting the boot mode switches on the board for example.
- **Image Directory**: A directory containing any OS image files that must be copied to the SD card. This directory should include the `image.ub` file for Linux, or a set of Linux files: `image.gz`, `ramdisk`, `devicetree.dtb`. This is required for Linux only.
- **Linker Script**: A file that 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. This is required for FreeRTOS and Standalone Only.
- **Boot Directory**: A directory containing any files referenced in the BIF file such as the FSBL, U-Boot, ARM Trusted Firmware, etc.

Compiler Settings: For each OS Configuration, compiler settings for the platform can be specified with include paths, and libraries to link against. These directories and libraries will be copied locally into the output platform at the time of platform generation. Users can add multiple include paths and libraries .

Platform Generation

After configuring the settings for your SDSoC platform, click the **Generate** button at the bottom of the SDSoC Platform Utility. This will start the process of copying and creating files, and generating metadata for your platform. If the output directory already exists, it will automatically be overwritten by the new platform files. You can regenerate the platform files as needed.

IMPORTANT: *If you encounter errors when the SDSoC Platform Utility is generating the platform, the cause may be related to the Vivado Design Suite project, and the IP Integrator block design. IP Locked errors can occur when the SDSoC Platform Utility invokes the Vivado tools to generate the hardware platform. This can be a result of improperly copying the Vivado project as described in Vivado Design Suite Project, or failing to upgrade the IP and output products as described in SDSoC Platform Migration.*

Utility Menus

The SDSoC Platform Utility GUI has two menus:

- **File**: provides options for saving the current configuration, opening a previously saved configuration, , and exiting the utility.
- **Export**: provides an option to export the platform configuration as a Makefile. This takes all the data entered into the GUI and outputs a file containing a bare-minimum Makefile with the options to call the command-line utility to generate a platform. This option may be helpful for advanced users. The exported Makefile cannot be read back into the SDSoC Platform Utility, so you are advised to also save the configuration using the File > Save command in addition to exporting it to a Makefile.

Utility Command Line

The `sdspfm -help` command displays the following information:

```
Usage: sdspfm -xpr <vivado_project.xpr> -pfmtcl <platform.tcl> [other
options]

Configuration Options:
  [-sds-cfg <required options> -sds-end]
    Required options:
       -arch <process arch> : [cortex-a9, cortex-a53, cortex-r5, microblaze]
       -os <OS type> : [standalone, freertos, linux]
       -name <configuration name>
       -id <configuration ID>
       -bif <bif file path>
       -readme <readme file path>
       -boot <boot files directory path>
       -lscript <linker script path> : Only for Standalone/FreeRTOS OSes
       -image <image directory path> : Only for Linux OSes
       -inc <include path> : adds path to list of include paths (can use
multiple times)
       -lib <library file path> : adds path to list of library files (can
use multiple times)
       -libfreertos <FreeRTOS library file path> : use custom FreeRTOS
library,only for
FreeRTOS OSes
       -incfreertos <FreeRTOS include path> : use custom FreeRTOS includes,
only for
FreeRTOS OSes
       -qemu-args <QEMU arguments file>

Optional Options:
```

```
  -o <output directory>
  -samples <samples directory path containing template.xml>
  -prebuilt <prebuilt directory path containing: portinfo.c/
h,apsys_0.xml,bitstream.bit,
<platform>.hdf,partitions.xml>
  -force : overwrites output directory if it already exists
  -cfg : load configuration saved from the GUI
  -gui : launch the GUI
  -verbose
  -version
  -help
```

Metadata Files

The SDSoC platform includes the following XML metadata files created by the SDSoC Platform Utility that describe the hardware and software interfaces.

- Top-Level Platform XML file (.xpfm): Every SDSoC platform includes a hardware platform metadata XML file, `<platform>.hpfm`, containing information about the platform hardware interfaces. SDSoC uses this information when creating the hardware system for your design, adding the data motion network and hardware accelerators along with the required connections for clocks, data and control signals. The top-level platform XML file is written by the SDSoC Platform Utility as `<platform>/<platform>.xpfm`, with references to the hardware and software XML files and the folders that contain them.
- Hardware Metadata file (.hpfm): The hardware platform XML metadata file is written as `<platform>/hw/<platform>.hpfm`, and is found with the Vivado platform project file (`.xpr`) and sources in `<platform>/hw/vivado`. It describes the hardware interfaces in the platform used by SDSoC when creating the hardware system containing the base hardware, hardware accelerators and data motion network.
- Software Metadata file (.spfm): The software platform XML file is written 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 a design 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 Utility by executing the following command, from within the **Output Directory** specified in the `sdspfm` GUI:

```
> sdscc -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 your custom platform repository it will read the platforms found there.

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

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

This command displays the details of the specified platform.

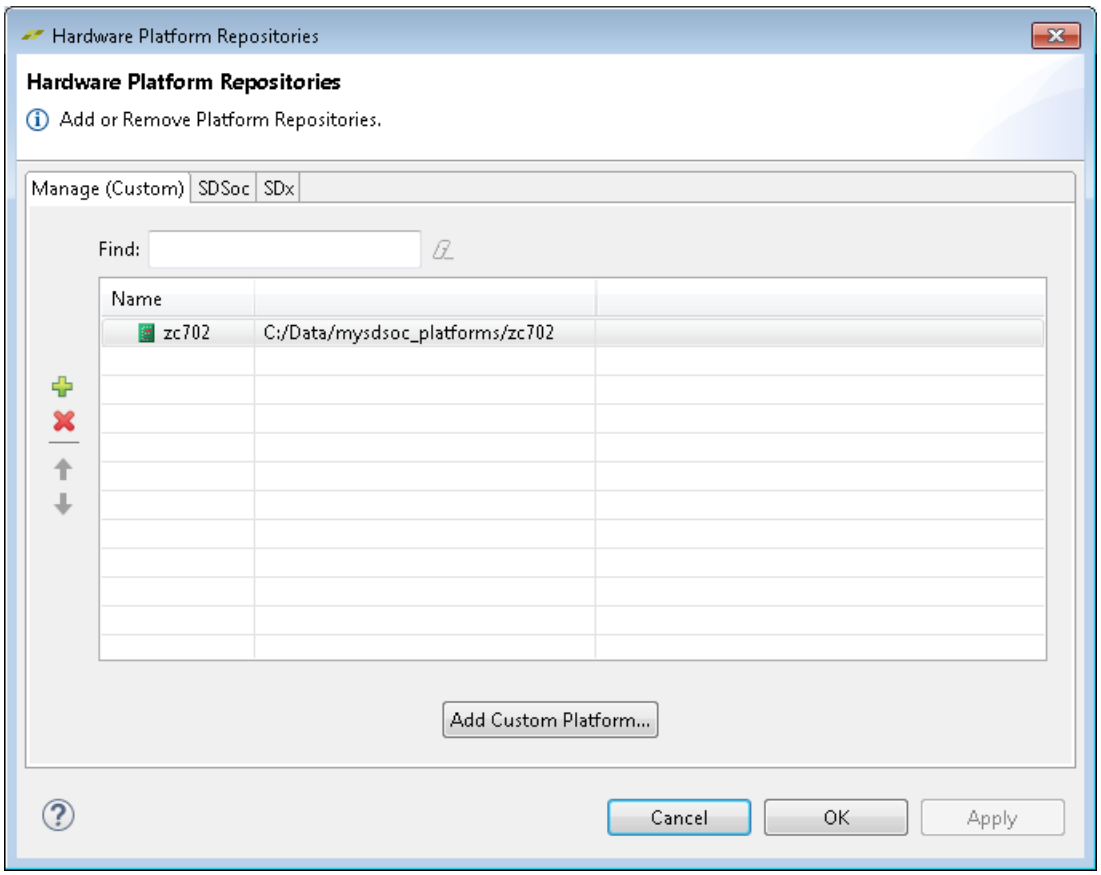
Finally, you can validate the hardware platform file directly using the following command while inside the *<platform>*/hw folder:

```
sds-pf-check <platform>.hpfm
```

This command simply determines if the hardware metadata file is valid, or invalid.

After checking the validity of the platform, you can add the hardware platform repository to SDx for use in new projects. In the SDx IDE you can use the **Xilinx → Add Custom Platform** command to add the newly created platform to the library of platforms available for SDSoC projects.

Figure 4: Add Hardware Platform



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

```
> sdscc -sds-pf <platform_name>
```

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

XILINX
ALL PROGRAMMABLE

Chapter 3

Hardware Platform Creation

Introduction

The hardware platform creation process consists of building a Vivado design and creating a Tcl script of SDSoC commands that captures the hardware interfaces supported, including clocks, interrupts, and bus interfaces. The SDSoC Platform Utility moves the Vivado project and associated files into the platform folder `<path_to_platform>/hw/vivado`.

This chapter assumes familiarity with the Vivado Design Suite and the ability to create a Vivado project for the hardware in your platform. It describes general requirements for the hardware in your platform, the Vivado project and the hardware platform folder.

Hardware Requirements

This section describes requirements on the hardware design component of an SDSoC platform. In general, nearly any design targeting the Zynq®-7000 All Programmable (AP) SoC or Zynq UltraScale+™ MPSoC 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. Create a Tcl script that uses the SDSoC Vivado Tcl Commands.
3. Specify the Vivado project and Tcl script in the SDSoC Platform Utility to build the platform.

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, the block diagram that has the same name as the hardware platform is the one that is used by the SDSoC Tcl script.*

- 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 SDSoC Tcl script.
- 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.

Vivado Design Suite Project

The SDx™ IDE uses the Vivado® Design Suite project file (*`platform`*`.xpr`) in the `<platform>/vivado` directory as a starting point to build an application-specific SoC. The project must include an IP Integrator block diagram and can contain any number of source files. Although nearly any project targeting a Zynq SoC or MPSoC can be the basis for an SDSoC project, there are a few restrictions as described in Hardware Requirements.

The Vivado Design Suite project must have the same name as the platform, and must reside in the following location:

`<platform>``/hw/vivado/`*`<platform>`*`.xpr`.

For example: `<path_to_install>/SDx/2017.x/platforms/zc702/hw/vivado/zc702.xpr`.

IMPORTANT:

*If you are moving the project file into the platform directory, 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 SDSoC `<platform>/vivado` directory.*

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 IP and output products.

Creating a Vivado Project

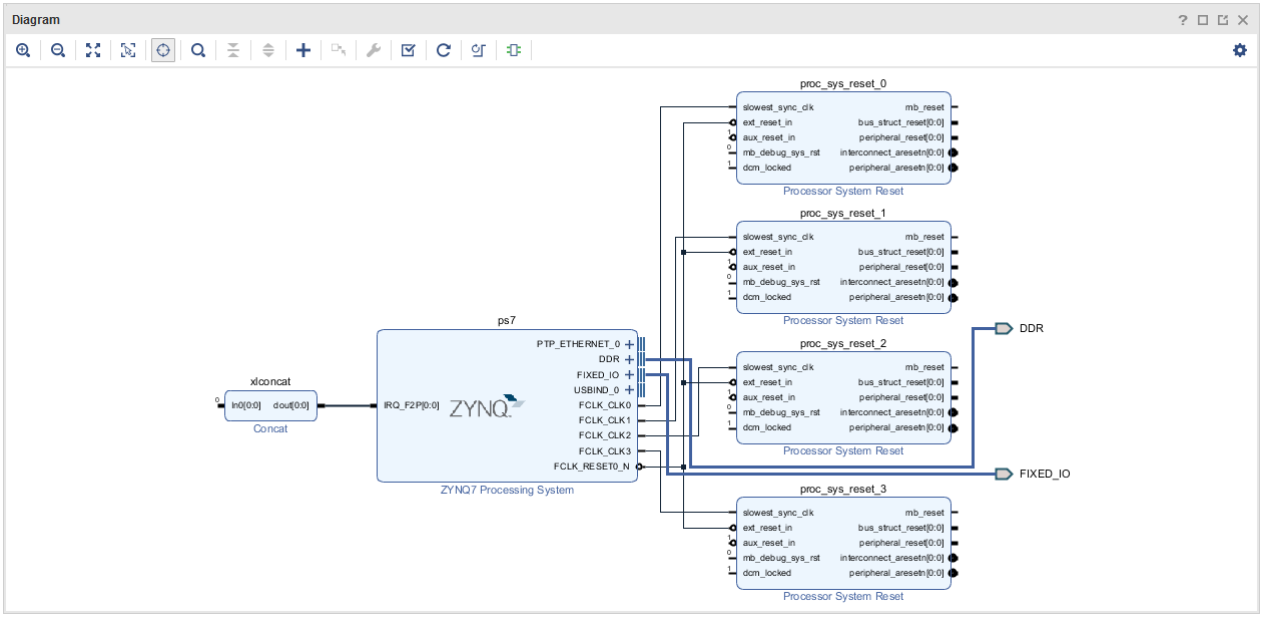
To create the Vivado Design Suite project for use in an SDSoC platform follow these steps:

1. Create a directory structure such as `<my_platform>/hw/vivado`
2. Launch the Vivado IDE.
3. From the Vivado Design Suite use the **Create New Project** command to create the platform project called `<my_platform>` in that directory.

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

4. Select the Xilinx device or a supported board, such as the ZC702 or Zybo board, to use for your SDSoC platform. For more information on creating projects and selecting parts or boards, refer to *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994).
5. 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>`.
6. 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 as shown in the following figure. For more information on creating an embedded processor block design refer to *Vivado Design Suite User Guide: Embedded Processor Hardware Design* (UG898).

Figure 5: Example Block Design in IP Integrator



7. **Validate** the block design to ensure everything is correct, and **Save** the design.
8. **Generate Output Products** of the IP in the block design.
9. Use the **Create Wrapper** command to create the top-level RTL design.
10. **Export Hardware** to SDK to provide the boot loaders and target operating system required for the software elements of the platform. Refer to Software Platform Data Creation for more information on defining the software platform libraries.
11. **Archive Project** and move it into the SDSoC platform directory structure if needed.

SDSoC Tcl Commands in Vivado

Introduction

After you complete the hardware platform design project in the Vivado Design Suite, you must create a Tcl script that the SDSoC Platform Utility uses to generate the hardware platform metadata file from the Vivado project. The Tcl script uses the SDSoC Vivado Tcl commands as described in the following section.

SDSoC Vivado Tcl Commands

The Vivado Design Suite provides SDSoC specific Tcl commands that specify the hardware interface of an SDSoC platform, which includes clock, reset, interrupt, AXI, and AXI4-Stream type interfaces. You must use the SDSoC Tcl commands in the Vivado Design Suite to create a script with the following steps:

1. Declare the hardware platform: (`sdsoc::create_pfm`)
2. Define the hardware platform name: (`sdsoc::pfm_name`)
3. Define a brief description of the platform: (`sdsoc::pfm_description`)
4. Declare the platform clock ports: (`sdsoc::pfm_clock`)
5. Declare the platform AXI bus interfaces: (`sdsoc::pfm_axi_port`)
6. Declare the platform AXI4-Stream bus interfaces: (`sdsoc::pfm_axis_port`)
7. Declare the available platform interrupts: (`sdsoc::pfm_irq`)
8. Write the hardware platform description metadata file: (`sdsoc::generate_hw_pfm`)

Each of these commands is detailed as follows. You may also refer to the Complete Example of this process provided below.

Defining the Hardware Platform Name and Description

The following describes the TCL API to be used within a block diagram in the Vivado IP Integrator feature.

- To create a new hardware platform file, set the name and description, use:

  ```
  sdsoc::create_pfm <platform>.hpfm
  ```

  Arguments:

  ```
  <platform>     - platform name
  ```

  Returns:

  ```
  new platform handle
  ```

- To set the platform name and description:

```
sdsoc::pfm_name        <platform handle> <vendor> <library> <platform>
<version>
sdsoc::pfm_description <platform handle> "<Description>"
```

Example:

```
set pfm [sdsoc::create_pfm zc702.hpfm]
sdsoc::pfm_name        $pfm "xilinx.com" "xd" "zc702" "1.0"
sdsoc::pfm_description $pfm "Zynq ZC702 Board"
```

IMPORTANT: *Your Tcl script must write the hardware platform metadata file with the `.hpfm` extension local to the Vivado project, and not in a separate directory. To conform to this requirement your Tcl script should use the command:*

```
set pfm [sdsoc::create_pfm <platform_name>.hpfm]
```

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. To declare clocks, use:

```
sdsoc::pfm_clock <pfm_handle> <port> <instance> <id> <is_default>
<proc_sys_reset>
```

Arguments:

Argument	Description
`pfm_handle`	`pfm` handle
`port`	Clock port name
`instance`	Instance name of the block that contains the port
`id`	Clock id (user-defined, must be a unique non-negative integer)
`is_default`	True if this is the default clock, false otherwise
`proc_sys_reset`	Corresponding `proc_sys_reset` block instance for synchronized reset signals

Every platform must declare one default clock for the SDSoC environment to use when no explicit clock has been specified. A clock is the default clock when the "is_default" argument is set to true.

Examples:

```
sdsoc::pfm_clock       $pfm FCLK_CLK0 ps7 0 false proc_sys_reset_0
sdsoc::pfm_clock       $pfm FCLK_CLK1 ps7 1 false proc_sys_reset_1
sdsoc::pfm_clock       $pfm FCLK_CLK2 ps7 2 true  proc_sys_reset_2
sdsoc::pfm_clock       $pfm FCLK_CLK3 ps7 3 false proc_sys_reset_3
```

Declaring AXI Ports

To declare AXI ports, use:

```
sdsoc::pfm_axi_port    <pfm> <axi_port> <instance> <memport>
```

Arguments:

Argument	Description
pfm	pfm handle
port	AXI port name
instance	Instance name of the block that contains the port
memport	Corresponding memory interface port type. Values: • 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 • MIG – An AXI slave connected to a MIG memory controller

Examples:

```
sdsoc::pfm_axi_port    $pfm M_AXI_GP0 ps7 M_AXI_GP
sdsoc::pfm_axi_port    $pfm M_AXI_GP1 ps7 M_AXI_GP
sdsoc::pfm_axi_port    $pfm S_AXI_ACP ps7 S_AXI_ACP
sdsoc::pfm_axi_port    $pfm S_AXI_HP0 ps7 S_AXI_HP
sdsoc::pfm_axi_port    $pfm S_AXI_HP1 ps7 S_AXI_HP
sdsoc::pfm_axi_port    $pfm S_AXI_HP2 ps7 S_AXI_HP
sdsoc::pfm_axi_port    $pfm S_AXI_HP3 ps7 S_AXI_HP
```

Example for an AXI interconnect:

```
sdsoc::pfm_axi_port    $pfm S01_AXI axi_interconnect_0 MIG
```

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 an 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 used within the platform itself. Declaring as many as possible will allow sds++ to avoid cascaded axi_interconnects in generated user systems.

As an example, the zc702_acp_pfm.tcl file includes the following declarations for interconnect ports connected to M_AXI_GP0 and S_AXI_ACP.

```
for {set i 1} {$i < 64} {incr i} {
  sdsoc::pfm_axi_port  $pfm M[format %02d $i]_AXI axi_ic_gp0 M_AXI_GP
}
for {set i 1} {$i < 8} {incr i} {
  sdsoc::pfm_axi_port  $pfm S[format %02d $i]_AXI axi_ic_acp S_AXI_ACP
}
```

Declaring AXI4-Stream Ports

To declare AXI4-Stream ports, use:

```
sdsoc::pfm_axis_port   <pfm> <axis_port> <instance> <type>
```

Arguments:

Argument	Description
`pfm`	`pfm` handle
`port`	AXI4-Stream port name
`instance`	Instance name of the block that contains the port
`type`	Interface type (values: M_AXIS, S_AXIS)

Examples:

```
sdsoc::pfm_axis_port   $pfm S_AXIS axis2io S_AXIS
sdsoc::pfm_axis_port   $pfm M_AXIS io2axis M_AXIS
```

Declaring Interrupt Ports

Interrupts must be connected to the platform Processing System 7 IP block through an IP integrator Concat block (`xlconcat`). If any IP within the platform includes interrupts, these must occupy the least significant bits of the Concat block without gaps.

To declare interrupt ports, use:

```
sdsoc::pfm_irq       <pfm> <port> <instance>
```

Arguments:

Argument	Description
pfm	pfm handle
port	irq port name
instance	Instance name of the concat block that contains the port

Example:

```
for {set i 0} {$i < 16} {incr i} {
  sdsoc::pfm_irq       $pfm In$i xlconcat
}
```

Declaring IO Devices

If you use the Linux UIO framework, you must declare the devices. To declare an instance to be a Linux IO platform device, use:

```
sdsoc::pfm_iodev       <pfm> <port> <instance> <type>
```

Arguments:

Argument	Description
pfm	pfm handle
port	I/O port name
instance	Instance name of the block that contains the UIO
type	I/O device type (e.g., UIO, KIO)

Example:

```
sdsoc::pfm_iodev       $pfm S_AXI axio_gpio_0 uio
```

Writing the Hardware Platform Description File

After using the above Tcl API commands to describe your platform, use the following to write the hardware platform description file:

```
sdsoc::generate_hw_pfm <pfm_handle>
```

Example:

```
sdsoc::generate_hw_pfm $pfm
```

This command will write the file specified in the sdsoc::create_pfm command.

Complete Example

All platforms included in the SDSoC release include the Tcl script used to generate the corresponding hardware description file. The Tcl script is located inside the `hw/vivado` directory and is called `<platform>_pfm.tcl`.

The following is a complete example of the usage of the Tcl API to generate a ZC702 platform

```
# zc702_pfm.tcl --
#
# This file uses the SDSoC Tcl Platform API to create the
# zc702 hardware platform file
#
# Copyright (c) 2015 Xilinx, Inc.
#
# Uncomment and modify the line below to source the API script
# source -notrace <SDSOC_INSTALL>/scripts/vivado/sdsoc_pfm.tcl
set pfm [sdsoc::create_pfm zc702_hw.pfm]
sdsoc::pfm_name        $pfm "xilinx.com" "xd" "zc702" "1.0"
sdsoc::pfm_description $pfm "Zynq ZC702 Board"
sdsoc::pfm_clock       $pfm FCLK_CLK0 ps7 0 false proc_sys_reset_0
sdsoc::pfm_clock       $pfm FCLK_CLK1 ps7 1 false proc_sys_reset_1
sdsoc::pfm_clock       $pfm FCLK_CLK2 ps7 2 true  proc_sys_reset_2
sdsoc::pfm_clock       $pfm FCLK_CLK3 ps7 3 false proc_sys_reset_3
sdsoc::pfm_axi_port    $pfm M_AXI_GP0 ps7 M_AXI_GP
sdsoc::pfm_axi_port    $pfm M_AXI_GP1 ps7 M_AXI_GP
sdsoc::pfm_axi_port    $pfm S_AXI_ACP ps7 S_AXI_ACP
sdsoc::pfm_axi_port    $pfm S_AXI_HP0 ps7 S_AXI_HP
sdsoc::pfm_axi_port    $pfm S_AXI_HP1 ps7 S_AXI_HP
sdsoc::pfm_axi_port    $pfm S_AXI_HP2 ps7 S_AXI_HP
sdsoc::pfm_axi_port    $pfm S_AXI_HP3 ps7 S_AXI_HP

for {set i 0} {$i < 16} {incr i} {
  sdsoc::pfm_irq       $pfm In$i xlconcat
}

sdsoc::generate_hw_pfm $pfm
```

XILINX
ALL PROGRAMMABLE

Chapter 4

Software Platform Data Creation

Introduction

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 Zynq®-7000 All Programmable (AP) SoC or Zynq UltraScale+™ MPSoC device, and generating a software platform metadata XML 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 XML file (.spfm) 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 to provide:

- Boot files - first stage bootloader or FSBL; U-boot; Linux unified boot 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 to save time and SDSoC data files.
- 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 Power Management Unit firmware (PMUFW) and ARM Trusted firmware (ATF).*

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

Software Requirements

This section describes requirements for the run-time software component of an SDSoC platform.

The SDx IDE currently supports Linux, standalone (bare metal), and FreeRTOS operating systems running on the Zynq®-7000 AP SoC target, but a platform is not required to support all of them. The SDx IDE supports Linux and standalone (bare-metal) operating systems running on the Zynq UltraScale+™ MPSoC.

When platform peripherals require Linux kernel drivers, you must configure the kernel to include several SDx IDE specific drivers which are available with the `linux-xlnx` kernel sources in the `drivers/staging/apf` directory. The base platforms included with the SDx environment provide instructions in README files, for example `platforms/zc702/sw/boot/generic.readme`.

This linux kernel and the associated device tree are based on the 4.6 version of the linux kernel. To build the kernel:

1. Clone/pull from the master branch of the Xilinx/linux-xlnx tree at github, and check out the `xilinx-v2016.4-sdsoc` tag.

   ```
   git checkout -b sdsoc_release_tag xilinx-v2017.1-sdsoc
   ```

2. Add the following CONFIG attributes to xilinx_zynq_defconfig and then configure the kernel.

   ```
   CONFIG_STAGING=y
   CONFIG_XILINX_APF=y
   CONFIG_XILINX_DMA_APF=y
   CONFIG_DMA_CMA=y
   CONFIG_CMA_SIZE_MBYTES=256
   CONFIG_CROSS_COMPILE="arm-linux-gnueabihf-"
   CONFIG_LOCALVERSION="-xilinx-apf"
   ```

   One way to do this is:

   ```
   cp arch/arm/configs/xilinx_zynq_defconfig arch/arm/configs/tmp_defconfig
   ```

3. Edit `arch/arm/configs/tmp_defconfig` using a text editor and add the above config lines to the bottom of the file.

   ```
   make ARCH=arm tmp_defconfig
   ```

4. Build the kernel using:

   ```
   make ARCH=arm UIMAGE_LOADADDR=0x8000 uImage
   ```

By default, the SDSoC system compiler generates an SD card image for booting the platform.

For creating a standalone platform, you must first build the hardware component using the IP Integrator feature of the Vivado Design Suite, and run the hardware export command (`write_hwdef`) to create the hardware handoff file. Using this newly generated hardware handoff file, use the SDx environment IDE to create a hardware platform project. From this project, you can create a new board support project. The `system.mss` file, as well as the linker script can now be delivered as part of the platform. Details of this process can be found in SDSoC Platform Examples.

Pre-built Hardware

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

The pre-built hardware should reside in a subdirectory of the platform software directory. Data in the subdirectory is pointed to by the `<sdx:prebuilt>` element of the `<sdx:configuration>` for the corresponding pre-built hardware.

For a given `<sdx:prebuilt sdx:data="prebuilt_platform_path"/>` in a platform xml, the location is:

```
<path_to_platform>/sw/prebuilt_platform_path
```

The path is relative to the software platform folder. For example, the element:

```
 <sdx:prebuilt sdx:data="prebuilt_data"/>
```

Indicates the prebuilt hardware bitstream and generated files are found in `<path_to_platform>/sw/prebuilt_data`. The `prebuilt_data` 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, use the following `sdscc` option:

```
-rebuild-hardware
```

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

- bitstream.bit
  - File found in `_sds/p0/ipi/<platform>.runs/impl_1/bitstream.bit`
- <platform>.hdf
  - Files found in `_sds/p0/ipi/<platform>.sdk`
- 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 reside in a subdirectory of the platform directory pointed to by the `sdx:includePaths` attribute for the corresponding OS in the platform software description file.

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

```
<platform root directory>/<relative_include_path>
```

Example:

For `sdx:includePaths="aarch32-linux/include"`:

```
<sdx_root>/samples/platforms/zc702_axis_io/sw/aarch32-linux/include/
zc702_axis_io.h
```

To use the header file in application code, use the following line:

```
#include "zc702_axis_io.h"
```

Use the colon (`:`) character to separate multiple include paths:

```
sdx:includePaths="<relative_include_path1>:<relative_include_path2>"
```

For example in a platform software description file that defines a list of two include paths:

```
<platform_root_directory>/<relative_include_path1>
<platform root_directory>/<relative_include_path2>
```

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. We recommend putting the files in the standard location as described in the platform XML file.*

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 pointed to by the `sdx:libraryPaths` attribute for the corresponding OS in the platform software description file.

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

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

Example:

For `sdx:libraryPaths="aarch32-linux/lib"`:

```
<sdx_root>/samples/platforms/zc702_axis_io/sw/aarch32-linux/lib/
libzc702_axis_io.a
```

To use the library file, use the following linker switch:

```
-lzc702_axis_io
```

Use the colon : character to separate multiple library paths. For example,

```
sdx:libraryPaths="<relative_lib_path1>:<relative_lib_path2>"
```

in a platform software description defines a list of two library paths

```
<platform_root>/sw/<relative_lib_path1>
<platform_root>/sw/<relative_lib_path2>
```

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. Xilinx recommend putting the files in the standard location as described in the platform software description file.*

Linux Boot Files

The SDx™ IDE can create an SD card image to boot the Linux operating system on the board. After booting completes, a Linux prompt is available for executing the compiled applications. For this, the SDx IDE requires several objects as part of the platform including:

- First Stage Boot Loader (FSBL)
- U-Boot
- Device Tree (optional)
- Linux Image
- Ramdisk Image (optional)

The SDx IDE permits the use of Linux images packaged in two forms:

1. As separate components consisting of a kernel image, ramdisk image, and device tree as separate files.
2. As a "unified boot" image, which consists of the three previously mentioned components packed into a single file, entitled `image.ub`.

For this reason, a Linux Image is mandatory, but a Ramdisk Image and Device Tree are optional, depending on how the `image.ub` gets packed. Refer to the *PetaLinux Tools Documentation Reference Guide* (UG1144) for more information.

TIP: *Additionally, a hardware platform that targets a Zynq UltraScale+™ MPSoC device requires PMU firmware, and ARM trusted firmware, which must subsequently be packed into BOOT.BIN.*

The SDx environment uses the Xilinx® bootgen utility program to combine the FSBL, PMU-FW, ARM trusted firmware, u-boot files with the bitstream into a `BOOT.BIN` file in a folder called `sd_card`, along with the required kernel image files, and the optional ramdisk/device-tree. The end-user copies the contents of this folder into the root of an SD card to boot the platform.

TIP: *For detailed instructions on how to build the boot files, refer to the Xilinx Wiki at http://wiki.xilinx.com. Under the heading of Zynq AP SoC & Zynq UltraScale+ MPSoC (ZU+) you will find links to topics like Building U-Boot, Building the Linux Kernel, and other topics..*

First Stage Boot Loader (FSBL)

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

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

Create a new software project **File** → **New** → **Application Project** with the name `fsbl` as you would using the Xilinx SDK.

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 System.

Once you generate the FSBL, you must copy it into a standard location for the SDx environment flow.

Example:

```
samples/platforms/zc702_axis_io/sw/boot/fsbl.elf
```

For the SDx system compiler to use an FSBL, a BIF file must point to it, as defined by the `sdx:bif` attribute of the `<sdx:image>` element. Refer to the Software Platform XML Metadata Reference for more information on the `sdx:bif` attribute. The file must reside in the `<path_to_platform>/sw/boot/` folder.

TIP: *The BIF file for a Zynq AP SoC is very different from the BIF file for the Zynq UltraScale+ MPSoC device.*

The following is an example `boot.bif` file for the Zynq®-7000 All Programmable (AP) SoC:

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

The following is an example `boot.bif` file for the Zynq UltraScale+™ MPSoC device:

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

U-Boot

U-Boot is an open source boot loader. Follow the Building U-Boot instructions at wiki.xilinx.com to download U-Boot and configure it for your platform. If you use PetaLinux to create Linux boot files, it will configure and create U-boot for you.

For the SDx system compiler to use a U-Boot, a BIF file must point to it, as defined by the `sdx:bif` attribute of the`<sdx:image>` element. Refer to the Software Platform XML Metadata Reference for more information on the `sdx:bif` attribute.

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

Example: `samples/platforms/zc702_axis_io/sw/boot/u-boot.elf`

Device Tree

The Device Tree is a data structure for describing hardware so that the details do not have to be hard coded in the operating system. This data structure is passed to the operating system at boot time. Use Xilinx SDK to generate the device tree for the platform. For detailed instructions on how to build the device-tree refer to the Build the Device Tree Compiler link on the Xilinx Wiki at wiki.xilinx.com to download the device tree generator support files, and install them for use with Xilinx SDK. There is one device tree per platform.

A small change is required to a device tree in order to make use of SDx:SDSoC in a booted linux system. You must manually add the following text at the bottom of the top-most device tree file:

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

IMPORTANT: *Note, SDK will NOT automatically generate this change. You must manually add this to the top-level device tree file .*

If PetaLinux is used to create Linux boot files, the device tree is included in the unified boot image file `image.ub`, rather than as a separate file.

Whether you provide an `image.ub` file, or separate Linux boot files (device tree, kernel, ramdisk), you will need to define the `sdx:imageData` attribute of the `<sdx:image>` element to specify the platform folder that contains these files:

```
sdx:imageData=image
```

Location: `samples/platforms/zc702_axis_io/sw/image`

Linux Image

A Linux image is required to boot the hardware platform. Xilinx provides single platform-independent pre-built Linux image that works with all the SDSoC platforms supplied by Xilinx.

However, if you want to configure Linux for your own platform, follow the instructions at wiki.xilinx.com to download and build the Linux kernel. Make sure you enable the SDx IDE APF drivers, and the Contiguous Memory Allocator (CMA) when configuring Linux for your platform.

If PetaLinux is used to create Linux boot files, the Linux kernel image is included in the unified boot image file, `image.ub`, rather than as a separate file.

Whether you provide an `image.ub` file, or separate Linux boot files (device tree, kernel, ramdisk), you will need to define the `sdx:imageData` attribute of the `<sdx:image>` element to specify the platform folder that contains these files. For `sdx:imageData="image"` the location of the `image.ub` file would be `<path_to_platform>/sw/image/image.ub`.

Sample xml description:

```
sdx:imageData="image"
```

Location: `samples/platforms/zc702_axis_io/sw/image`

Ramdisk Image

A ramdisk image is required to boot. A single ramdisk image is included as part of the SDx environment IDE installation. If you need to modify it, or create a new ramdisk, follow the instructions at wiki.xilinx.com.

If PetaLinux is used to create Linux boot files, the ramdisk is included in the unified boot image file, `image.ub`, rather than as a separate file.

Whether you provide an `image.ub` file, or separate Linux boot files (device tree, kernel, ramdisk), you will need to define the `sdx:imageData` attribute of the `<sdx:image>` element to specify the platform folder that contains these files.

Sample xml description:

```
sdx:imageData="image"
```

Location: `samples/platforms/zc702_axis_io/sw/image`

Using PetaLinux to Create 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:

- Set up your shell environment with PetaLinux tools in your PATH environment variable.
- Create and `cd` into a working directory.
- Create a new PetaLinux project targeting a BSP that corresponds to the type of board you are targetting:

  ```
  petalinux-create -t project <path_to_project> -s <path_to_base_BSP>
  ```

- 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 can be produced with the use of the Vivado Design Suite project included in `<install>/SDx/2017.1/platforms/<platform name>/hw/vivado` for a specific platform. Implement the Vivado project to generate the bitstream, and export the HDF with the bitstream included.*

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 unified boot 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.

To build the PetaLinux image, use the following steps:

1. Source PetaLinux `settings.sh`

2. Create a PetaLinux project against base BSP corresponding to the selected board (for example, for zc702: xilinx-zc702-v2017.1-final.bsp)

```
petalinux-create -t project -n <platform name> -s <petalinux_install>/<base BSP name>
```

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

```
petalinux-config -p <platform name> --get-hw-description=<HDF path>
```

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)

4. Configure PetaLinux kernel:

```
petalinux-config -p <platform name> -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)*

5. Configure petalinux rootfs:

```
petalinux-config -p <platform name> -c rootfs
```

Add stdc++ libs:

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

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

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

7. Build the PetaLinux image:
   - petalinux-build
8. In the directory *<petalinux project>*/images/linux/ there are a number of important files that are partitioned into two categories:
   a. 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.
   b. 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 *fslb.elf ./boot/fsbl.elf
$ cp bl31.elf ./boot/bl31.elf
$ cp linux/pmufw.elf ./boot/pmufw.elf
$ cp image.ub ./image/image.ub
```

9. Finally, create a boot image file, or BIF, that is used to compile the contents of the boot folder into a BOOT.BIN file.

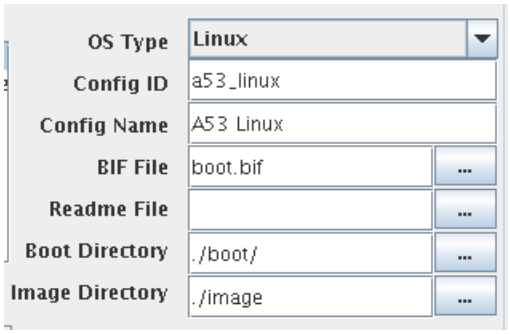
   An SDSoC boot image file looks similar to a standard BIF, with symbolic constants instead of paths specified. SDSoC typically relies on a .bif file that is written using patterns that are replaced with generated content, rather than direct paths to boot files. This is because the bitstream file will be procedurally generated, and some of the elements do not have known file names at the time the BIF file is being created.

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

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

Taken together, the `boot` directory, the `image` directory, and the `.bif` file, constitute the software artifacts that the SDSoC Platform Utility needs as input for the Linux OS:

Figure 6: Linux Boot Information



Standalone Boot Files

If no OS is required, you can create a boot image that automatically executes a generated executable.

First Stage Boot Loader (FSBL)

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

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

Create a new software project **File** → **New** → **Application Project** with the name `fsbl` as you would using the Xilinx SDK.

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 System.

Once you generate the FSBL, you must copy it into a standard location for the SDx environment flow.

Example:

```
samples/platforms/zc702_axis_io/sw/boot/fsbl.elf
```

For the SDx system compiler to use an FSBL, a BIF file must point to it, as defined by the `sdx:bif` attribute of the `<sdx:image>` element. Refer to the Software Platform XML Metadata Reference for more information on the `sdx:bif` attribute. The file must reside in the `<path_to_platform>/sw/boot/` folder.

TIP: *The BIF file for a Zynq AP SoC is very different from the BIF file for the Zynq UltraScale+ MPSoC device.*

The following is an example `boot.bif` file for the Zynq®-7000 All Programmable (AP) SoC:

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

The following is an example `boot.bif` file for the Zynq UltraScale+™ MPSoC device:

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

Executable

For the SDx environment to use an executable in a boot image, a BIF file must point to it.

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

The SDx environment automatically inserts the generated bitstream and ELF files.

FreeRTOS Configuration/Version Change

The SDx™ IDE FreeRTOS support uses a pre-built library using the default `FreeRTOSConfig.h` file included with the v8.2.3 software distribution, along with a predefined linker script.

To change the FreeRTOS v8.2.3 configuration or its linker script, or use a different version of FreeRTOS, follow the steps below:

1. Copy the folder `<path_to_install>/SDx/<version>/platforms/zc702` to a local folder.
2. To just modify the default linker script, modify the file `<path_to_your_platform>/zc702/sw/freertos/lscript.ld`.

3. To change the FreeRTOS configuration (`FreeRTOSConfig.h`) or version:
   a. Build a FeeRTOS library as `libfreertos.a`.
   b. Add include files to the folder `<path_to_your_platform>/zc702/sw/freertos/include`.
   c. Add the library `libfreertos.a` to `<path_to_your_platform>/zc702/sw/freertos/li`.
   d. Change the paths in `<path_to_your_platform>/zc702/zc702.spfm` for the section containing the line `("sdx:os sdx_name="freertos" (sdx:includePaths="/aarch32-none/include and sdx:libraryPaths="/aarch-32-none/lib/freertos").`
4. In your makefile, change the SDSoC platform option from `-sds-pf zc702` to `-sds-pf <path_to_your_platform>/zc702`.
5. Rebuild the library:

   The SDx IDE folder `<path_to_install>/SDx/2016.x/tps/FreeRTOS` includes the source files used to build the pre-configured FreeRTOS v8.2.3 library `libfreertos.a`, along with a simple makefile and an `SDSoC_readme.txt` file. See the `SDSoC_readme.txt` file for additional requirements and instructions.
   a. Open a command shell.
   b. Run the SDx IDE `<path_to_install>/SDx/2016.x/settings64.sh` script, or `settings64.bat`, to set up the environment to run command line tools (including the ARM GNU toolchain for the Zynq®-7000 AP SoC).
   c. Copy the folder to a local folder.
   d. Modify `FreeRTOSConfig.h`.
   e. Run the make command.

If you are not using FreeRTOS v8.2.3, see the notes in the `SDSoC_readme.txt` file describing how the source was derived from the official software distribution. After uncompressing the ZIP file, a very small number of changes were made (incorporate `memcpy`, `memset` and `memcmp` from the demo application `main.c` into a library source file and change include file references from `Task.h` to `task.h`) but the folder structure is the same as the original. If the folder structure is preserved, the makefile created to build the preconfigured FreeRTOS v8.2.3 library can be used.

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/2017.1/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="http://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="http://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>	Boolean function that defines the matching 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.
<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 SDSoC 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 SDx 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 SDx 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 SDx 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 SDSoC 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"/>
```

XILINX
ALL PROGRAMMABLE

Appendix A

Platform Checklist

Overview

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 XML 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 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 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 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 unified 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). In addition, 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 `template.xml` file used by the SDx Environment IDE New Project Wizard when creating an application.
   - See Platform Sample Applications.
4. Create a platform directory with platform XML metadata files.
   - The basic platform directory structure is shown below. For folders and files not listed, their naming is flexible, otherwise use the specified names (replace *<platform>* with the name of your platform). The samples folder is optional.
     - myplatforms
       - *<platform>*
         - *<platform>*.xpfm
         - hw
           - vivado
           - *<platform>*.hpfm
         - sw
           - *<processor_boot_folder>*
           - *<platform>*.spfm
         - samples
           - *<sample_application_folder>*
           - template.xml
5. Validate your platform supports the SDSoC environment.
   - The 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 with the SDx Environment 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 Environment IDE.
   - Start the SDx Environment IDE and create an SDSoC project using the New Project Wizard. After specifying a project name, you are presented with a list of platforms. Click on Add Custom Platform to navigate to your platform folder and 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.

XILINX
ALL PROGRAMMABLE

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 platform to the latest release, updating the IP used in the design, and regenerating output products for the project. Updating the IP Integrator design may be a simple mater of plugging in the latest IP revision for the current release. However, it can also be complicated by the addition or removal of interface signals on the IP, or updated parameters, in the case of major version changes from one release to another. In this case, upgrading the Vivado Design Suite project can require more effort.

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 the top-level platform and hardware description files using the SDSoC Platform Utility. 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 Utility 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 Vivado Design Suite 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.

XILINX
ALL PROGRAMMABLE

Appendix C

Tutorial: Using the SDSoC Platform Utility

Introduction

This appendix provides a simple tutorial for creating example platforms. The majority of settings for a platform can be configured in the SDSoC Platform Utility GUI, and the platform generated from the configuration. After the platform has been created it can be modified to provide further customization for advanced use cases.

When working through the following tutorial labs, you should begin by copying the sample files provided at `<install>/SDx/2017.1/samples/sdspfm` to a new location such as `C:/temp`, or `/temp`. Copying the `samples/sdspfm` folder will allow you to make changes to the source files while preserving the original files, as well as avoiding any permission issues with regard to the installation directory.

TIP: *In addition to the platform examples in the following labs, you should also examine the standard SDx platforms that are included in the `<install>/SDx/2017.1/platforms` directory.*

Lab1: Creating the ZC702 Platform

IMPORTANT: *In the following tutorial lab the samples folder location is called `<samples_dir>`, and represents your local copy of the sample platforms from the SDx installation.*

This tutorial will recreate the base platform for the ZC702 platform that is delivered in the SDSoC installation. It demonstrates how to start and interact with the SDSoC Platform Utility GUI, and create a simple platform targeting Linux, FreeRTOS, and Standalone OSes. For more information on the SDSoC Platform Utility refer to Creating an SDSoC Platform.

1. Launch an SDx Terminal window:
   - From Windows, choose **Start → All Programs → Xilinx Design Tools → SDx 2017.1 → SDx Terminal 2017.1**.
   - From Linux, source the `settings.sh/csh` file as appropriate for the shell type you're using.

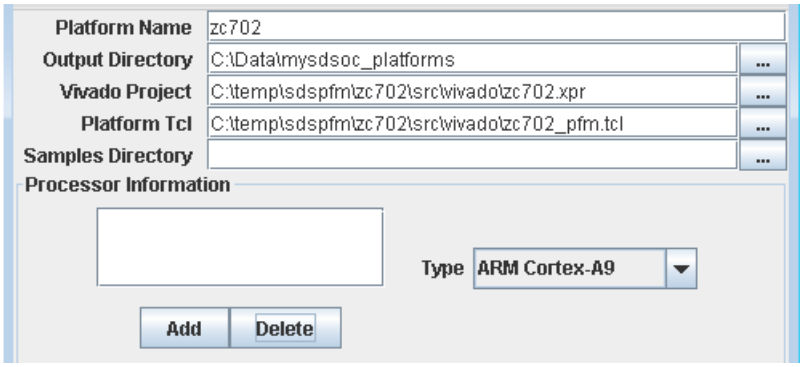
2. Launch the SDSoC Platform Utility GUI from the SDx Terminal:

IMPORTANT: *The SDSoC Platform Utility should be launched from the C:/ directory on the Windows OS, or an equivalent root directory, to avoid path name length limitations that can cause problems when generating the platform files on Windows.*

   - In your terminal or shell type `sdspfm -gui` to launch the GUI.

3. After the SDSoC Platform Utility GUI opens, enter the following info for the base platform information to match the following figure.

Figure 7: Specify Hardware Platform



   - **Platform Name**: zc702

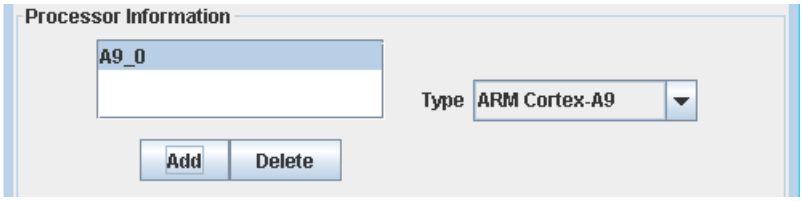
IMPORTANT: *The platform name must match the name of the Vivado Design Suite project that defines the hardware, as well as the IP Integrator block design contained in the project.*

   - Choose an appropriate **Output Directory** for your system.
   - The **Vivado Project** and **Platform Tcl** files can be found in the `<samples_dir>/zc702/src/vivado` directory.

4. Notice that after the Vivado project file, `zc702.xpr`, has been specified, the **Processor Type** field displays `ARM Cortex-A9`.
5. Click the **Add** button in the Processor Information pane to add the A9 processor to the platform configuration.

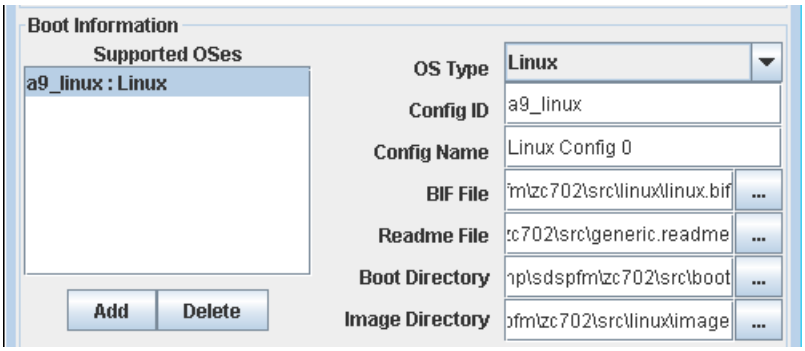
TIP: *Only one processor of each type is required for the platform configuration, although your platform will still be able to use all cores.*

Figure 8: Add Processor



6. In the Boot Information pane you will configure the Linux OS first, but the order you configure OSes is not important.

Figure 9: Linux Boot Information



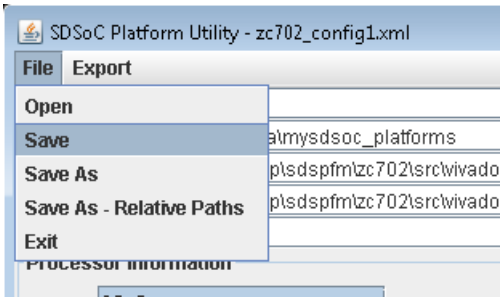
a. Click the **Add** button in the Boot Information panel to add a new configuration.
b. **Config ID:** `a9_linux`, although this can be any name you want.
c. **OS Type**: `Linux`, selected from the drop down menu.
d. **Config Name**: This name is automatically generated, but can be edited as needed.
e. **BIF File**: `<samples_dir>/zc702/src/linux/linux.bif`
f. **Readme File**: `<samples_dir>/zc702/src/generic.readme`
g. **Boot Directory**: `<samples_dir>/zc702/src/boot`
h. **Image Directory**: `<samples_dir>/zc702/src/linux/image`

TIP: *OS settings are automatically saved to the selected configuration when a change is made to the settings on the right.*

---

7. At this point you can save the platform configuration to a file by choosing **File** → **Save** from the menu at the top of the GUI. You'll be prompted to enter a name and location for the configuration file. You can save the platform configuration file to whatever location you prefer, but you should not store it inside of the generated SDSoC platform as regenerating the platform may delete the configuration file.

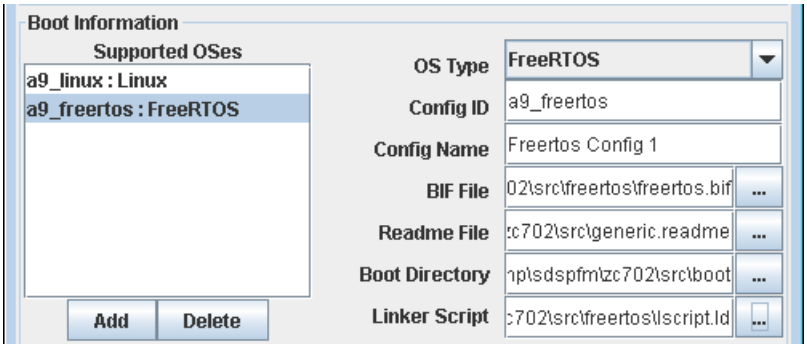
Figure 10: Save Configuration



Saving the platform configuration file lets you reload the details of the platform in the SDSoC Platform Utility and regenerate it as needed, or use it as the starting point for a new platform. You can load a saved configuration file using the **File** → **Open** command.

8. Add and configure FreeRTOS to the platform using the following settings:

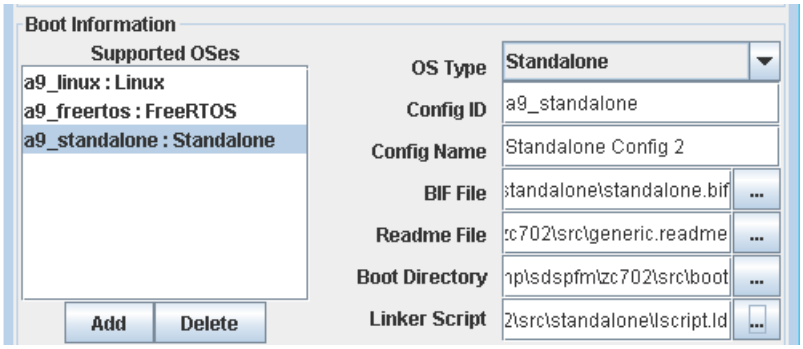
Figure 11: FreeRTOS Configuration



a. Click the **Add** button in the Boot Information panel to add a new configuration.
b. **Config ID:** `a9_freertos`.
c. **OS Type**: `FreeRTOS`, selected from the drop down menu.
d. **Config Name**: Change this name to `FreeRTOS Config 1`.
e. **BIF File**: `<samples_dir>/zc702/src/freertos/freertos.bif`
f. **Readme File**: `<samples_dir>/zc702/src/generic.readme`
g. **Boot Directory**: `<samples_dir>/zc702/src/boot`
h. **Linker Script**: `<samples_dir>/zc702/src/freertos/lscript.ld`

9. Add and configure Standalone OS to the platform using the following settings:

Figure 12: Standalone Configuration



a. Click the **Add** button in the Boot Information panel to add a new configuration.
b. **Config ID:** `a9_standalone`.
c. **OS Type**: `Standalone`, selected from the drop down menu.
d. **Config Name**: Change this name to `Standalone Config 2`.
e. **BIF File**: `<samples_dir>/zc702/src/standalone/standalone.bif`
f. **Readme File**: `<samples_dir>/zc702/src/generic.readme`
g. **Boot Directory**: `<samples_dir>/zc702/src/boot`
h. **Linker Script**: `<samples_dir>/zc702/src/standalone/lscript.ld`

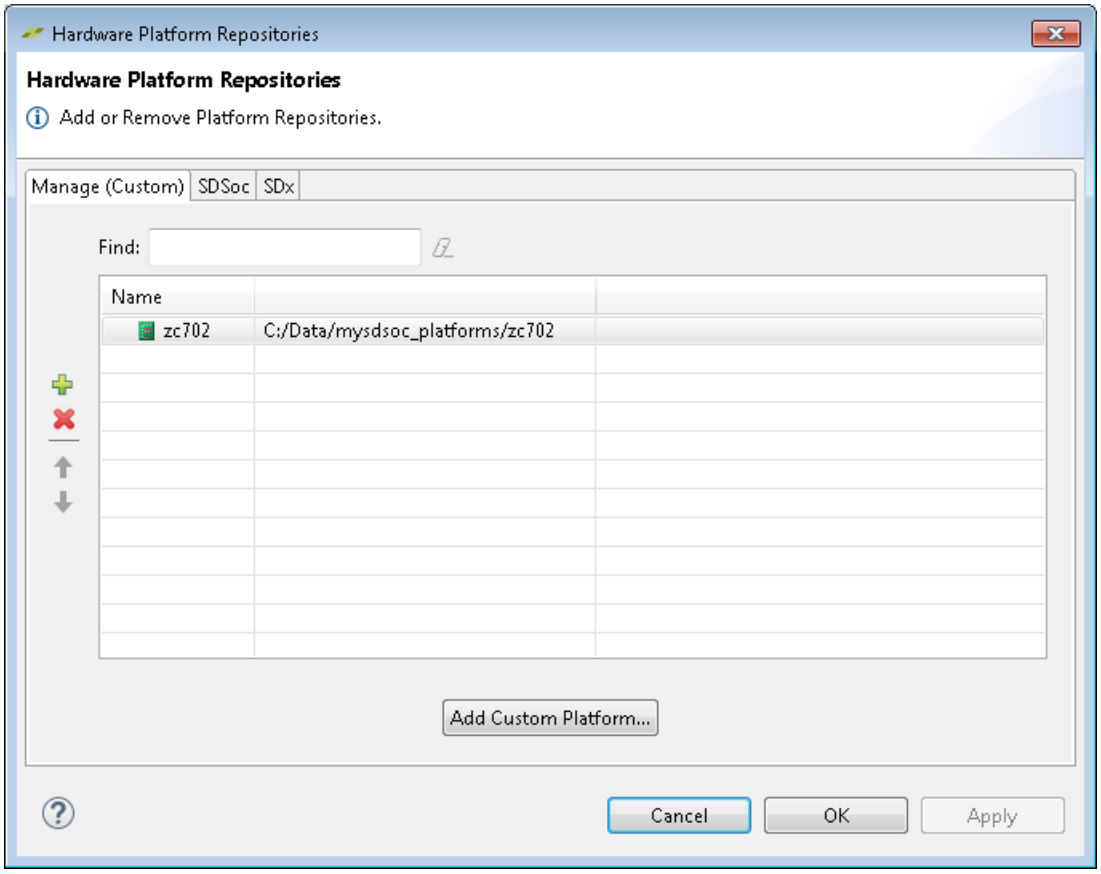
10. Save the configuration file again, using the **File** → **Save** command.

11. Click the **Generate** button at the bottom of the GUI. This causes the GUI to send your platform configuration settings to the `sdspfm` command line utility which creates your platform. You can see progress updates in the SDx Terminal from where you launched the GUI as your platform is created. When your platform is finished generating, you'll see a popup message in the GUI with the output from the SDx Terminal if there are any error messages. Click **Ok** to close this message.
12. Exit the SDSoC Platform Utility by using the **File** → **Exit** command.

With the platform created, you can navigate to the output directory to view the generated platform files. Notice that the Vivado project is included in the `hw` folder, and the various OS configurations are included in the `sw` folder.

From within the SDx IDE you can use the **Xilinx** → **Add Custom Platform** command to add the newly created platform to the library of platforms available for SDx projects. You can also move your platform anywhere on the file system, prior to adding it to the library.

Figure 13: Add Hardware Platform



Lab2: ZC702_AXIS_IO Platform

IMPORTANT: *In the following tutorial lab the samples folder location is called `<samples_dir>`, and represents your local copy of the sample platforms from the SDx installation.*

In Lab #1, you created a basic ZC702 platform with Linux, Standalone, and FreeRTOS OSes. The lab discussed how to use the SDSoC Platform Utility GUI to configure the platform, save the configuration, and generate the platform. In this tutorial you will create the `zc702_axis_io` platform using the platform utility. See Example: Direct I/O in an SDSoC Platform for a more detailed discussion of this platform.

The `zc702_axis_io` platform has a platform-specific include header, and a library that must be linked. It also contains a few sample applications. To begin this exercise, you will make the static library file for the platform.

1. Launch an SDx Terminal window:
   - From Windows, choose **Start → All Programs → Xilinx Design Tools → SDx 2017.1 → SDx Terminal 2017.1**.
   - From Linux, source the `settings.sh/csh` file as appropriate for the shell type you're using.
2. From this terminal window you will need to navigate to the include folder for the `zc702_axis_io` platform, and `make` the static library (`.a`) file for both the Linux and Standalone operating systems:

```
cd <samples_dir>/zc702_axis_io/src/src
cd linux
make
cd ../standalone
make
cd C:/
```

With the static library file made, you can now use the SDx Terminal window to define the `zc702_axis_io` platform configuration file and generate the platform.

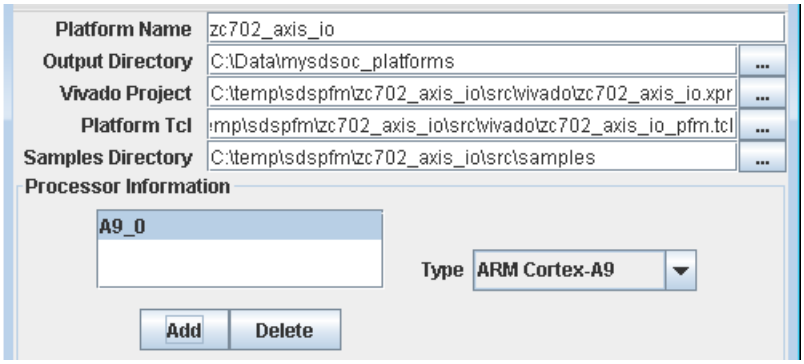
1. Launch the SDSoC Platform Utility GUI from the SDx Terminal:

IMPORTANT: *The SDSoC Platform Utility should be launched from the C:/ directory on the Windows OS, or an equivalent root directory, to avoid path name length limitations that can cause problems when generating the platform files on Windows.*

   - In your terminal or shell type `sdspfm -gui` to launch the GUI.

2. After the SDSoC Platform Utility GUI opens, enter the following info for the base platform information to match the following figure.

Figure 14: Specify Hardware Platform



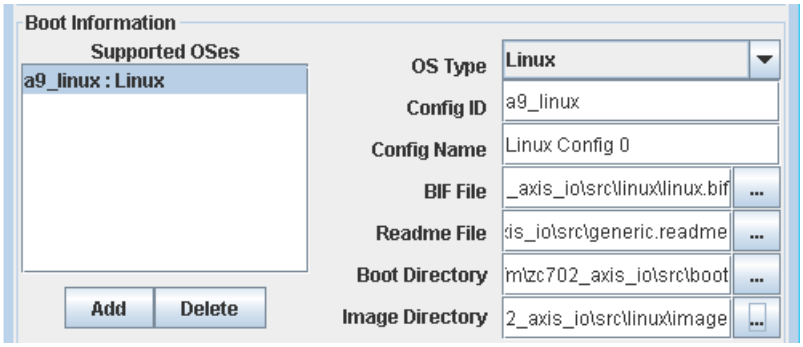
- **Platform Name**: zc702_axis_io
- Choose an appropriate **Output Directory** for your system.
- The **Vivado Project** and **Platform Tcl** files can be found in the `<samples_dir>/zc702_axis_io/src/vivado` directory.
- **Samples Directory**: `<samples_dir>/zc702_axis_io/src/samples`

3. Click the **Add** button in the Processor Information pane to add the A9 processor to the platform configuration.

TIP: *Only one processor of each type is required for the platform configuration, although your platform will still be able to use all cores.*

4. In the Boot Information pane you will configure the Linux OS first, but the order you configure OSes is not important.

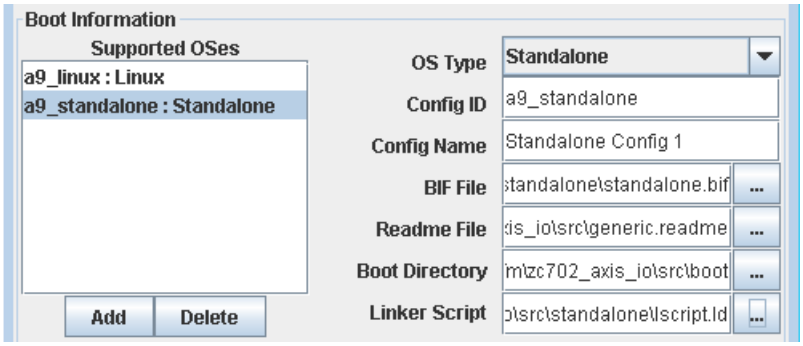
Figure 15: Linux Boot Information



a. Click the **Add** button in the Boot Information panel to add a new configuration.
b. **Config ID:** `a9_linux`, although this can be any name you want.
c. **OS Type**: `Linux`, selected from the drop down menu.
d. **Config Name**: This name is automatically generated, but can be edited as needed.
e. **BIF File**: `<samples_dir>/zc702_axis_io/src/linux/linux.bif`
f. **Readme File**: `<samples_dir>/zc702_axis_io/src/generic.readme`
g. **Boot Directory**: `<samples_dir>/zc702_axis_io/src/boot`
h. **Image Directory**: `<samples_dir>/zc702_axis_io/src/linux/image`

5. Add and configure Standalone OS to the platform using the following settings:

Figure 16: Standalone Configuration



a. Click the **Add** button in the Boot Information panel to add a new configuration.
b. **Config ID:** `a9_standalone`.
c. **OS Type**: `Standalone`, selected from the drop down menu.
d. **Config Name**: Change this name to `Standalone Config 1`.
e. **BIF File**: `<samples_dir>/zc702_axis_io/src/standalone/standalone.bif`
f. **Readme File**: `<samples_dir>/zc702_axis_io/src/generic.readme`
g. **Boot Directory**: `<samples_dir>/zc702_axis_io/src/boot`
h. **Linker Script**: `<samples_dir>/zc702_axis_io/src/standalone/lscript.ld`

6. Under **Supported OSes**, select the Linux OS, `a9_linux` and configure the Include Paths and Libraries for this configuration:
   - On the **Include Paths** tab, specify the include path as `<samples_dir>/zc702_axis_io/src/src/inc` and click **Add** to add it to the list.
   - On the **Libraries** tab, in the **Library Path** field select the `<samples_dir>/zc702_axis_io/src/src/linux/libzc702_axis_io.a` file and click **Add** to add it to the list.

Figure 17: Include Paths and Libraries

7. Select the Standalone OS, `a9_standalone` and configure the Include Paths and Libraries for this configuration:
   - On the **Include Paths** tab, specify the include path as `<samples_dir>/zc702_axis_io/src/src/inc` and click **Add** to add it to the list.
   - On the **Libraries** tab, in the **Library Path** field select the `<samples_dir>/zc702_axis_io/src/src/standalone/libzc702_axis_io.a` file and click **Add** to add it to the list.
8. Save the configuration file using the **File** → **Save** command, and specify a name and location for the platform configuration file.
9. Click the **Generate** button to create your platform. You can see progress updates in the SDx Terminal from where you launched the GUI as your platform is created. When your platform is finished generating, you'll see a popup message in the GUI with the output from the SDx Terminal if there are any error messages. Click **Ok** to close this message.
10. Exit the SDSoC Platform Utility by using the **File** → **Exit** command.

Lab3: ZCU102 Platform

IMPORTANT: *In the following tutorial lab the samples folder location is called `<samples_dir>`, and represents your local copy of the sample platforms from the SDx installation.*

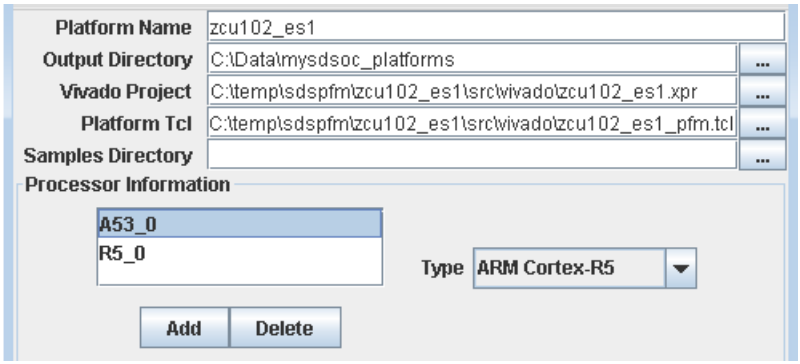
In this exercise, you will create the `zcu102_es1` platform.

1. Launch an SDx Terminal window:
   - From Windows, choose **Start → All Programs → Xilinx Design Tools → SDx 2017.1 → SDx Terminal 2017.1**.
   - From Linux, source the `settings.sh/csh` file as appropriate for the shell type you're using.
2. Launch the SDSoC Platform Utility GUI from the SDx Terminal:

IMPORTANT: *The SDSoC Platform Utility should be launched from the C:/ directory on the Windows OS, or an equivalent root directory, to avoid path name length limitations that can cause problems when generating the platform files on Windows.*

   - In your terminal or shell type `sdspfm -gui` to launch the GUI.
3. After the SDSoC Platform Utility GUI opens, enter the following info for the base platform information to match the following figure.

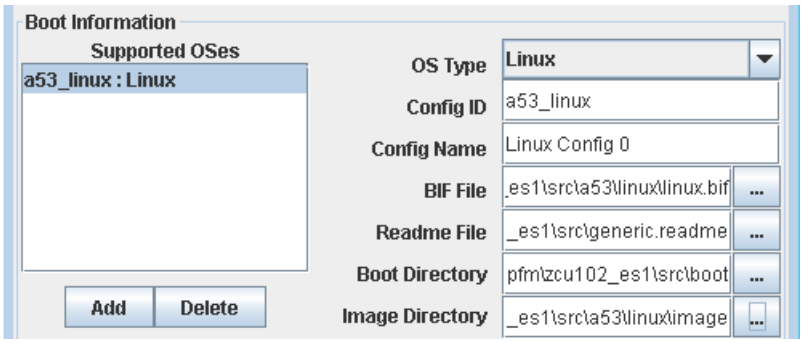
Figure 18: Specify Hardware Platform



   - **Platform Name**: zcu102_es1
   - Choose an appropriate **Output Directory** for your system.
   - The **Vivado Project** and **Platform Tcl** files can be found in the *<samples_dir>*`/zcu102_es1/src/vivado` directory.
4. Under **Processor Information**, in the **Type** field, select the `ARM Cortex-A53` processor and click **Add** to add the processor core to the platform configuration.
5. Again in the **Type** field, select the `ARM Cortex-R5` processor and click **Add** to add this core to the platform configuration as well.

6. With the `A53_0` processor selected, add and configure the Linux OS under **Boot Information**:

Figure 19: A53 Processor Core - Linux Boot Information



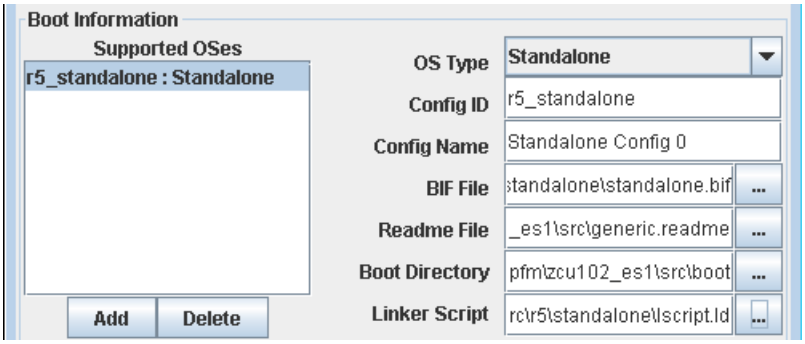
   a. Click the **Add** button in the Boot Information panel to add a new configuration.
   b. **Config ID:** `a53_linux`, although this can be any name you want.
   c. **OS Type**: `Linux`, selected from the drop down menu.
   d. **Config Name**: This name is automatically generated, but can be edited as needed.
   e. **BIF File**: *<samples_dir>*`/zcu102_es1/src/a53/linux/linux.bif`
   f. **Readme File**: *<samples_dir>*`/zcu102_es1/src/generic.readme`
   g. **Boot Directory**: *<samples_dir>*`/zcu102_es1/src/boot`
   h. **Image Directory**: *<samples_dir>*`/zcu102_es1/src/a53/linux/image`

7. With the `A53_0` processor still selected, add and configure Standalone OS to the platform using the following settings:

   a. Click the **Add** button in the Boot Information panel to add a new configuration.
   b. **Config ID:** `a53_standalone`.
   c. **OS Type**: `Standalone`, selected from the drop down menu.
   d. **Config Name**: Change this name to `Standalone Config 1`.
   e. **BIF File**: *<samples_dir>*`/zcu102_es1/src/a53/standalone/standalone.bif`
   f. **Readme File**: *<samples_dir>*`/zcu102_es1/src/generic.readme`
   g. **Boot Directory**: *<samples_dir>*`/zcu102_es1/src/boot`
   h. **Linker Script**: *<samples_dir>*`/zcu102_es1/src/a53/standalone/lscript.ld`

8. Under **Processor Information** select the `R5_0` processor, and add and configure the Standalone OS for this processor core:

Figure 20: R5 Processor Core - Standalone Boot Information



   a. Click the **Add** button in the Boot Information panel to add a new configuration.
   b. **Config ID:** `r5_standalone`.
   c. **OS Type**: `Standalone`, selected from the drop down menu.
   d. **Config Name**: Change this name to `Standalone Config 0`.
   e. **BIF File**: `<samples_dir>/zcu102_es1/src/r5/standalone/standalone.bif`
   f. **Readme File**: `<samples_dir>/zcu102_es1/src/generic.readme`
   g. **Boot Directory**: `<samples_dir>/zcu102_es1/src/boot`
   h. **Linker Script**: `<samples_dir>/zcu102_es1/src/r5/standalone/lscript.ld`

9. Save the configuration file using the **File** → **Save** command, and specify a name and location for the platform configuration file.
10. Click the **Generate** button to create your platform. You can see progress updates in the SDx Terminal from where you launched the GUI as your platform is created. When your platform is finished generating, you'll see a popup message in the GUI with the output from the SDx Terminal if there are any error messages. Click **Ok** to close this message.
11. Exit the SDSoC Platform Utility by using the **File** → **Exit** command.

XILINX
ALL PROGRAMMABLE

Appendix D

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.

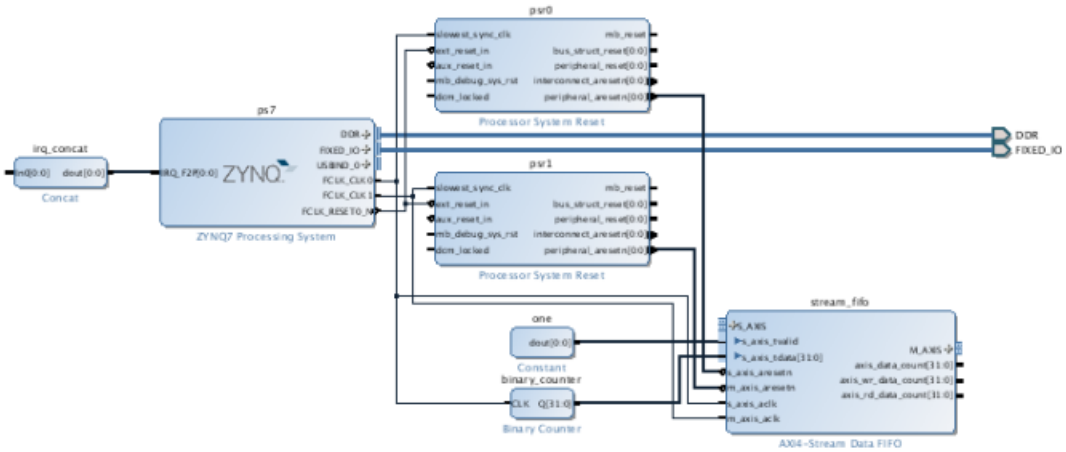
Example: Direct I/O in an SDSoC Platform

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. In "Using External I/O" the *SDSoC Environment User Guide* (UG1027) includes a discussion of the `zc702_axis_io` sample platform. 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/sdspfm/zc702_axis_io`.

The hardware component of an SDSoC platform is represented in a Vivado project which is located in the `src/vivado` subdirectory. In an SDx Terminal shell, you can view the block diagram by opening the Vivado project using the command, `vivado zc702_axis_io.xpr` and clicking **Open Block Design** on the left in the Flow Navigator.

Figure 21: zc702_axis_io Block Diagram



Instead of live I/O from off-chip, this platform contains 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.

Like input streaming off of an analog-to-digital converters, this data stream is not packetized, so in the AXI4 stream there is no TLAST signal. This means that any SDSoC application that consumes the data stream must be capable of handling unpacketized streams. Within the SDx environment, all data mover IP cores other than `zero_copy` require packetized streams, so to consume streaming input from this platform, an application must employ direct connections to the AXI4-Stream port.

NOTE: *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.*

RECOMMENDED: *In this release of the SDx environment, all exported AXI and AXI4-Stream platform interfaces must run on the same "data motion" clock (*`dmclkid`*). If your platform I/O requires a clock that is not one of the exported SDx platform clocks, you can use the AXI4-Stream Data FIFO IP within the Vivado IP catalog for clock domain crossing.*

Generating the SDSoC Hardware Platform Description

As described in SDSoC Tcl Commands in Vivado, the hardware platform port interface is defined in the Vivado project, and is extracted by the SDSoC Platform Utility using Tcl commands which are contained in the script file `zc702_axis_io/src/vivado/zc702_axis_io_pfm.tcl`.

1. The following command creates a hardware platform object.

```
set pfm [sdsoc::create_pfm zc702_axis_io.hpfm]
```

2. The following commands declare the platform name and provide a brief description that will be displayed when a user executes '`sdscc -sds-pf-info zc702_axis_io`'.

```
sdsoc::pfm_name        $pfm "xilinx.com" "xd" "zc702_axis_io" "1.0"
sdsoc::pfm_description $pfm "Zynq ZC702 Board With Direct I/O"
```

3. The following commands declare the clocks; the default platform clock has id 1:. The 'true' argument indicates that this clock is the platform default.

```
sdsoc::pfm_clock       $pfm FCLK_CLK0 ps7 0 false  psr0
sdsoc::pfm_clock       $pfm FCLK_CLK1 ps7 1 true  psr1
```

TIP: *The command also specifies the associated `proc_sys_reset` IP instances (`psr0, psr1`) that are required of every platform clock.*

4. The following commands declare the platform AXI interfaces. Each AXI port requires a "memory type" declaration, which must be one of `{M_AXI_GP, S_AXI_ACP, S_AXI_HP, MIG}`, i.e., a general purpose AXI master, a cache coherent slave interface, a high-performance port or an interface to an external memory controller respectively. The choice of AXI ports is up to the platform creator. Note that although this platform declares both general purpose masters, the coherent port, and all four high performance ports on the processing system IP block, the only requirement is that at least one general purpose AXI master and one AXI slave port must be declared.

```
sdsoc::pfm_axi_port    $pfm M_AXI_GP0 ps7 M_AXI_GP
sdsoc::pfm_axi_port    $pfm M_AXI_GP1 ps7 M_AXI_GP
sdsoc::pfm_axi_port    $pfm S_AXI_ACP ps7 S_AXI_ACP
sdsoc::pfm_axi_port    $pfm S_AXI_HP0 ps7 S_AXI_HP
sdsoc::pfm_axi_port    $pfm S_AXI_HP1 ps7 S_AXI_HP
sdsoc::pfm_axi_port    $pfm S_AXI_HP2 ps7 S_AXI_HP
sdsoc::pfm_axi_port    $pfm S_AXI_HP3 ps7 S_AXI_HP
```

5. The following command declares the `stream_fifo` master AXI4-Stream bus interface that proxies the direct I/O:

```
sdsoc::pfm_axis_port   $pfm M_AXIS stream_fifo M_AXIS
```

6. The following commands declare the interrupt inputs:

```
for {set i 0} {$i < 16} {incr i} {
  sdsoc::pfm_irq       $pfm In$i irq_concat
}
```

7. Having declared all of the interfaces, the following command generates the platform hardware metadata file `zc702_axis_io.hpfm` when the platform is generated by the SDSoC Platform Utility.

```
sdsoc::generate_hw_pfm $pfm
```

You can validate the platform hardware description file for a platform with the following command.

```
sds-pf-check <sdx_root>/samples/platforms/zc702_axis_io/hw/
zc702_axis_io.hpfm
```

You should see a message such as *<sdx_root>/samples/platforms/zc702_axis_io/hw/zc702_axi_io_hw.pfm validates*.

SDSoC Platform Software Libraries

Every platform IP that exports a AXI4 Stream interface must have hardware functions packaged in a C-callable library that an application can invoke to connect accelerators to the exported interface. You can use the SDSoC `sdslib` utility to create a static C-callable library for the platform as described in Creating a Library.

1. The source code for the platform can be found in `the <sdx_root>/samples/platforms/zc702_axis_io/src` directory.

   The platform AXI4-Stream Data FIFO IP requires a C-callable function to access its `M_AXIS` port, which in this case will be called `pf_read_stream`. The hardware function is defined in `pf_read.cpp`, as follows. The function declaration is included in the file, `zc702_axis_io.h`.

```
void pf_read_stream(unsigned *rbuf) {}
```

   The function body is empty; when called from an application, the `sdscc` compiler fills in the stub function body with the appropriate code to move data. Note that multiple functions can map to a single IP, as long as the function arguments all map onto the IP ports, and do so consistently; for example, two array arguments of different sizes cannot map onto a single AXIS port on the corresponding IP.

2. For each function in the C-callable interface, you must provide a mapping from the function arguments to the IP ports. The mappings for the `pf_read_stream` IP is captured in `zc702_axis_io.fcnmap.xml`.

```
<xd:repository xmlns:xd="http://www.xilinx.com/xd">
    <xd:fcnMap xd:fcnName="pf_read_stream"
xd:componentRef="zc702_axis_io">
        <xd:arg
            xd:name="rbuf"
            xd:direction="out"
            xd:busInterfaceRef="stream_fifo_M_AXIS"
            xd:portInterfaceType="axis"
            xd:dataWidth="32"
        />
```

```
   </xd:fcnMap>
</xd:repository>
```

Each function argument requires name, direction, IP bus interface name, interface type, and data width.

IMPORTANT: *The `fcnMap` associates the platform function `pf_read_stream` with the platform bus interface `stream_fifo_M_AXIS` on the platform component `zc702_axis_io`, which is a reference to a bus interface on an IP within the platform that implements the function. In `zc702_axis_io.hpfm` the platform bus interface ("port") named `stream_fifo_M_AXIS` contains the mapping to the IP in the `xd:instanceRef` attribute.*

---

3. IP customization parameters must be set at compile time in an XML file. In this example, the platform IP has no parameters, so the file `zc702_axis_io.params.xml` is particularly simple. To see a more interesting example, open `<sdx_root>/samples/fir_lib/build/fir_compiler.params.xml` in the SDx install tree.
4. The `src/linux` directory contains a makefile to build the library with the following commands.

```
sdslib -lib libzc702_axis_io.a \
 pf_read_stream pf_read.cpp \
 -vlnv xilinx.com:ip:axis_data_fifo:1.1 \
 -ip-map zc702_axis_io.fcnmap.xml \
 -ip-params zc702_axis_io.params.xml
```

The library file is stored in `zc702_axis_io/sw/aarch32-linux/lib/libzc702_axis_io.a`.

Platform Sample Designs

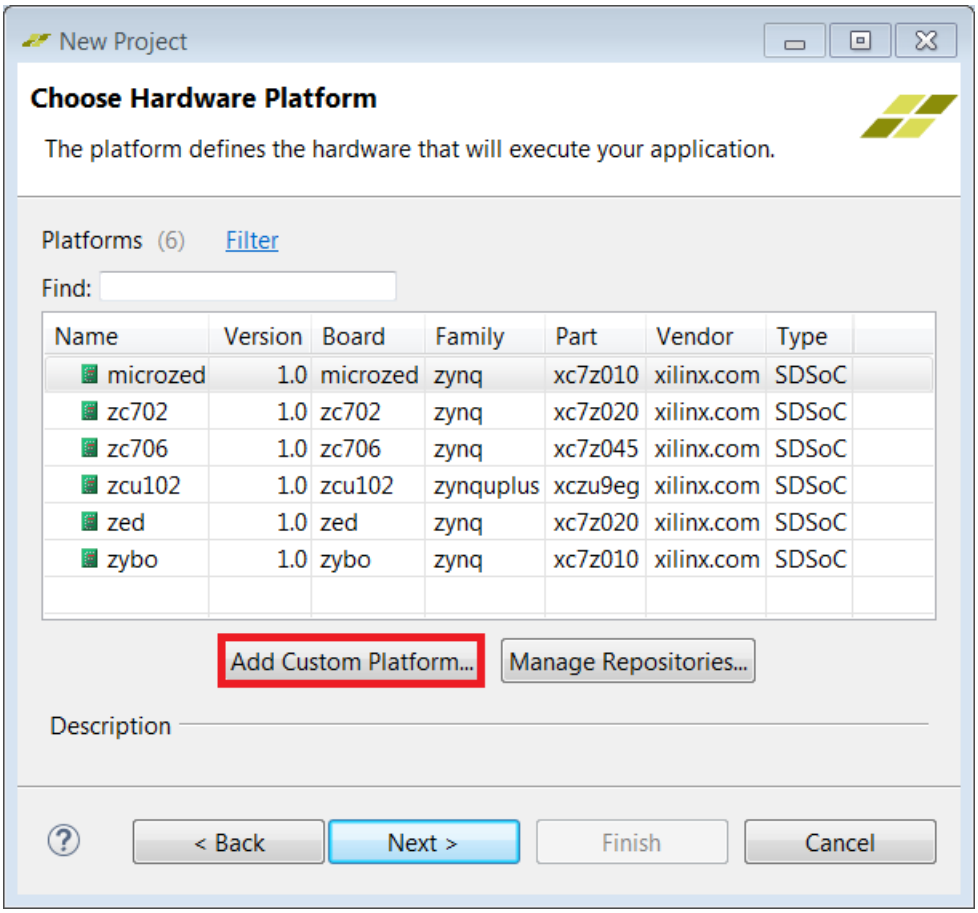
An SDSoC platform can include sample applications that demonstrate its use. The SDx IDE looks for a file called `samples/template.xml` for information on where the sample application source files reside within the platform. The template file for the `zc702_axis_io` platform lists several test applications, each of which is of specific interest.

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

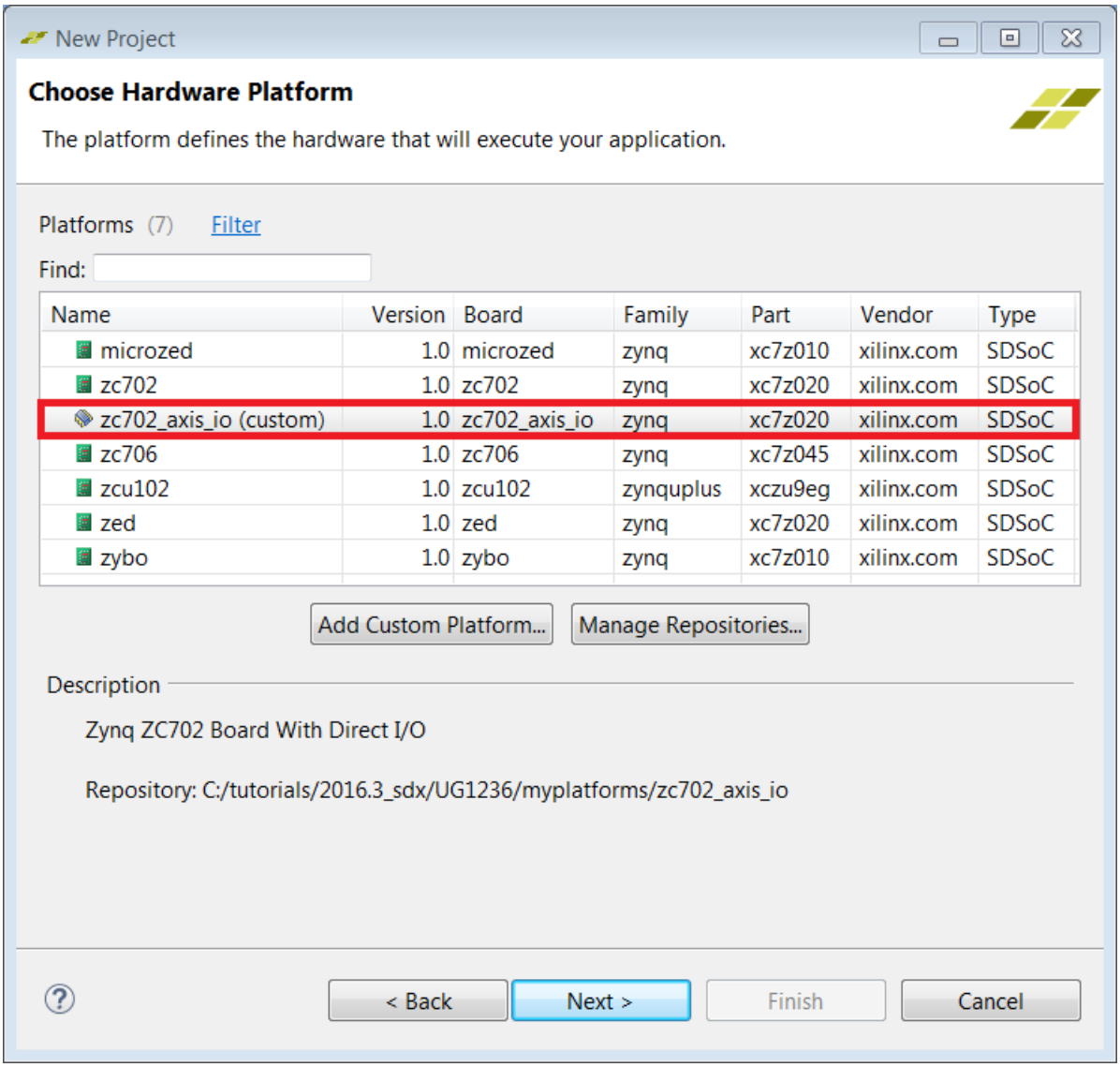
1. Launch Xilinx SDx and provide a path to your workspace such as `<path_to_tutorial>/myplatforms/`.

2. Create a new project by selecting **File → New → Xilinx SDx Project**.
3. Specify a project name in the Create New SDx Project page such as `my_zc702_axis_io`, and click **Next**.
4. In the Choose Hardware Platform page click **Add Custom Platform**.



5. Navigate to the folder containing the platform `<sdx_root>/samples/platforms/zc702_axis_io`.

6. The platform will show up in the Choose Hardware Platform Page. Select `zc702_axis_io (custom)` and click **Next**.



7. On the Choose Software Platform and Target CPU, keep the default **Linux SMP (Zynq 7000)** for System Configuration and click **Next**.
8. To test the platform with one of the sample applications, in the Templates page, select `Unpacketized AXI4-Stream to DDR` and click **Finish**. The `s2mm_data_copy` function is pre-selected for hardware. 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.

9. Open up `main.cpp`. Key points to observe are:
   - The ways in which buffers are allocated using `sds_alloc` to guarantee physically contiguous allocation required for the zero_copy datamover.

```
unsigned *bufs[NUM_BUFFERS];
bool error = false;
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_wrapper(bufs[0]);
for(int i=0; i<NUM_BUFFERS; i++) {
     s2mm_data_copy_wrapper(bufs[i]);
}
```

   - The way that the platform functions are invoked to read from platform input.

```
void copy_data_wrapper(unsigned int *buf)
void s2mm_data_copy_wrapper(unsigned *buf)
{
    unsigned rbuf0[1];
    pf_read_stream(rbuf0);
    s2mm_data_copy(rbuf0,buf);
}
```

10. 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.
11. 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

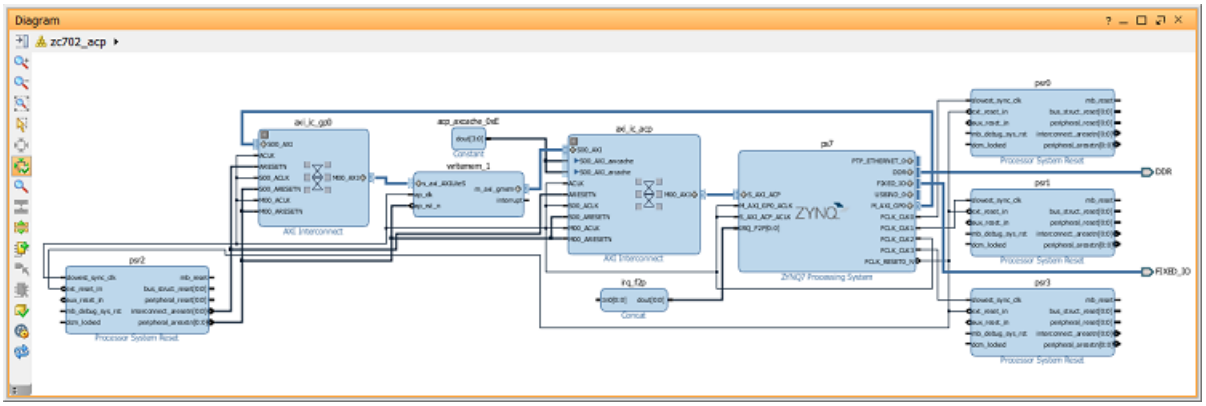
To share an AXI master (slave) interface between a platform IP and the accelerator and data motion IPs generated by the SDSoC compilers, you employ the SDSoC Tcl API to declare the first unused AXI master (slave) port (in index order) on the AXI interconnect IP block connected to the shared interface.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

SDSoC Platform Hardware Description

The hardware system is contained in the Vivado project `<sdx_root>/samples/sdspfm/zc702_acp/src/vivado/zc702_acp.xpr`.

1. The block diagram looks similar to the following figure.

Figure 22: ZC702_acp Block Diagram



2. As described in SDSoC Tcl Commands in Vivado, the hardware platform port interface is defined in the Vivado project, and is extracted by the SDSoC Platform Utility using Tcl commands which are contained in the script file `src/vivado/zc702_acp_pfm.tcl`. The following commands create a hardware platform object, give the platform a name and provide a brief description.

```
set pfm [sdsoc::create_pfm zc702_acp.hpfm]
sdsoc::pfm_name        $pfm "xilinx.com" "xd" "zc702_acp" "1.0"
sdsoc::pfm_description $pfm "Zynq XC702 platform with shared GP and ACP
ports"
```

3. This command declares the default platform clock to have id 2. The 'true' argument indicates that this clock is the platform default.

```
sdsoc::pfm_clock       $pfm FCLK_CLK2 ps7 2 true  psr2
```

TIP: *The command also specifies the associated `proc_sys_reset` IP instance (`psr2`) that is required of every platform clock as discussed in Declaring Clocks.*

4. The following commands declare the platform AXI interfaces:

```
sdsoc::pfm_axi_port    $pfm M_AXI_GP0 ps7 M_AXI_GP
sdsoc::pfm_axi_port    $pfm S_AXI_HP0 ps7 S_AXI_HP
sdsoc::pfm_axi_port    $pfm S_AXI_HP1 ps7 S_AXI_HP
sdsoc::pfm_axi_port    $pfm S_AXI_HP2 ps7 S_AXI_HP
sdsoc::pfm_axi_port    $pfm S_AXI_HP3 ps7 S_AXI_HP
for {set i 1} {$i < 64} {incr i} {
  sdsoc::pfm_axi_port  $pfm M[format %02d $i]_AXI axi_ic_gp0 M_AXI_GP
}
for {set i 1} {$i < 8} {incr i} {
```

```
  sdsoc::pfm_axi_port  $pfm S[format %02d $i]_AXI axi_ic_acp S_AXI_ACP
}
```

Each AXI port requires a "memory type" declaration, which must be one of `{M_AXI_GP, S_AXI_ACP, S_AXI_HP, MIG}`, i.e., a general purpose AXI master, a cache coherent slave interface, a high-performance port or an interface to an external memory controller, respectively.

The `for` loop with API calls to the `Mxy_AXI` (`y > 0`) ports declares available master ports on the interconnect attached to the `M_AXI_GP0` port of the processing system, and the loop with API calls to the `Sxy_AXI` (`y > 0`) ports declares available slave ports on the interconnect attached to the `S_AXI_ACP` port of the processing system.

Also observe in the Vivado block diagram shown above that the platform uses the least significant indexed ports on each of the interconnects within the platform as required.

5. The following commands declare the interrupt inputs:

```
for {set i 0} {$i < 16} {incr i} {
  sdsoc::pfm_irq       $pfm In$i irq_f2p
}
```

6. The following command creates the `zc702_acp.hpfm` hardware platform metadata file when the platform is generated by the SDSoC Platform Utility.

```
sdsoc::generate_hw_pfm $pfm
```

When this platform is used, the `sdscc` compiler will expand the platform interconnects attached to the `M_AXI_GP0` and `S_AXI_ACP` ports as needed, which in effect share the CPU and DDR memory access between platform and SDSoC application logic.

XILINX
ALL PROGRAMMABLE

Appendix E

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 guide:

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. *ZC702 Evaluation Board for the Zynq-7000 XC7Z020 All Programmable SoC User Guide* (UG850)
9. *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)
10. *PetaLinux Tools Documentation: Workflow Tutorial* (UG1156)
11. Vivado® Design Suite Documentation
12. *Vivado Design Suite User Guide: Creating and Packaging Custom IP* (UG1118)

Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at www.xilinx.com/legal.htm#tos; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at www.xilinx.com/legal.htm#tos.

AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.