SDSoC Environment Platform Development Guide

UG1146 (v2016.4) March 9, 2017

XILINX
ALL PROGRAMMABLE™

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
03/09/2017	2016.4	Added Appendix B: SDSoC Platform Examples. Minor edits for SDx™ IDE 2016.4.
11/30/2016	2016.3	Initial documentation release for SDx IDE 2016.3, which includes both the SDSoC™ Environment and the SDAccel™ Environment. Due to this major change in tool architecture, this document has undergone substantial changes in structure and content since the previous release.

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 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.

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.

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.

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

XILINX
ALL PROGRAMMABLE™

Chapter 2

SDSoC Platforms

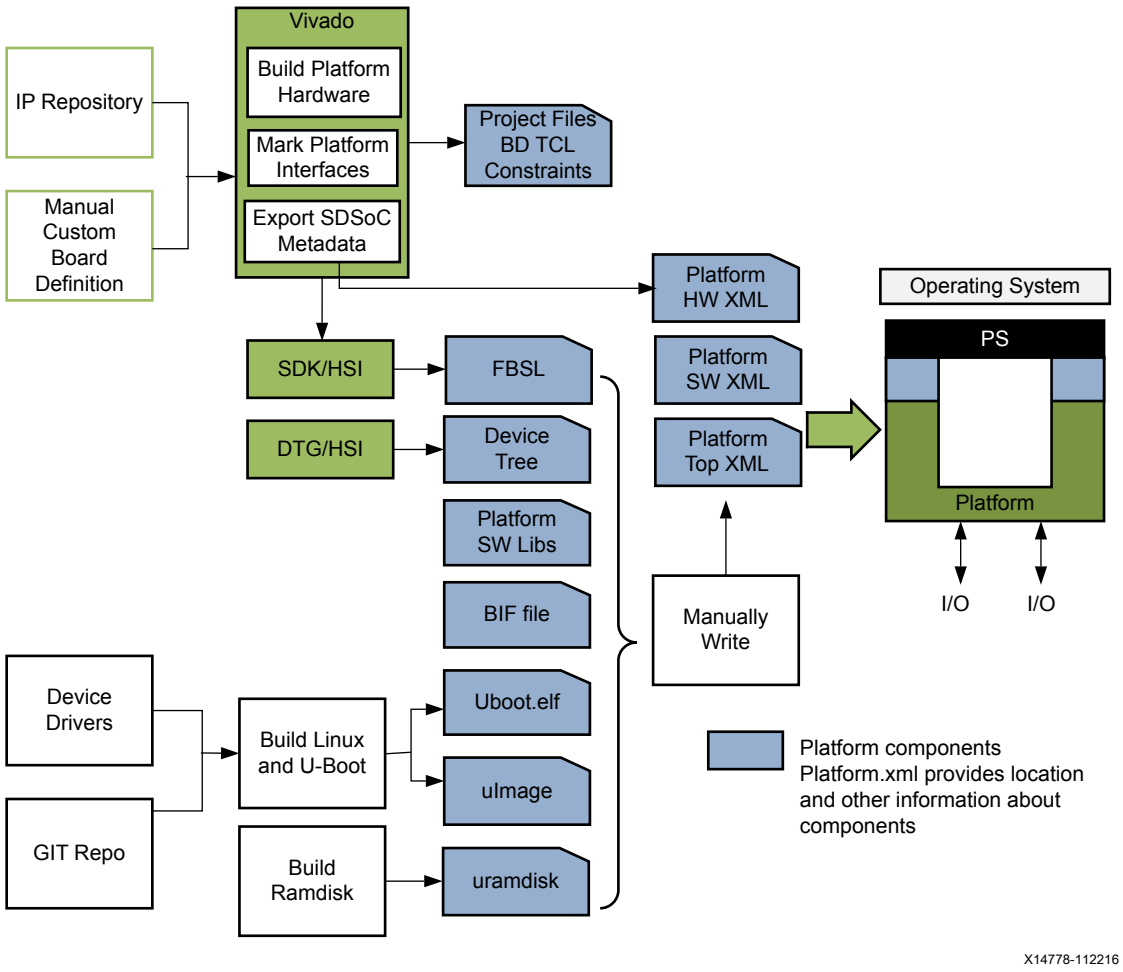
Introduction

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 XML metadata files that describe the hardware and software interfaces used by the SDSoC compilers to target the platform.

A platform provider designs the platform hardware using the Vivado Design Suite and IP Integrator. After the hardware has been built and verified, the platform provider executes Tcl commands within the Vivado tools to specify SDSoC platform hardware interfaces and generate the SDSoC platform hardware metadata file.

The platform creator 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.

Figure 1: Primary Components of an SDSoC Platform

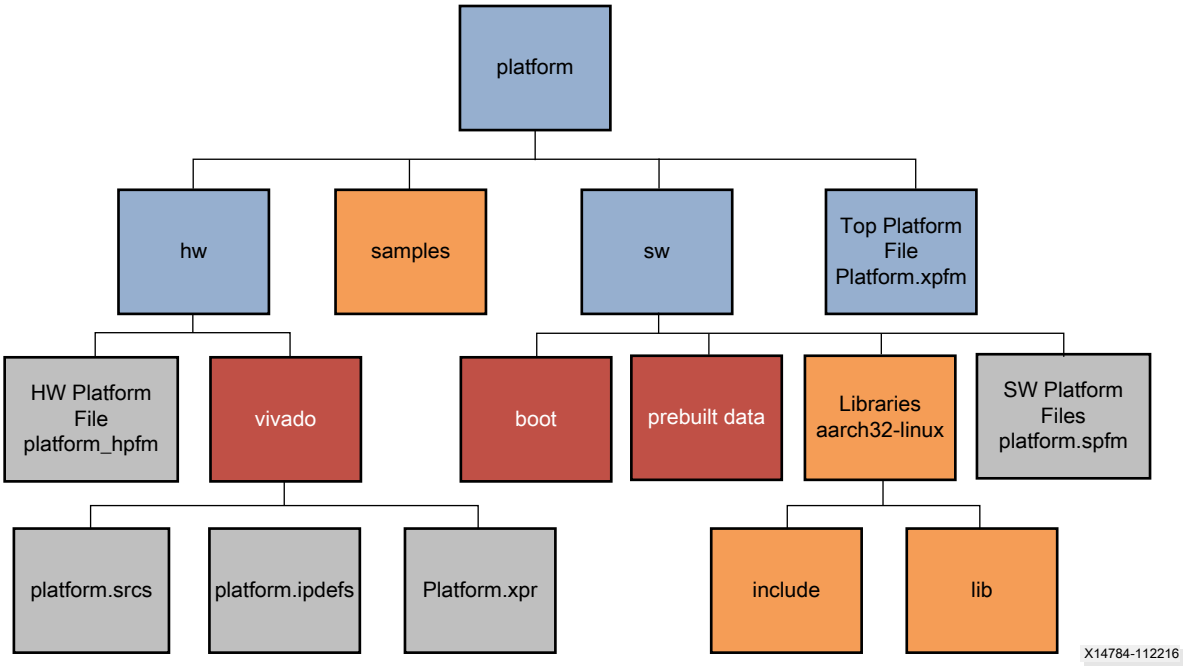


An SDSoC platform consists of the following elements:

- Metadata files
  - Platform top-level XML description file (`<platform>.xpfm`) written by hand.
  - Platform hardware XML description file (`<platform>.hpfm`) generated using Vivado tools
  - Platform software XML description file (`<platform>.spfm`) written by hand
- Vivado Design Suite project
  - Sources
  - Constraints
  - 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. For information on hardware platform creation and guidelines, see Hardware Platform Creation. For information on software platform components, see Software Platform Data Creation.

Platform GUI/Command Line Usage

When creating an SDSoC design, the user specifies an SDSoC platform and a system configuration that captures the software environment running on the target device, including the operating system (OS). A platform supports one or more system configurations. For example the zc702 platform (for the Zynq-7000 All Programmable SoC ZC702 Evaluation Kit board) is included in the SDx installation and offers three configurations: Linux, Standalone BSP and FreeRTOS.

The platform folder contains hardware and software platform data, with platform XML metadata that defines the supported hardware interfaces and software environments (system configurations), along with the software components they use (boot files, Linux image, header files, library files, and others).

To specify the platform and system configuration for the SDSoC Compiler command line tools `sdscc` or `sds++`, use the `-sds-pf <platform>` and `-sds-sys-config <system_configuration>` options. Refer to the *SDSoC Environment User Guide* (UG1027) for more information on these commands and options.

The *<platform>* can be a platform name or the path to a platform folder. If `-sds-sys-config` is not specified, a platform defined default system configuration is used. An example command line is shown below:

```
sdscc -sds-pf zc702 -sds-sys-config linux -c main.c
```

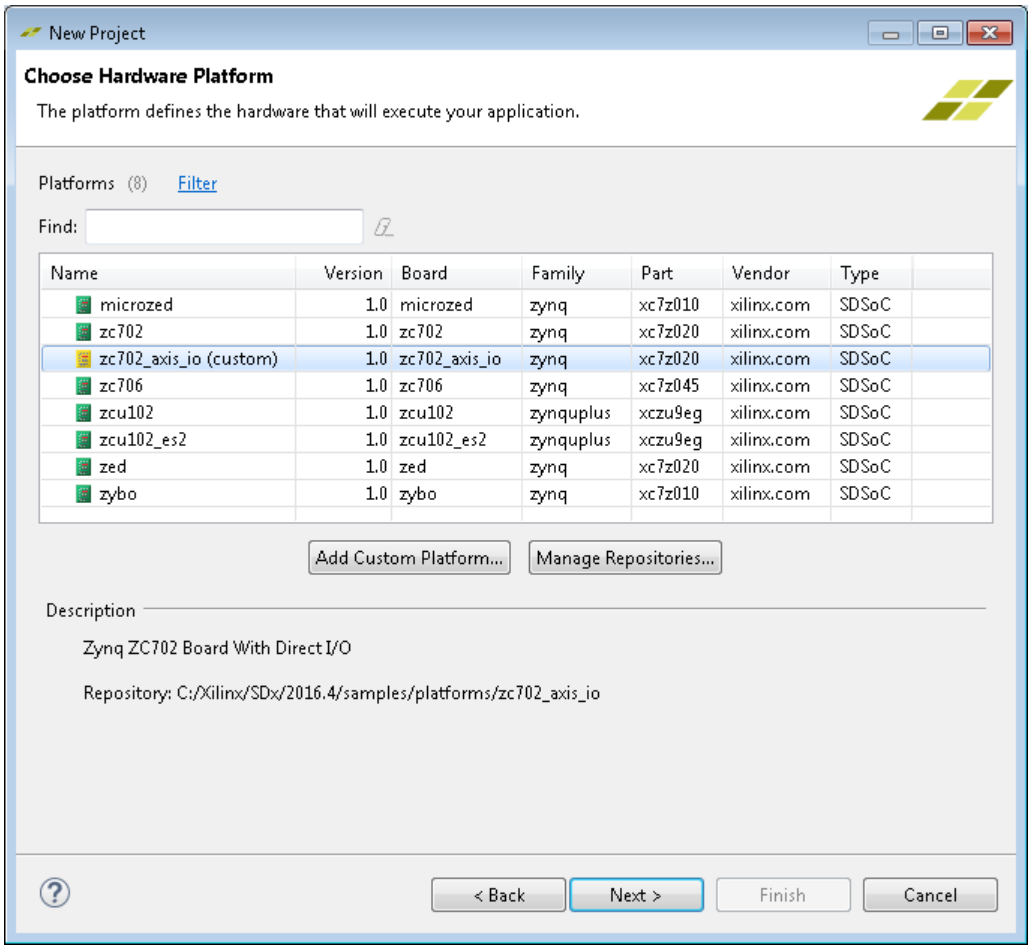
Command line arguments are available to display information about a platform. Use `-sds-pf-list` to list available factory platforms and `-sds-pf-info <platform>` to print information about the specified platform:

```
sdscc -sds-pf-list
sdscc -sds-pf-info zc702
```

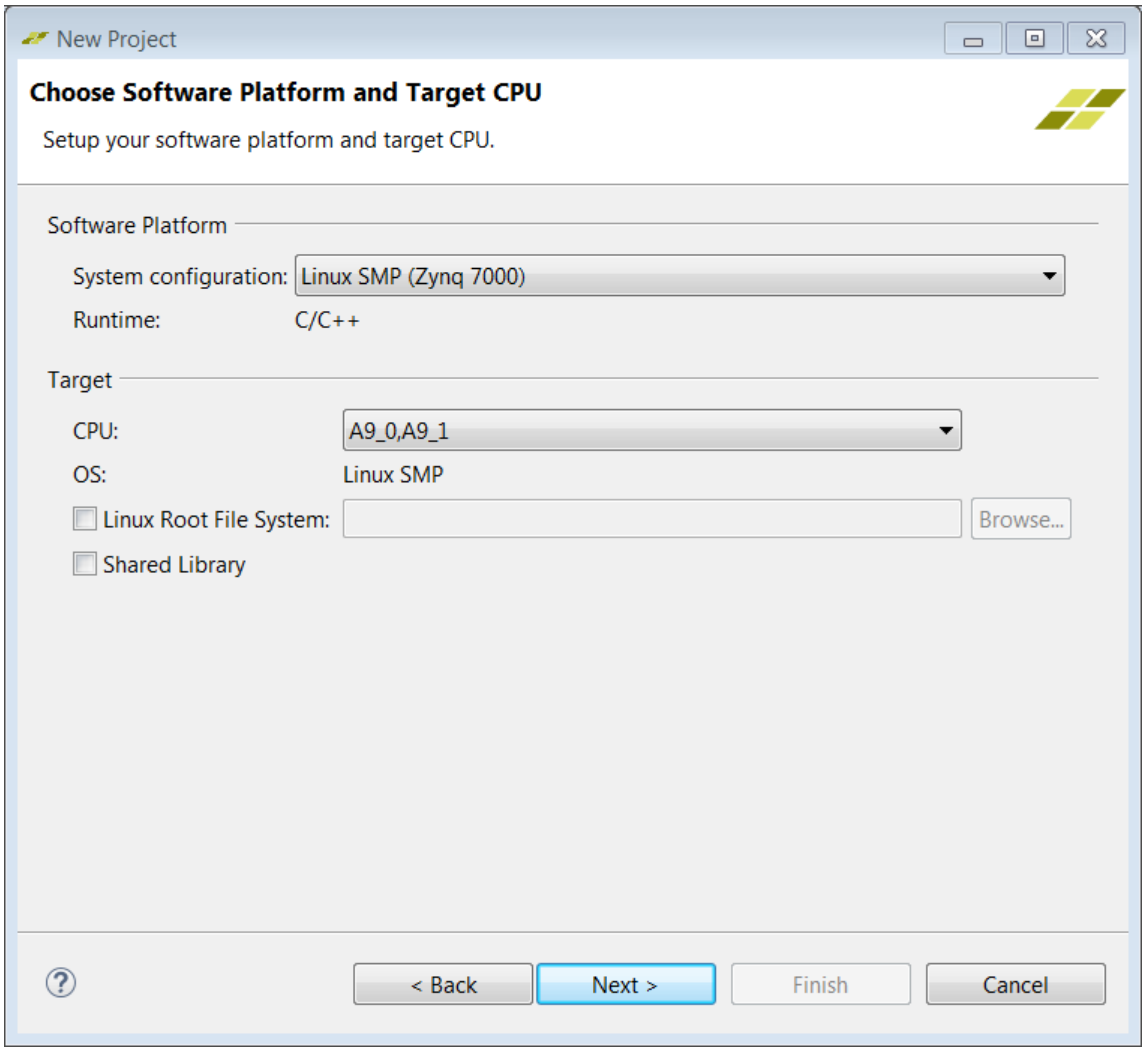
In the SDx Environment IDE, use **File > New > Xilinx SDx** to launch the New Project Wizard to create an application project. After entering a project name, the Choose Hardware Platform page appears, as shown in the figure below. Select a platform in the installation (factory platform), or click **Add Custom Platform** to navigate to and add a new platform folder. Xilinx provided platforms are found in `<SDx_install_path>/platforms`.

TIP: *You can also select a folder containing multiple platform folders to add multiple platforms to your repository in a single step.*

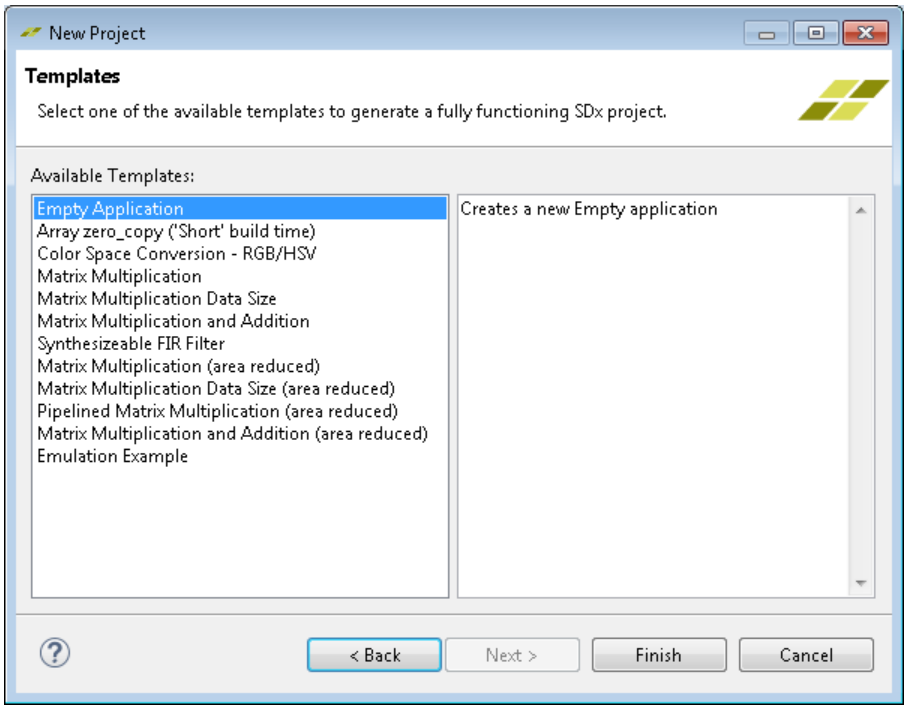
In the figure below, the **Add Custom Platform** command was used to add the `zc702_axis_io platform` in `<SDx_install_path>/samples/platforms/zc702_axis_io`.



After choosing a hardware platform, the Choose Software Platform and Target CPU page is displayed, as shown in the following figure, to let you define the system configuration. The values shown in the **System configuration** dropdown list, the **CPU** dropdown list, and the **OS** text field are defined in the software platform XML metadata file (`./sw/<platform>.spfm`) for the specified *`<platform>`*. The text strings used as command line arguments are defined by `sdx:name` attributes in the XML file, while GUI text strings are defined by `sdx:displayName` attributes. See Software Platform XML Metadata Reference for information on creating these files, which are created by the platform provider.



After choosing the software platform system configuration, the Templates page is displayed as shown below. On the Templates page, you can select a platform sample application if the platform folder contains a `samples` folder and `samples/template.xml` file which describes the available template applications.



When you click **Finish** on the Templates page, the SDx Environment IDE creates the application project, copying source files from the platform folder and automatically generating a Make file. If your platform does not provide sample applications, only the Empty Application will be shown and you can manually copy source files into the project's source folder or create source files using the source editor view.

Platform Directory Structure

The SDSoC factory platforms are found in `<path_to_install>/SDx/2016.x/platforms`, which contains sub-folders for each of the platforms. For example, the `zc702` platform files for the Zynq 7000 All Programmable SoC ZC702 Evaluation Kit board are shown below, details of the files are omitted:

- <path_to_install>/SDx/2016.x/platforms
  - zc702
    - hw
      - vivado
      - zc702.hpfm
    - sw
      - aarch32-none
      - boot
      - freertos
      - image
      - prebuilt_platform
      - qemu
      - zc702.spfm
    - zc702.xpfm

The `zc702.xpfm` file contains references to the hardware (`zc702.hpfm`) and software (`zc702.spfm`) metadata files. All path references in the `.spfm` file are relative to the folder containing that file.

The installation folder `<path_to_install>/SDx/2016.x/samples/platforms` contains a few SDSoC example platforms that are documented in SDSoC Platform Examples. For example, the `zc702_axis_io` platform as shown below contains additional folders and files that are not found in the default `zc702` platform:

- <path_to_install>/SDx/2016.x/samples/platforms
  - zc702_axis_io
    - hw
      - vivado
      - zc702_axis_io.hpfm
    - samples
      - aximm
      - pull_packet
      - stream
      - template.xml
    - src
    - sw
      - aarch32-linux
      - aarch32-none
      - boot
      - image
      - zc702.spfm
    - readme
    - zc702_axis_io.xpfm

TIP: *Aside from the basic directory structure and XML metadata file naming conventions, folder and file names of the platform are largely left to the discretion of the platform developer.*

Metadata Files

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

- Top-Level Platform XML file (.xpfm)
- Platform hardware description file (.hpfm)
- Platform software description file (.spfm)

Top-Level Platform XML File (.xpfm)

The top-level platform XML file is found in `<platform>/<platform>.xpfm`, and contains references to the hardware and software XML files and the folders that contain them. To create an `.xpfm` file for your platform, you may copy an existing `.xpfm` file and change instances of the platform name and file paths as needed.

Shown below is the file `<install>/platforms/zc702/zc702.xpfm`. It includes references to the hardware (`.hpfm`) and software (`.spfm`) platform XML metadata files found in `<install>/platforms/zc702/hw/zc702.hpfm` and `<install>/platforms/zc702/sw/zc702.spfm`, respectively.

IMPORTANT: *Note that the hardware and software platform data paths are relative to the top-level platform directory.*

```
<?xml version="1.0" encoding="UTF-8"?>
<sdx:platform sdx:vendor="xilinx.com"
              sdx:library="sdx"
              sdx:name="zc702"
              sdx:version="1.0"
              sdx:schemaVersion="1.0"
              xmlns:sdx="http://www.xilinx.com/sdx">
  <sdx:description>Basic platform targeting the ZC702 board, which includes
1GB of DDR3,
  16MB Quad-SPI Flash and an SDIO card interface. More information is
available at
  https://www.xilinx.com/products/boards-and-kits/ek-z7-zc702-g.html
  </sdx:description>

  <!-- Support multiple DSAs for multi-FPGA platforms -->
  <sdx:hardwarePlatforms>
    <sdx:hardwarePlatform sdx:path="hw" sdx:name="zc702.hpfm"/>
  </sdx:hardwarePlatforms>

  <sdx:softwarePlatforms>
    <sdx:softwarePlatform sdx:path="sw" sdx:name="zc702.spfm"/>
  </sdx:softwarePlatforms>
</sdx:platform>
```

Hardware Platform XML File (.hpfm)

The hardware platform XML metadata file is found in `<platform>/hw/<platform>.hpfm`, 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.

A portion of the zc702 hardware platform XML file, `<install>/platforms/zc702/hw/zc702.hpfm`, is shown below with clock information, including clocks available for the data motion network and accelerators.

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<!-- zc702.hpfm -->
<xd:repository xmlns:xd="http://www.xilinx.com/xd" xd:name="zc702"
xd:library="xd"
 xd:version="1.0" xd:vendor="xilinx.com">
  <xd:component xd:name="zc702" xd:library="xd" xd:version="1.0"
xd:vendor="xilinx.com"
   xd:type="platform" xd:BRAM="140" xd:DSP="220" xd:FF="106400"
xd:LUT="53200">
    <xd:platformInfo>
      <xd:deviceInfo xd:name="xc7z020clg484-1" xd:architecture="zynq"
xd:device="xc7z020"
       xd:package="clg484" xd:speedGrade="-1"/>
      <xd:registeredDevices xd:kio="0" xd:uio="0"/>
      <sdx:description>Basic platform targeting the ZC702 board, which
includes 1GB of DDR3,
       16MB Quad-SPI Flash and an SDIO card interface. More information is
available at
       https://www.xilinx.com/products/boards-and-kits/ek-z7-zc702-g.html
      </sdx:description>
      <xd:systemClocks xd:defaultClock="2">
        <xd:clock xd:name="CPU" xd:instanceRef="ps7"
xd:componentRef="processing_system7"
         xd:frequency="666.666687" xd:period="1.500000"
xd:status="reserved"/>
        <xd:clock xd:name="ps7_FCLK_CLK0" xd:instanceRef="ps7"
xd:componentRef="processing_system7"
         xd:id="0" xd:frequency="166.666672" xd:period="6.000000"
xd:normalizedPeriod="4.000000"
         xd:status="changeable"/>
        <xd:clock xd:name="ps7_FCLK_CLK1" xd:instanceRef="ps7"
xd:componentRef="processing_system7"
         xd:id="1" xd:frequency="142.857132" xd:period="7.000001"
xd:normalizedPeriod="4.666667"
         xd:status="changeable"/>
        <xd:clock xd:name="ps7_FCLK_CLK2" xd:instanceRef="ps7"
xd:componentRef="processing_system7"
         xd:id="2" xd:frequency="100.000000" xd:period="10.000000"
xd:normalizedPeriod="6.666667"
         xd:status="changeable"/>
        <xd:clock xd:name="ps7_FCLK_CLK3" xd:instanceRef="ps7"
xd:componentRef="processing_system7"
         xd:id="3" xd:frequency="200.000000" xd:period="5.000000"
xd:normalizedPeriod="3.333333"
         xd:status="changeable"/>
      </xd:systemClocks>
    </xd:platformInfo>
```

For a complete discussion on the hardware platform XML file, see Hardware Platform Metadata Creation.

Software Platform XML File (.spfm)

The software platform XML file is found in `<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.

Using the metadata in the `.spfm` file, the SDSoC tools know where to find boot files (first stage boot loader or FSBL) and Linux images used to generate an SD card image. If the platform includes optional header and library files, SDSoC automatically adds paths to the files when compiling and linking the user's application. Optional prebuilt hardware bitstreams reduce run times when building applications without hardware accelerators.

The SDx Environment IDE uses metadata in the `.spfm` file to create lists of system configurations, define default selections, and create labels for GUI elements in the New Project Wizard.

A portion of the zc702 software platform XML file, `<install>/platforms/zc702/sw/zc702.spfm`, is shown below to provide an overview of the schema, including metadata used in the command line and GUI.

```
<?xml version="1.0" encoding="UTF-8"?>
<sdx:platform sdx:vendor="xilinx.com"
              sdx:library="sdx"
              sdx:name="zc702"
              sdx:version="1.0"
              sdx:schemaVersion="1.0"
              xmlns:sdx="http://www.xilinx.com/sdx">
      <sdx:description>Basic platform targeting the ZC702 board, which
includes 1GB of DDR3,
       16MB Quad-SPI Flash and an SDIO card interface. More information is
available at
       https://www.xilinx.com/products/boards-and-kits/ek-z7-zc702-g.html
      </sdx:description>
    <sdx:systemConfigurations sdx:defaultConfiguration="linux">
        <sdx:configuration sdx:name="linux"
                           sdx:displayName="Linux SMP (Zynq 7000)"
                           sdx:defaultProcessorGroup="a9_0">
            <sdx:description>Linux SMP running on Zynq
7000</sdx:description>
            <sdx:prebuilt sdx:data="prebuilt_platform"/>
            <sdx:bootImages sdx:default="standard">
                <sdx:image sdx:name="standard"
                           sdx:bif="boot/linux.bif"
                           sdx:imageData="image"
                           sdx:mountPath="/mnt"
                           sdx:readme="boot/generic.readme"
                           sdx:qemuDevicetree="qemu/devicetree.dtb"
                           sdx:qemuBoot="boot/u-boot.elf"
                           sdx:qemuArguments="qemu/qemu_args.txt"
                />
            </sdx:bootImages>
            <sdx:processorGroup sdx:name="a9_0"
                                sdx:displayName="A9_0,A9_1"
```

```
                                sdx:cpuType="cortex-a9">
                <sdx:os sdx:name="linux"
                        sdx:displayName="Linux SMP"
                />
            </sdx:processorGroup>
        </sdx:configuration>
```

For a complete description of the software platform XML metadata file, see Software Platform Metadata Creation.

Platform Checklist

The overview of the platform creation process in this chapter touched 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.

This guide assumes familiarity with the Vivado Design Suite and its use in creating Zynq-7000 or Zynq UltraScale+ MPSoC designs, SDSoC from the perspective of a user of platforms, Xilinx software development tools such as the Xilinx Software Development Kit (SDK), and embedded software environments (Linux or bare-metal). It provides greater depth on topics that are SDSoC-specific, for example platform hardware requirements and platform metadata XML files, with general guidance or references to material that you may already be familiar with.

If you are new to the SDSoC platform creation process, read the introductions in the remaining 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 created SDSoC platforms in the past, the process remains largely the same: 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>
         - hw
           - vivado
           - <platform>.hpfm
         - sw
           -
         - samples
         - <platform>.xpfm
   - For more information on creating the XML metadata files refer to Hardware Platform Metadata Creation and Software Platform Metadata Creation.
5. Validate your platform supports the SDSoC environment.
   - The Introduction of this chapter describes a platform checklist with 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.
    - See Platform GUI/Command Line Usage for an overview of this process.

XILINX
ALL PROGRAMMABLE

Chapter 3

Hardware Platform Creation

Introduction

The hardware platform creation process consists of building a Vivado design and generating the hardware platform metadata XML file (.hpfm) that describes the hardware interfaces supported, including clocks, interrupts, and bus interfaces. The platform folder `<path_to_platform>/hw/vivado` contains the Vivado project, while the hardware platform metadata XML file is found in `<path_to_platform>/hw/<platform>.hpfm`.

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 platform hardware folder.

The creation of the hardware platform metadata XML (`.hpfm`) is described in detail in Hardware Platform Metadata Creation.

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.
2. Create a Tcl script that uses the SDSoC Vivado TCL APIs.
3. Execute the Tcl script in the Vivado Tcl Console to create the hardware platform metadata XML file (.hpfm). The metadata captures platform clocks, AXI and AXI4-Stream bus interfaces, interrupts and other hardware information.

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 platform AXI4-Stream interface requires `TLAST` and `TKEEP` sideband signals to comply with the Vivado tools data mover IP used by the SDSoC environment.
- 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 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.

File name and location: `<path_to_install>/SDx/2016.x/platforms/<platform>/hw/vivado/<platform>.xpr`

Example: `<path_to_install>/SDx/2016.x/platforms/zc702/hw/vivado/zc702.xpr`

IMPORTANT: *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; the Vivado tools manage internal 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 Vivado command* ***File→Archive Project*** *to create a zip archive. Copy and unzip this archive file into the SDSoC platform directory where the hardware platform resides.*

The Vivado tool requires you to ***Upgrade IP*** *for every new version of the Vivado Design Suite. To migrate an SDSoC hardware platform from a prior release, open the project in the new version of the Vivado tools, and upgrade the block design and all IP. 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.*

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.

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 a text or XML editor to create the software platform metadata XML file. The creation of the software platform metadata XML (`.spfm`) is described in Software Platform Metadata Creation.

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-v2016.4-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. The platform can include the libraries and header files using the procedure defined in Software Platform XML File (.spfm) and Software Platform Metadata Creation.

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. Refer to Software Platform XML Metadata Reference for more information.

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. For example

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

in a platform software description file 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 hatrdware 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/fsbl.elf` 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-7000 or Zynq-UltraScale+ MPSoC designs.

Before starting, you should complete the following:

- Set up your shell environment with PetaLinux 2016.3 tools in your PATH environment variable.
- 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 sdx_release_tag xilinx-v2016.4-sdsoc
  ```

- 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.

The steps below include basic setup, loading the hardware handoff file, kernel configuration, root file system configuration, and building the Linux image. 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. Set up the working directory by running the following commands:
   - `mkdir components/linux-kernel/`
   - `ln -s <path_to_linux_checkout> components/linux-kernel/sdsoc`

2. Configure the hardware description:
   - `petalinux-config --get-hw-description=<PATH TO HDF>`
   - Apply the settings in the configuration menu as shown below:

```
linux Components Selection
     kernel [sdsoc]
Kernel Bootargs
     generate boot args automatically [OFF]
     <Zynq> user set kernel bootargs [console=ttyPS0,115200
earlyprintk quiet]
     <MPSoC> user set kernel bootargs
[earlycon=cdns,mmio,0xFF000000,115200n8 console=ttyPS0,115200 quiet]
```

3. Configure the kernel:
   - `petalinux-config -c kernel`
   - Apply the settings in the configuration menu as shown below:

```
Device Drivers
      Generic Driver Options
           <Zynq only> Size in Mega Bytes [256]
           <MPSoC only> Size in Mega Bytes [1024]
      Staging drivers [ON]
            Xilinx APF Accelerator driver [ON]
            Xilinx APF DMA engines support [ON]
```

4. Configure the root file system:
   - `petalinux-config -c rootfs`
   - Apply the settings in the configuration menu as shown below:

```
Filesystem Packages
      base
           gcc-runtime-xilinx
                   libstdc++6 [ON]
           tcf-agent
                   tcf-agent-lic [ON]
                   tcf-agent [ON]
      console/network
             dropbear
                     dropbear-openssh-sftp-server [OFF]
Apps
      <everything> [OFF]
```

5. Update the device tree:
   - Append the text below to the end of `subsystems/linux/configs/system-top.dts`:

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

6. Update the U-boot configuration:
   - Append the text below to the end of `subsystems/linux/configs/u-boot/platform-top.h`:

```
#undef CONFIG_PREBOOT
#define CONFIG_PREBOOT "echo U-BOOT for ${hostname};setenv preboot;
echo"
```

7. Build the PetaLinux image:
   - `petalinux-build`

After the PetaLinux build completes, the `images/linux/` directory contains an `image.ub` file and a `u-boot.elf` file.

TIP: *For Zynq UltraScale+ MPSoC platforms, this directory also contains `bl31.elf`, which is a required ARM trusted firmware file necessary for booting the platform, and a pre-compiled power management unit firmware file, `pmufw.elf`, in `pre-built/linux/images/`.*

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/fsbl.elf` 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-SDx` script 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

A platform can optionally include sample application templates to demonstrate the usage of the platform. Sample applications must reside in the `samples` directory of a platform: `<platform_install>/samples`.

The file that describes the applications to SDx is called `template.xml` and it resides inside the `samples` directory. The `template.xml` file uses a very simple format. Here is an example for the zc702_led sample platform found in `<SDx_install>/samples/platforms/zc702_led/samples/template.xml`.

```
<?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="arraycopy" name="Array copy"
            description="Simple test application">
    <supports>
      <and>
        <os name="Linux"/>
      </and>
    </supports>
    <accelerator name="arraycopy" location="arraycopy.cpp"/>
  </template>
  <template location="arraycopy_sa" name="Array copy"
            description="Simple test application">
    <supports>
      <and>
        <os name="Standalone"/>
      </and>
    </supports>
    <accelerator name="arraycopy" location="arraycopy.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="myapp" name="My App" description="Sample application">
```

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 SDSoC 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 containingg 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"/>
```

Complete Example of template.xml

The following is a complete example of a `template.xml` file:

```
<?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="myapp" name="My App"
            description="Sample application">
    <supports>
      <and>
        <or>
          <os name="Linux"/>
          <os name="Standalone"/>
          <os name="FreeRTOS"/>
        </or>
      </and>
    </supports>
    <accelerator name="myaccel" location="myaccel.cpp"/>
    <includepaths>
      <path location="myinclude"/>
      <path location="dir/include"/>
    </includepaths>
    <libraries>
      <lib name="mylib1"/>
      <lib name="mylib2"/>
    </libraries>
    <exclude>
      <directory name="MyDir"/>
      <directory name="MyOtherDir"/>
      <file name="MyFile.txt"/>
      <file name="MyOtherFile.txt:/>
    </exclude>
  </template>
  <!-- Multiple template elements allowed -->
</manifest:Manifest>
```

XILINX
ALL PROGRAMMABLE

Chapter 6

Hardware Platform Metadata Creation

Introduction

Every SDSoC platform includes a hardware platform metadata XML file, `<platform>.hpfm`, containing information about its 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. This chapter describes the steps used to create the `.hpfm` file, the hardware platform Tcl API, and the Vivado commands used to load and run the Tcl script, Finally, an example of a complete hardware platform Tcl script is presented.

The `.hpfm` file is found in the folder `<path_to_platform>/hw`. For example, the hardware platform metadata XML file for the Zynq 7000 ZC702 Evaluation Kit board is found in `<SDx_install>/platforms/zc702/hw/zc702.hpfm`. An example Zynq UltraScale+ MPSoC hardware platform file can be found at `<SDx_install>/platforms/zcu102/hw/zcu102.hpfm`.

After you complete a hardware platform design project in the Vivado Design Suite, you can create a Tcl script that will generate the hardware platform description file from the Vivado project using the SDSoC Vivado Tcl commands as described in the following section. With your Vivado project loaded, execute the Tcl file commands to write out the `<platform>.hpfm` file.

SDSoC Vivado Tcl Commands

This section describes the Vivado® IP Integrator Tcl commands that specify the hardware interface of an SDSoC™ platform, which includes clock information and clock, reset, interrupt, AXI, and AXI4-Stream type interfaces. Once you have built and verified your hardware system within the Vivado Design Suite, the process of creating an SDSoC platform hardware description file consists of the following steps.

1. Load the SDSoC Vivado Tcl API by executing the following command in the Vivado Design Suite Tcl console.

```
source -notrace <sdx_install_path>/scripts/vivado/sdsoc_pfm.tcl
```

2. Execute Tcl APIs in Vivado to accomplish the following steps:
   a. Declare the hardware platform name.
   b. Define a brief description of the platform.
   c. Declare the platform clock ports.
   d. Declare the platform AXI bus interfaces.
   e. Declare the platform AXI4-Stream bus interfaces.
   f. Declare the available platform interrupts.
   g. Write the hardware platform description metadata file.

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"
```

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
```

Testing the Platform Hardware Description File

The SDx Environment includes an XML Schema for validating your platform hardware description file. For example, to validate your platform hardware description XML file in an SDx terminal, use this command:

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

After putting the hardware platform description file (`<platform>.hpfm`) and software platform description file (`<platform>.spfm`) in the platform directory, you can verify that the SDx IDE can read the files correctly by executing the following command, which lists all the available platforms:

```
> sdscc -sds-pf-list
```

If you see the platform you have created in the displayed list, then the SDx IDE has found it. To display more information about your platform, use the following command:

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

XILINX
ALL PROGRAMMABLE

Chapter 7

Software Platform Metadata Creation

Introduction

Each hardware platform you develop requires a software platform metadata file, `<platform>.spfm`, that contains information about supported software environments, referred to as system configurations. The software platform metadata file can be found in `<path_to_platform_folder>/sw/<platform>.spfm`. You must manually create the `.spfm` file using a text or XML editor. You can create a new SPFM file by editing an existing file, and modifying it to suit your hardware platform.

When building user applications, the SDSoC compiler needs to know any include paths and library paths located in the platform, and also needs to locate files required to build SD card images. The SDx environment IDE requires this information to populate GUI elements for selecting the system configuration. For example, the ZC702 platform included in the SDx environment installation supports system configurations for Linux, Standalone OS, and FreeRTOS OS, that target the Xilinx Zynq-7000 All Programmable SoC ZC702 Evaluation Kit board.

This chapter walks you through the contents of the ZC702 software platform metadata XML file, `zc702.spfm`, with annotations that describe elements of the file. A metadata reference section follows that presents each of the XML elements and attributes, and describes their use.

Example ZC702.spfm File

The `zc702.spfm` software platform XML file for the ZC702 platform is displayed below for reference. The complete file can be found in the SDx environment installation `<SDx_install_path>/platforms/zc702/sw/zc702.spfm`. Each of the system configurations and their metadata are described and annotated. Where paths are specified, they are relative to the folder containing the `.spfm` file (`<path_to_platform>/sw`) unless otherwise noted.

As shown below, the `zc702.spfm` begins with the `sdx:platform` tag.

The `sdx:description` should summarize the features of the hardware platform, and can be displayed by the SDSoC environment.

```
<?xml version="1.0" encoding="UTF-8"?>
          <sdx:platform sdx:vendor="xilinx.com"
              sdx:library="sdx"
              sdx:name="zc702"
```

```
                sdx:version="1.0"
                sdx:schemaVersion="1.0"
                xmlns:sdx="http://www.xilinx.com/sdx">
            <sdx:description>Basic platform targeting the ZC702 board, which
          includes 1GB of DDR3, 16MB Quad-SPI Flash and an SDIO card
interface. More information at
            http://www.xilinx.com/products/boards-and-kits/
EK-Z7-ZC702-G.htm
            </sdx:description>
```

The `zc702.spfm` contains three System Configurations in the `sdx:systemConfigurations` section, each described by a `sdx:configuration` section.

TIP: *You only need to specify one if that is all your platform supports.*

The `sdx:systemConfigurations` designates one of the `sdx:configuration` sections as the default to use by the SDx environment New Project Wizard, or as an sdscc/sds++ -sds-sys-config option when none is specified (`sdx:defaultConfiguration`).

The `sdx:configuration` includes a name used as the sdscc/sds++ -sds-sys-config option value (`sdx:name`), a string used in the SDx Environment IDE New Project Wizard System Configuration dropdown menus (`sdx:displayName`), and mouse over text for the System Configuration (`sdx:description`).

The `sdx:defaultProcessorGroup` defines the default Target CPU in the New Project Wizard dropdown menu, and as the sdscc/sds++ -sds-proc option value if not specified.

The `sdx:prebuilt` element defines a folder with a prebuilt bitstream that represents the hardware platform implemented without any accelerators. The `sdx:prebuilt` attribute `sdx:data` specifies the path to the folder, relative to the `.spfm` file, containing the prebuilt bitstream and project files used when SDSoC builds applications containing no hardware accelerators. The `sdx:prebuilt` entity is optional, but eliminates time spent building the bitstream when specified.

The `sdx:prebuilt` platform data is the same for all system configurations of a platform, since it represents the hardware platform implemented without accelerators, and the hardware is the same regardless of the OS that runs on it, as long as the platform supports the OS.

TIP: *For the zc702, `sdx:prebuilt` is defined separately for each system configuration, but they all point to the same folder.*

```
<sdx:systemConfigurations
          sdx:defaultConfiguration="linux">
        <sdx:configuration sdx:name="linux"
        sdx:displayName="Linux SMP (Zynq 7000)"
        sdx:defaultProcessorGroup="a9_0">
            <sdx:description>Linux SMP running on Zynq
7000</sdx:description>
            <sdx:prebuilt sdx:data="prebuilt_platform"/>
```

The `sdx:bootImages` metadata contains one or more `sdx:image` sections used by SDSoC when creating SD card images.

While `sdx:default` specifies the default image creation metadata, in this initial release only one `sdx:image` section is supported. The `sdx:image` must specify a .bif file used with the "bootgen" tool to create the SD card image (`sdx:bif` specifies the path to the .bif file relative to the "sw" folder).

The `sdx:imageData` attribute specifies a folder containing boot files to copy to the SD card root folder – since the System Configuration is for Linux, the image folder includes the kernel, ramdisk and device tree (as separate files or a .ub unified boot file).

The `sdx:qemu*` values are optional and used for SDSoC emulation flows.

The `sdx:mountPath` is the path to the SD card in the Linux file system once the system boots.

```
<sdx:bootImages sdx:default="standard">
                <sdx:image sdx:name="standard"
                           sdx:bif="boot/linux.bif"
                           sdx:imageData="image"
                           sdx:mountPath="/mnt"
                           sdx:readme="boot/
generic.readme"
                           sdx:qemuDevicetree="qemu/
devicetree.dtb"
                           sdx:qemuBoot="boot/
u-boot.elf"
                           sdx:qemuArguments="qemu/qemu_args.txt" />
                </sdx:bootImages>
```

The `sdx:processorGroup` includes a name used as the sdscc/sds++ -sds-sys-proc option value (`sdx:name`), a string used in the SDx Environment IDE New Project Wizard Target CPU dropdown menus (`sdx:displayName`), and mouse over text for the Target CPU (`sdx:description`).

The `sdx:cpuType` is cortex-a9 for Zynq, and cortex-a53 or cortex-r5 for Zynq UltraScale+.

The `sdx:os` includes a name and a string used in the SDx Environment IDE New Project Wizard OS label (`sdx:displayName`).

The `sdx:crossCompilerType` specifies whether to use the Linux or baremetal Linaro ARM cross-compiler and the `sdx:toolConfig` specifies an SDSoC file containing settings used to build the user application (include paths, library paths, compiler options, linker options, etc).

```
<sdx:processorGroup sdx:name="a9_0"
          sdx:displayName="A9_0,A9_1"
          sdx:cpuType="cortex-a9">
                <sdx:os sdx:name="linux"
                 sdx:displayName="Linux SMP"
            </sdx:processorGroup>
            </sdx:configuration>
```

The standalone system configuration metadata is similar to the metadata for the Linux configuration. The processor instance name (`sdx:cpuInstance`) is required to build the standalone BSP `libxil.a`. There are no additional boot files copied to the SD card, so `sdx:imageData` is not used in this configuration. The configuration also requires a linker script (`sdx:ldscript`). The emulation flow does not require U-boot (`sdx:qemuBoot`), but it does require a device tree (`sdx:Devicetree`) for QEMU to register devices.

```
<sdx:configuration sdx:name="standalone"
                           sdx:displayName="Standalone OS (Zynq
7000)"
          sdx:defaultProcessorGroup="a9_0">
            <sdx:description>Standalone OS running on Zynq
7000</sdx:description>
            <sdx:prebuilt sdx:data="prebuilt_platform"/>
            <sdx:bootImages sdx:default="standard">
                <sdx:image sdx:name="standard"
          sdx:bif="boot/standalone.bif"
          sdx:readme="boot/generic.readme"
          sdx:qemuDevicetree="qemu/devicetree.dtb"
          sdx:qemuArguments="qemu/qemu_args.txt" />
            </sdx:bootImages>
            <sdx:processorGroup sdx:name="a9_0"

sdx:displayName="A9_0"
          sdx:cpuInstance="ps7_cortexa9_0"
          sdx:cpuType="cortex-a9">
                <sdx:os sdx:name="standalone"
                        sdx:displayName="Standalone
OS"
          sdx:ldscript="aarch32-none/lscript.ld"
            </sdx:processorGroup>
          </sdx:configuration>
```

The FreeRTOS System Configuration metadata is also similar to the metadata for the Linux configuration. The processor instance name (`sdx:cpuInstance`) is required to build the standalone BSP `libxil.a` (FreeRTOS is code is additive to the standalone BSP, implementing a task scheduler and other functions). There are no additional boot files copied to the SD card, so `sdx:imageData` is not used in this configuration. The configuration also requires a linker script (`sdx:ldscript`). The emulation flow is not implemented for FreeRTOS running on the zc702 platform.

Include paths (`sdx:includePaths`), library paths (`sdx:libraryPaths`) and a library (`sdx:libraryNames`) are defined, with the leading slash character '/' indicating the path is relative to the SDSoC installation, for example `<install>/aarch32-none/include/freertos`. If there were no leading slash character '/', the path is relative to the "sw" folder.

```
<sdx:configuration sdx:name="freertos"
                           sdx:displayName="FreeRTOS (Zynq 7000)"
                                 sdx:defaultProcessorGroup="a9_0">
            <sdx:description>FreeRTOS running on Zynq 7000</sdx:description>
            <sdx:prebuilt sdx:data="prebuilt_platform"/>
            <sdx:bootImages sdx:default="standard">
                <sdx:image sdx:name="standard"
          sdx:bif="boot/freertos.bif"
          sdx:readme="boot/generic.readme" />
            </sdx:bootImages>
            <sdx:processorGroup
sdx:name="a9_0"
          sdx:displayName="A9_0"
          sdx:cpuInstance="ps7_cortexa9_0"
          sdx:cpuType="cortex-a9">
```

```
                <sdx:os sdx:name="freertos"
                        sdx:displayName="FreeRTOS"
          sdx:includePaths="/aarch32-none/include/
freertos"
          sdx:libraryPaths="/aarch32-none/lib/freertos"
                        sdx:libraryNames="freertos"
                         sdx:ldscript="freertos/
lscript.ld"
            </sdx:processorGroup>
          </sdx:configuration>
          </sdx:systemConfigurations></sdx:platform>
```

Software Platform XML Metadata Reference

An SDSoC software platform XML file (`.spfm`) defines one or more system configurations. When building an SDSoC application, the user specifies the system configuration that defines a software runtime environment for a target operating system (OS), and metadata used by the SDSoC system compilers to generate application-specific systems-on-chip built upon the hardware platform. The information includes:

- Optional prebuilt platform data, for example bitstream and files normally generated by SDSoC, which minimizes SDSoC build times when a design contains no accelerators.
- SD card image data files, for example bootloaders, boot files for Linux (kernel, ramdisk, devicetree) and the Boot Image File (BIF) template used with by SDSoC with the bootgen utility to create the images.
- Compile and link software files and settings for a targeted processor:
  - Optional include files and libraries required by platform software applications.
  - Linker scripts.
  - Optional BSP configuration files and repositories.
  - Target device information.

The informal software platform XML schema is shown below, although not all elements or attributes are required:

```
<?xml version="1.0" encoding="UTF-8"?><sdx:platform sdx:vendor="xilinx.com"
              sdx:library="sdx"
              sdx:name="platform_name"
              sdx:version="1.0"
              sdx:schemaVersion="1.0"
              xmlns:sdx="http://www.xilinx.com/sdx">
          <sdx:description>software_platform_description
          </sdx:description>
        <sdx:systemConfigurations
sdx:defaultConfiguration="default_name">
        <sdx:configuration sdx:name="name"

sdx:displayName="GUI_name"
          sdx:defaultProcessorGroup="default_name">
          <sdx:description>configuration_description
               </sdx:description>
```

```
            <sdx:prebuilt sdx:data="prebuilt_platform_path"/>
             <sdx:bootImages sdx:default="default_name">
                <sdx:image sdx:name="name"
                           sdx:bif="path_to_bif_file"
                           sdx:imageData="path_to_image_data"
                           sdx:extraData="path_to_more_image_data"
                           sdx:mountPath="sd_card_mount_path"
                           sdx:readme="path_to_readme_file"
                           sdx:qemuDevicetree="path_qemu_devicetree"
                           sdx:qemuBoot="path_qemu_uboot"
                           sdx:qemuArguments="path_arguments_file" />
            </sdx:bootImages>
            <sdx:processorGroup
sdx:name="name"
          sdx:displayName="GUI_name"
                                sdx:cpuType="cpu_type"
                                sdx:cpuInstance="instance_name">
                <sdx:os sdx:name="os_name"
                        sdx:displayName="GUI_name"
                        sdx:includePaths="path_to_include_folder"
                        sdx:libraryPaths="path_to_library_folder"
                        sdx:libraryNames="list_of_library_names"
                        sdx:ldscript="path_to_linker_script"
                        sdx:bspConfig="path_to_mss_file"
                        sdx:bspRepo="path_to_bsp_repository"
                        sdx:compilerOptions="compiler_options"
                        sdx:linkerOptions="linker_options"
            </sdx:processorGroup>
          </sdx:configuration>
              :
              .
           <sdx:configuration sdx:name="name"

sdx:displayName="GUI_name"
          sdx:defaultProcessorGroup="default_name">
              :
              .
        </sdx:configuration>
          </sdx:systemConfigurations></sdx:platform>
```

The first line of the file defines the format of the file to be XML and is mandatory:

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

<sdx:platform>

The `<sdx:platform>` XML element is required as a container for all software platform information. It has the following attributes:

```
<sdx:platform sdx:vendor="xilinx.com"
 sdx:library="sdx"
 sdx:name="platform_name"
 sdx:version="1.0"
```

```
sdx:schemaVersion="1.0"
xmlns:sdx="http://www.xilinx.com/sdx">
```

Table 3: Attributes of <sdx:platform>

Attribute	Description
sdx:vendor	Vendor information
sdx:library	Library name space
sdx:name	Platform name
sdx:version	Platform version
sdx:schemaVersion	Platform schema version
xmlns:sdx	XML namespace is sdx

<sdx:description>

The `<sdx:description>` element describes the software platform and may be used when SDSoC displays summary information.

```
<sdx:description>software_platform_description
</sdx:description>
```

<sdx:systemConfigurations>

The `<sdx:systemConfigurations>` XML element is required as a container for all system configurations.

The `<sdx:systemConfigurations>` element specifies the name of the default `sdx:configuration` to use as the selection in the SDx environment IDE New Project Wizard drop-down list of system configurations, or when the SDSoC compiler tool sdscc/sds++ command line does not include the –sds-sys-config <system_configuration_name> option.

```
<sdx:systemConfigurations sdx:defaultConfiguration="default_name">
```

The `<sdx:configuration>` element defines a System Configuration and includes the following attributes:

```
<sdx:configuration sdx:name="name"
sdx:displayName="GUI_name"
sdx:defaultProcessorGroup="default_name">
```

Table 4: Attributes of <sdx:configuration>

Attribute	Description
sdx:name	System Configuration name, used in the sdscc/sds++ command line option -sds-sys-config <system_configuration_name>.

Attribute	Description
sdx:displayName	System Configuration name, used in the SDx Environment IDE New Project Wizard selection menus. This is typically a longer, more descriptive name than sdx:name.
sdx:defaultProcessorGroup	Default processor group name, specifies the default sdx:processorGroup to use as the selection in the SDx Environment IDE New Project Wizard drop-down list of Target CPUs, or when the sdscc/sds++ command line does not include the –sds-proc <processor_group_name> option.

The `<sdx:configuration>` element can also contain multiple XML sub-elements:

Table 5: Sub-elements of <sdx:configuration>

Element	Description
<sdx:description>	Description of the System Configuration
<sdx:prebuilt>	Prebuilt data for the platform, for example bitstream and other generated files
<sdx:bootImages>	Boot image creation data required to create an SD card image
<sdx:processorGroup>	Processor and OS-specific software data and files used to build user applications

<sdx:description> Element

The sdx:description element contains a description of the System Configuration. This is used in the SDx Environment IDE New Project Wizard when a more detailed description is required, for example mouse over the name in a list of system configurations, or when sdscc/sds++ -sds-pf-list or –sds-pf-info options are used to print information about a platform

```
 <sdx:description>configuration_description
</sdx:description>
```

<sdx:prebuilt> Element

The `<sdx:prebuilt>` element specifies pre-generated data for the platform used with the system configuration when building user applications that contain no accelerators. This optional element can be used to reduce build times, since you do not need to wait for a hardware compile of the platform itself to create a bitstream and other required files.

The `<sdx:prebuilt>` element contains a single attribute `sdx:data` which specifies the path to a folder containing pre-generated files.

```
 <sdx:prebuilt sdx:data="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 folder name can be defined by the platform provider).

Pre-built hardware files are automatically used by the SDSoC compiler when an application has no hardware functions specified in the SDx Environment IDE in the SDx Project Settings view or in the sdscc/sds++ command line using the platform, system configuration and processor flags:

```
-sds-pf zc702 -sds-sys-config linux -sds-proc a9_0
```

To force a full Vivado tools bitstream and SD card image compile, use the following sdscc option:

```
-rebuild-hardware
```

Files used to populate the `<path_to_platform>/sw/prebuilt_data` are found in the _sds project folder after creating the application ELF and bitstream:

- bitstream.bit : found in _sds/p0/ipi/<platform>.runs/impl_1/bitstream.bit
- <platform>.hdf : found in _sds/p0/ipi/<platform>.sdk
- apsys_0.xml and partitions.xml : found in _sds/.llvm
- portinfo.c and portinfo.h : found in _sds/swstubs

If the `<sdx:prebuilt>` element is not defined, SDSoC will generate the bitstream for applications without hardware accelerators. Examples can be found in factory platforms in `<install>/platforms/<platform>/sw/prebuilt_data`.

<sdx:bootImages> Element

The `<sdx:bootImages>` element is a container for one or more `<sdx:image>` elements, each of which includes attributes used to create an SD card image. The attribute `sdx:default` specifies the name of the default `<sdx:image>` to use.

TIP: *In the 2016.4 release each `<sdx:bootImages>` element only contains a single `<sdx:image>` element.*

```
<sdx:bootImages sdx:default="default_name">
<sdx:image sdx:name="name"
sdx:bif="path_to_bif_file"
sdx:imageData="path_to_image_data"
sdx:extraData="path_to_more_image_data"
sdx:mountPath="sd_card_mount_path"
sdx:readme="path_to_readme_file"
sdx:qemuDevicetree="path_qemu_devicetree"
sdx:qemuBoot="path_qemu_uboot"
sdx:qemuArguments="path_arguments_file"
/>
</sdx:bootImages>
```

The `<sdx:image>` element contains attributes used when SDSoC creates SD card boot images.

Table 6: Attributes of <sdx:image>

Attribute	Description
sdx:name	Image name
sdx:bif	Path to Boot Image File (BIF) used to create the SD card
sdx:imageData	Path to a folder containing boot files to copy to the SD card

Attribute	Description
sdx:extraData	Path to a folder containing additional files to copy to the SD card
sdx:mountPath	SD card mount path for Linux running on the target
sdx:readme	Readme file for the SD card
sdx:qemuDevicetree	Path to QEMU devicetree file (SDSoC emulation flows)
sdx:qemuBoot	Path to QEMU U-boot executable (SDSoC emulation flows)
sdx:qemuArguments	Path to QEMU arguments file (SDSoC emulation flows)

sdx:name Attribute

The `sdx:name` attribute is required and specifies the name of the `<sdx:image>` element.

sdx:bif Attribute

The `sdx:bif` attribute is required and specifies the path to a Boot Image File (BIF) relative to the software platform folder where the BIF file must exist. For example, the attribute:

```
sdx:bif="boot/linux.bif"
```

resolves to the path `<path_to_platform>/sw/boot/linux.bif`, where `<path_to_platform>/sw` is the folder containing the `<platform>.spf` file and both of these values are attributes specified in the file `<path_to_platform>/<platform>.xpfm` file.

An example platform BIF file template for a Linux target has the following contents:

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

During system generation, the SDSoC system compiler reads this template and inserts application-specific file names to generate the BIF file. This file is passed to the bootgen utility to create the boot image.

```
/* linux */
the_ROM_image:
{
[bootloader]<path_to_platform>/boot/fsbl.elf
<path_to_generated_bitstream>/<project_name>.elf.bit
<path_to_platform>/boot/u-boot.elf
}
```

An example `standalone.bif` file has the following contents:

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

```
<elf>
}
```

During system generation, the SDSoC system compiler reads this template and inserts application-specific file names to generate the BIF file. This file is passed to the bootgen utility to create the boot image.

```
/* standalone */
the_ROM_image:
{
[bootloader]<path_to_platform>/boot/fsbl.elf
<path_to_generated_bitstream_directory>/<project_name>.elf.bin
<path_to_generated_application_elf_directory>/<project_name>.elf
}
```

sdx:imageData Attribute

The `sdx:imageData` attribute is required for Linux images and specifies a folder containing boot files to copy to the SD card image whose path is relative to the software platform folder. For example, the attribute:

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

resolves to the path `<path_to_platform>/sw/image`, where <path_to_platform>/sw is the folder containing the `<platform>.spfm` file.

The folder typically contains Linux boot files, and SDSoC copies all folders and files found in the folder to the root of the SD card. If you are using a unified boot image file (.ub) containing a kernel image, device tree and root file system, the folder typically contains an `image.ub` file. If you use separate `image.ub`, `devicetree.dtb` and `ramdisk.image.gz` files, the folder contains those files.

sdx:extraData Attribute

The `sdx:extraData` attribute is optional. It specifies a folder containing additional files to copy to the SD card folder whose path is relative to the software platform folder. For example, the attribute:

```
sdx:extraData="sdfiles"
```

resolves to the path `<path_to_platform>/sw/sdfiles`, where <path_to_platform>/sw is the folder containing the `<platform>.spfm` file.

The `sdx:extraData` attribute is optional and allows you isolate non-boot SD card files from the boot files specified by `sdx:imageData`.

sdx:mountPath Attribute

The `sdx:mountPath` attribute is optional. It specifies the SD card mount path, which defaults to /mnt if not specified.

sdx:readme Attribute

The `sdx:readme` attribute is optional. It specifies a readme file copied to the root of the SD card and whose path is relative to the software platform folder. For example, the attribute:

```
sdx:readme="linux/readme.txt"
```

resolves to the path `<path_to_platform>/sw/linux/readme.txt`, where <path_to_platform>/sw is the folder containing the `<platform>.spfm` file.

An example README file:

```
-- SD card boot image --

Platform: <platform>
Application: <elf>

1. Copy the contents of this directory to an SD card
2. Set boot mode to SD
   Revision C and earlier boards:
     Jumper J22 1-2
     Jumper J25 1-2
   Revision D and newer boards:
     DIP switch SW16 positions 3 and 4 set to 1
3. Insert SD card and turn board on
```

If the `sdx:readme` attribute is used, SDSoC prepends build information to the start of the file before writing it to the SD card. If build information is not required to be inserted into the README file, the README file can be added to a folder defined by the `sdx:imageData` or `sdx:extraData` attributes and the `sdx:readme` attribute can be omitted.

sdx:qemuDevicetree, sdx:qemuBoot, and sdx:qemuArguments Attributes

The `sdx:qemuDevicetree`, `sdx:qemuBoot` and `sdx:qemuArguments` attributes define the path to the device tree, U-boot executable and QEMU arguments files used in SDSoC emulation flows, where the user application executes on QEMU and interacts with programmable logic running in the Vivado simulator. The paths to these files are relative to the software platform folder, where <path_to_platform>/sw is the folder containing the `<platform>.spfm` file.

The `sdx:qemuDevicetree` attribute is required whether the operating system (OS) running on the target is Linux or baremetal. The device tree is used by QEMU to enable device support.

The `sdx:qemuBoot` attribute is required when Linux is running on the target, but not for baremetal software. It points to the U-boot executable used to create Linux SD card boot images.

The `sdx:qemuArguments` attribute is required for all target OS which support emulation flows.

For an example of QEMU software platform files and support, see the folder `<install>/platforms/zc702/sw/qemu`.

TIP: *Platform support for emulation is optional and these attributes are not required if your platform does not support emulation.*

<sdx:processorGroup> Element

The <sdx:processorGroup> element defines compile and link software files and settings for a targeted processor such as:

- Target device information.
- Optional include files and libraries required by platform software applications.
- Linker scripts.
- Optional BSP configuration files and repositories.

The <sdx:processorGroup> element includes the following attributes:

```
<sdx:processorGroup sdx:name="name"
sdx:displayName="GUI_name"
 sdx:cpuType="cpu_type"
 sdx:cpuInstance="instance_name">
```

Table 7: Attributes of <sdx:processorGroup>

Attribute	Description
sdx:name	Processor Group name, used in the sdscc/sds++ command line option -sds-proc <processor_name>, in addition to the –sds-sys-config <system_configuration_name> option. Use one of the following names depending on the target processor: a9_0, a53_0 or r5_0.
sdx:displayName	Processor Group name, used in the GUI Target CPU selection menus. This is typically a longer, more descriptive name than sdx:name.
sdx:cpuType	The target CPU type: cortex-a9, cortex-a53 or cortex-r5.
sdx:cpuInstance	The CPU instance name, for example ps7_cortexa9_0, psu_cortexa53_0, or psu_cortexr5_0. This is optional when running the Linux OS on the target, but required for the Standalone OS (used in BSP generation).

The <sdx:processorGroup> element contains a single sub-element <sdx:os>, which optionally contains paths to file and folders, settings and other data used to build the user application for the target operating system.

The <sdx:os> element contains the following attributes:

```
 <sdx:os sdx:name="os_name"
sdx:displayName="GUI_name"
sdx:includePaths="path_to_include_folder"
sdx:libraryPaths="path_to_library_folder"
sdx:libraryNames="list_of_library_names"
sdx:ldscript="path_to_linker_script"
sdx:bspConfig="path_to_mss_file"
sdx:bspRepo="path_to_bsp_repository"
sdx:compilerOptions="compiler_options"
sdx:linkerOptions="linker_options"
/>
</sdx:processorGroup>
```

Table 8: Attributes of <sdx:os>

Attribute	Description
sdx:name	Operating system name: linux, standalone or freertos

Attribute	Description
sdx:displayName	Operating system name, used in the GUI OS labels. This is typically a longer, more descriptive name than sdx:name.
sdx:includePaths	Directory paths containing platform include files
sdx:libraryPaths	Directory paths containing platform library files
sdx:libraryNames	Platform library names required to link user applications
sdx:ldscript	Linker script used to link the user application
sdx:bspConfig	Configuration file for building the standalone BSP (.mss)
sdx:bspRepo	Software repository used to build the standalone BSP
sdx:compilerOptions	Options required to compile applications using the platform
sdx:linkerOptions	Options required to link applications using the platform

sdx:name Attribute

The sdx:name attribute specifies the operating system running on the processor. Valid values are linux, standalone or freertos, if supported by the platform.

sdx:displayName Attribute

The sdx:displayName attribute is a descriptive name for the operating system, used in the SDx environment IDE as a label in the New Project Wizard in the Choose Software Platform and Target CPU page.

sdx:includePaths Attribute

The `sdx:includePaths` attribute is optional. It specifies the path to any platform software include folders that must be added when compiling user source files. Separate multiple include paths with a colon character ':'. The include folder is located in the software platform folder. When specified, the SDSoC Compiler tool sdscc/sds++ automatically adds the –I option during compilation. For example, the attribute:

```
sdx:includePaths="aarch32-none/include"
```

resolves to the path `<path_to_platform>/sw/aarch32-none/include`, where <path_to_platform>/sw is the folder containing the `<platform>.spfm` file. When compiling source files, sdscc/sds++ adds the option "–I <path_to_platform>/sw/aarch32-none/include".

When the SDSoC Compiler tool sdscc/sds++ compiles source files, the include path search order is:

- user include paths
- platform include paths
- SDSoC installation include paths
- Vivado HLS include paths (if required)

sdx:libraryPaths Attribute

The `sdx:libraryPaths` attribute is optional. It specifies the path to any platform software library folders that must be added when linking the user application ELF. Separate multiple library paths with a colon character ':'. The library folder is located in the software platform folder and contains libraries used to link applications. When specified, the SDSoC Compiler tool sdscc/sds++ automatically adds the –L option during compilation. For example, the attribute:

```
sdx:libraryPaths="aarch32-none/lib"
```

resolves to the path `<path_to_platform>/sw/aarch32-none/lib`, where <path_to_platform>/sw is the folder containing the `<platform>.spfm` file. When linking the application ELF, sdscc/sds++ adds the option "–L <path_to_platform>/sw/aarch32-none/lib".

When the SDSoC Compiler tool sdscc/sds++ links the application ELF, the library path order is:

- user library paths
- path to SDSoC generated board support package (BSP) libraries (standalone or FreeRTOS)
- platform library paths
- SDSoC installation library paths
- SDSoC generated libraries

TIP: *Do not place the standalone BSP library `libxil.a` in a folder specified using the `sdx:libraryPaths` attribute, since SDSoC automatically builds this library for the user.*

sdx:libraryNames Attribute

The `sdx:libraryNames` attribute is optional. It specifies the names of any platform software libraries that must be passed to the linker when linking the ELF file. Separate multiple library names with a colon character ':'. When specified, the SDSoC Compiler tool sdscc/sds++ automatically adds the –l option when linking the ELF. For example, the attribute:

```
sdx:libraryNames="numeric"
```

causes the –lnumeric option to be used when linking and the library `libnumeric.a` is linked in. The platform library path must be specified using the `sdx:libraryPaths` attribute.

The attribute is optional, since the user can specify the –l option in the SDSoC GUI C/C++ Build Settings dialog and in user Make files.

sdx:ldscript Attribute

The `sdx:ldscript` attribute specifies the path to a linker script file located in the software platform folder. For example, the attribute:

```
sdx:ldscript="standalone/lscript.ld"
```

resolves to the path `<path_to_platform>/sw/standalone/lscript.ld`, where <path_to_platform>/sw is the folder containing the `<platform>.spfm` file.

The linker script is used by the SDSoC Compiler tool sdscc/sds++ when linking your application ELF.

sdx:bspConfig Attribute

The `sdx:bspConfig` attribute is optional and specifies the path to a board support package (BSP) configuration file (.mss) located in the software platform folder. For example, the attribute:

```
sdx:bspConfig="bsp/system.mss"
```

resolves to the path `<path_to_platform>/sw/bsp/system.mss`, where <path_to_platform>/sw is the folder containing the `<platform>.spfm` file.

When the `sdx:bspConfig` attribute is used for generating a platform-specific standalone BSP, the `sdx:includePaths` attribute must be specified and refer to folder containing BSP header files. The SDSoC Compiler tool sdscc/sds++ command uses this .mss file instead of generating a default BSP configuration based on both the platform and hardware in the programmable logic (PL). Consequently the .mss file must specify drivers required in SDSoC user designs, including the Xilinx AXI DMA driver (scatter-gather mode). See *Generating Basic Software Platforms* (UG1138) for information about BSP configuration files (.mss).

sdx:bspRepo Attribute

The `sdx:bspRepo` attribute is optional and specifies the path to a board support package (BSP) repository folder located in the software platform folder. For example, the attribute:

```
sdx:bspRepo="bsp/repo"
```

resolves to the path `<path_to_platform>/sw/bsp/repo`, where <path_to_platform>/sw is the folder containing the `<platform>.spfm` file.

TIP: *When the `sdx:bspRepo` attribute is used, the `sdx:bspConfig` attribute must also be specified.*

The use case supports platforms which define drivers or libraries that must be included in the BSP. The SDSoC Compiler tool sdscc/sds++ command adds the folder to the repository search path when generating a standalone BSP llibrary (`libxil.a`). Refer to *Generating Basic Software Platforms* (UG1138) for information about BSP repositories.

sdx:compilerOptions Attribute

The `sdx:compilerOptions` attribute specifies compiler options required to compile an application that uses the platform, exclusive of include path attributes.

This attribute is optional and typically not required. The `template.xml` used by the SDSoC GUI when creating template applications supports a similar attribute and adds compiler options to the generated Make file, while the `sdx:compilerOptions` attribute causes the specified options to be inserted when the SDSoC Compiler tool sdscc/sds++ invokes the underlying cross-compiler to compile all source files for every application.

sdx:linkerOptions Attribute

The `sdx:linkerOptions` attribute specifies linker options required to link an application that uses the platform, exclusive of library path and library name attributes.

This attribute is optional and not typically used. The `template.xml` used by the SDSoC GUI when creating template applications supports a similar attribute and adds linker options to the generated Make file, while the `sdx:linkerOptions` attribute causes the specified options to be inserted when the SDSoC Compiler tool sdscc/sds++ invokes the underlying cross-compiler to link the ELF file for every application.

XILINX
ALL PROGRAMMABLE

Appendix A

SDSoC Platform Migration

Introduction

To support a newer release of the SDx Development Environment you must migrate a hardware platform from an earlier release. The SDSoC 2016.3 and 2016.4 platform folder structure and metadata are the same, so migrating from 2016.3 to 2016.4 is simply a matter of upgrading the Vivado IP Integrator block design to the latest release and rebuilding the software components with the 2016.4 SDSoC tools. The process of upgrading the block design is explained in the *Vivado Design Suite User Guide: Designing IP Subsystems with IP Integrator* (UG994).

However, the SDSoC 2016.2 platform folder structure and metadata is different from SDSoC 2016.3 and later releases. To migrate hardware platforms from the 2016.2 release, you must use the conversion utility and perform manual edits as explained in the following process.

Migrating from SDSoC 2016.2

An internal command line utility, `sdspfm_convert`, is available to migrate an SDSoC 2016.2 platform for Zynq®-7000 All Programmable (AP) SoC or Zynq UltraScale+™ MPSoC device to the SDx 2016.3 directory structure and platform XML files. While not a general utility that is guaranteed to work for all platforms, it minimizes the time required for migrating platforms from 2016.2 to 2016.4.

TIP: *The utility does not upgrade the Vivado Design Suite project defining the hardware platform, but it does create the new directory structure and platform XML files. After the initial conversion, you can manually update and modify the generated platform and files, including prebuilt platform files.*

To use the command in a command shell with its environment set up to run SDx 2016.4, type the following command:

```
sdspfm_convert <path_to_existing_platform> <output_converted_platform>
<zynq | mpsoc>
```

Where:

- `<path_to_existing_platform>` - defines the path to the existing 2016.2 hardware platform files.
- `<output_converted_platform>` - defines the output folder to write the converted hardware platform for 2016.4.
- <zynq | mpsoc> - specifies either the Zynq®-7000 All Programmable (AP) SoC or the Zynq UltraScale+™ MPSoC device as the processor targeted by the hardware platform being converted.

For example, the following command:

```
sdspfm_convert /home/mia/2016.2/platforms/zc702 /home/mia/2016.4/platforms
zynq
```

reads the 2016.2 zc702 platform directory structure:

- /home/mia/2016.2/platforms
  - zc702
    - aarch32-none
    - boot
    - freertos
    - hardware
    - qemu
    - vivado
    - zc702_hw.pfm
    - zc702_sw.pfm

and creates the 2016.4 platform structure and XML files, based on the existing platform structure and files in the 2016.2 platform folder.

- /home/mia/2016.4/platforms
  - zc702
    - hw
      - vivado
      - zc702.hpfm
    - sw
      - aarch32-none
      - boot
      - freertos
      - image
      - prebuilt_platform
      - qemu
      - zc702.spfm
    - zc702.xpfm

TIP: *For clarity in the example the complete directory structure is not shown.*

Conversion Results

The `sdspfm_convert` utility copies folders referenced in the `.spfm` file into the sw folder, copies the vivado folder into the hw folder, and copies other folders and files into the top-level folder.

If the original platform folder contained a "samples" folder and "README" file, they are copied into the top level of the new platform folder.

The `.hpfm` file is a renamed copy of the `_hw.pfm` file, while the `.xpfm` and `.spfm` files are generated based on existing `.pfm` files.

After conversion, you can edit metadata for the descriptive names used in the GUI, for example `sdx:displayName` metadata.

The SDSoC 2016.2 and 2016.4 software platform XML files contain similar information, but the 2016.4 schema groups metadata for the same OS, allows more groupings to be defined, and enables tools to be more data driven. The table below shows the 2016.2 metadata file structure on the left, and the 2016.4 structure on the right.

```
<xd:libraryFiles
 xd:os="standalone"
 xd:libDir="aarch32-none/lib"
 xd:ldscript="aarch32-none/
lscript.ld"
 />
<xd:libraryFiles
 xd:os="freertos"
 xd:osDepend="standalone"
 xd:includeDir="/aarch32-none/
include/freertos"
 xd:libDir="/aarch32-none/lib/
freertos"
 xd:libName="freertos"
 xd:ldscript="freertos/lscript.ld"
 />
<xd:bootFiles
 xd:os="linux"
 xd:bif="boot/linux.bif"
 xd:readme="boot/generic.readme"
 xd:devicetree="boot/devicetree.dtb"
 xd:linuxImage="boot/uImage"
 xd:ramdisk="boot/uramdisk.image.gz"
 />
<xd:bootFiles
 xd:os="standalone"
 xd:bif="boot/standalone.bif"
 xd:readme="boot/generic.readme"
 />
<xd:bootFiles
 xd:os="freertos"
 xd:bif="boot/freertos.bif"
 xd:readme="boot/generic.readme"
 />
```

```
<sdx:configuration
sdx:name="standalone"
 sdx:displayName="Standalone OS
(Zynq 7000)"
 sdx:defaultProcessorGroup="a9_0">
 <sdx:description>Standalone OS
running
   on Zynq 7000</sdx:description>
 <sdx:prebuilt
sdx:data="prebuilt_platform"/>
 <sdx:bootImages
sdx:default="standard">
    <sdx:image sdx:name="standard"
     sdx:bif="boot/standalone.bif"
     sdx:readme="boot/
generic.readme"
     sdx:qemuDevicetree="qemu/
devicetree.dtb"
     sdx:qemuArguments="qemu/
qemu_args.txt"
     />
  </sdx:bootImages>
  <sdx:processorGroup
sdx:name="a9_0"
   sdx:displayName="A9_0"

sdx:cpuInstance="ps7_cortexa9_0"
   sdx:cpuType="cortex-a9">
  <sdx:os sdx:name="standalone"
   sdx:displayName="Standalone OS"
   sdx:libraryPaths="aarch32-none/
lib"
   sdx:ldscript="aarch32-none/
lscript.ld"
```

	/> </sdx:processorGroup> </sdx:configuration>

The `sdspfm_convert` utility creates an updated platform folder structure and platform XML files which the platform provider can update, but the platform provider must manually update the Vivado project in `hw/vivado` for 2016.4, and any prebuilt platform data, if used. 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.

IMPORTANT: *Do not reuse prebuilt platform files from 2016.2 – certain files are no longer created in 2016.4 (like `devreg.c` and `devreg.h`) and if a design with accelerators is built with old files (`portinfo.c` and `portinfo.h`), the build will fail when compiling these files.*

To summarize the platform migration process:

1. Use the `sdspfm_convert` utility for the initial migration.
2. Manually update the Vivado Design Suite project (`hw/vivado`) to 2016.4.
3. Use the platform to create a software only design using the `-rebuild-hardware` option when linking, to force the bitstream to be regenerated and prevent using any prebuilt data files in the platform.
4. Replace any existing "prebuilt" files from the 2016.2 platform with the 2016.4 versions you just created:
   - `bitstream.bit` : found in `_sds/p0/ipi/<platform>.runs/impl_1/bitstream.bit`
   - `<platform>.hdf` : found in `_sds/p0/ipi/<platform>.sd`.
   - `apsys_0.xml` and `partitions.xml` : found in `_sds/.llvm`.
   - `portinfo.c` and `portinfo.h` : found in `_sds/swstubs` (`devreg.c` and `devreg.h` are no longer created in 2016.4).
5. The platform migration is complete.

XILINX
ALL PROGRAMMABLE

Appendix B

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 buillt upon the ZC702 board available from Xilinx. Only the `zc702_led` platform uses board-specific resources (LEDs).

- `zc702_axis_io` - Accessing a data stream that could represent direct I/O from FPGA pins in an SDSoC platform
- `zc702_led` - Providing software control of IP cores within a 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.

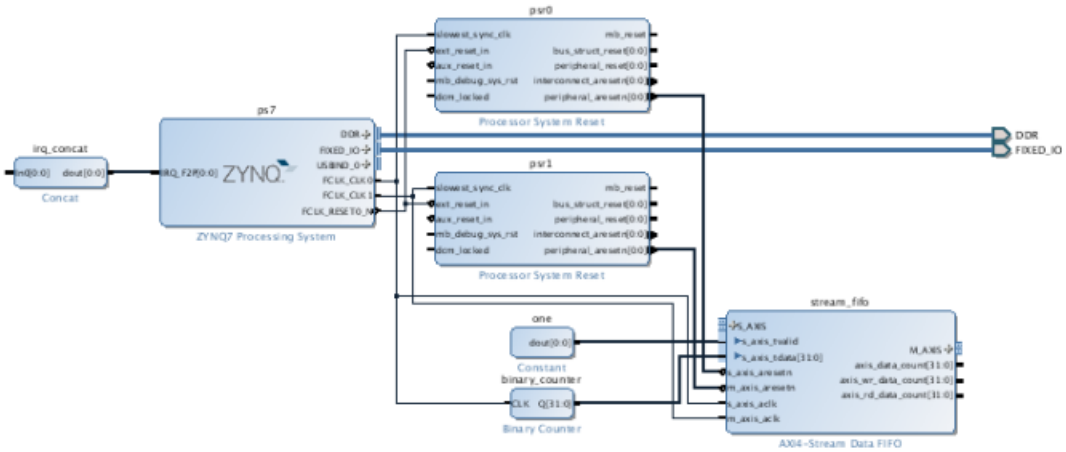
Example: Direct I/O in an SDSoC Platform

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/platforms/zc702_axis_io`.

The hardware component of an SDSoC platform is represented in a Vivado project which is located in the `hw/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 3: 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 Hardware Platform Metadata Creation, the platform hardware port interface is declared in Vivado using Tcl commands, which are contained in the script file `hw/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. Note also the declaration of the associated `proc_sys_reset` IP instances (`psr0, psr1`) that are required of every platform clock.

```
sdsoc::pfm_clock       $pfm FCLK_CLK0 ps7 0 false  psr0

sdsoc::pfm_clock       $pfm FCLK_CLK1 ps7 1 true  psr1
```

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 SDSoC platform hardware description file `zc702_axis_io.hpfm` in the platform `hw` directory.

```
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`.

SDSoC Platform Software Description

The SDSoC™ platform software description is an XML file that contains information required to link against platform libraries and create boot images to run user applications on the hardware platform.

The `zc702_axis_io` platform reuses all of the ZC702 boot files.

- The platform software description file is located in, `<sdx_root>/samples/platforms/zc702_axis_io/sw/zc702_axis_io.spfm`.

  The following element is used by the `sdscc` system compiler to find platform software libraries during compilation and linking.

```
<sdx:os sdx:name="linux"
    sdx:displayName="Linux SMP"
    sdx:includePaths="aarch32-linux/include"
    sdx:libraryPaths="aarch32-linux/lib"
    sdx:libraryNames="zc702_axis_io"
/>
```

Similarly, the boot files are specified as follows:

```
<sdx:bootImages sdx:default="standard">
    <sdx:image sdx:name="standard"
        sdx:bif="boot/linux.bif"
        sdx:imageData="image"
        sdx:mountPath="/mnt"
        sdx:readme="boot/generic.readme"
   />
</sdx:bootImages>
```

Platform Sample Designs

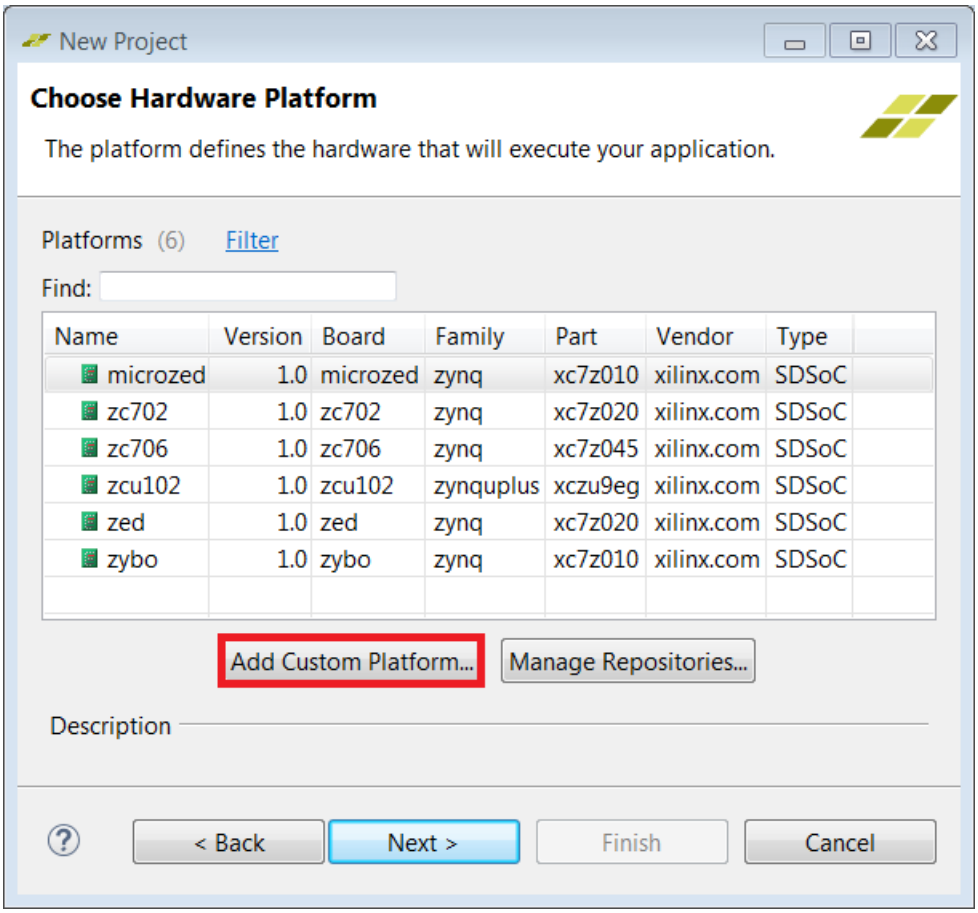
An SDSoC platform can include sample applications that demonstrate its use. The SDx IDE looks for a file called `samples/template.xml` for information on where the sample application source files reside within the platform. The template file for the `zc702_axis_io` platform lists several test applications,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

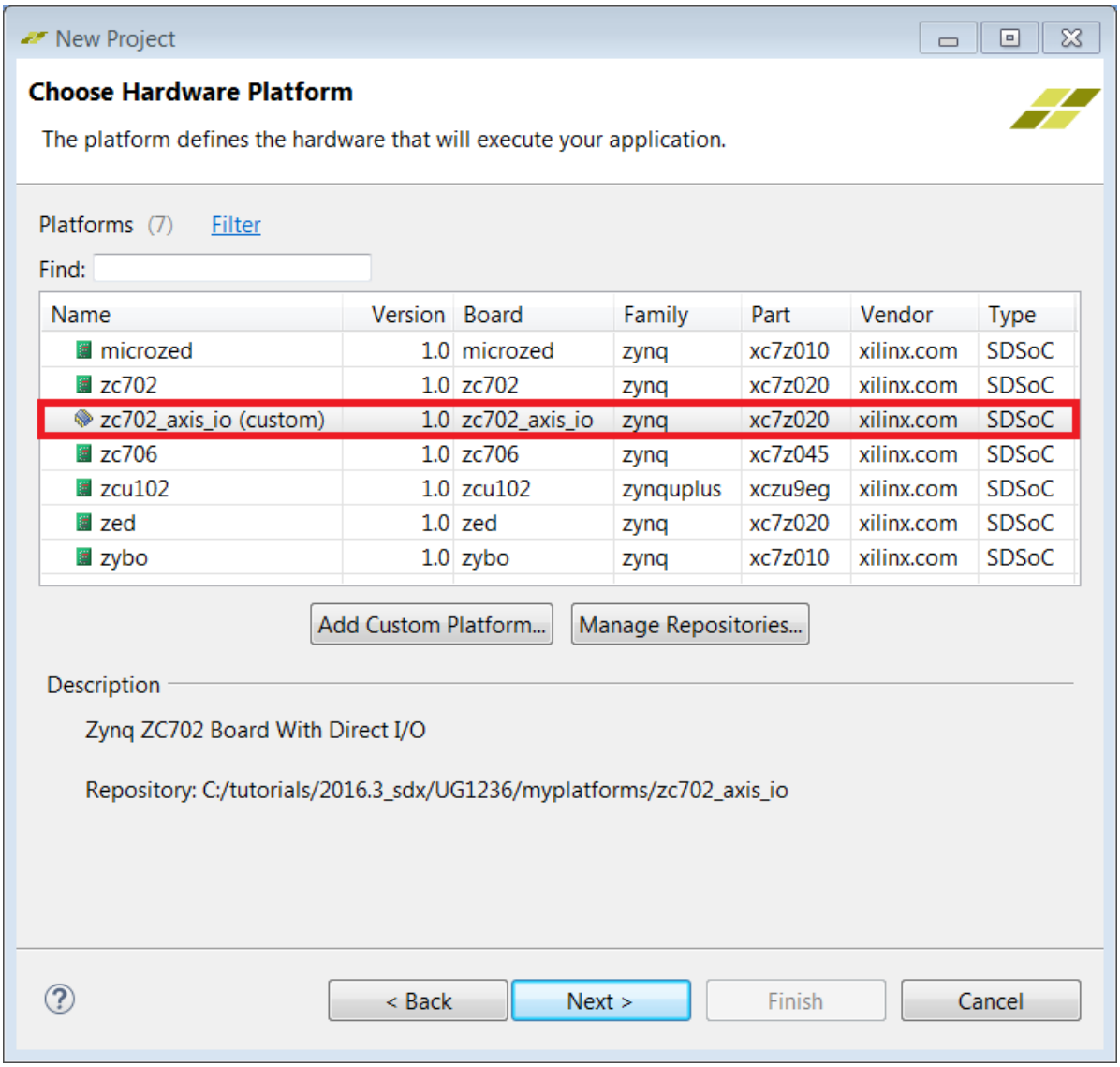
1. Launch Xilinx SDx 2016.x and provide a path to your workspace such as `<path_to_tutorial>/myplatforms/`.

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: Software Control of Platform IP

This example demonstrates how an SDSoC platform can provide a software library that can be linked into an application, independent of the system inference and generation process. Such libraries can be used, for example, to control IP blocks within a platform.

This platform example includes a general purpose I/O (AXI GPIO) IP block implemented in programmable logic to write to LEDs on a ZC702 board. The same platform hardware system supports both standalone and Linux applications. The standalone software library uses the GPIO standalone driver, while the Linux target library uses the Linux UIO (userspace I/O) framework to communicate with the GPIO peripheral directly from application code.

SDSoC Platform Hardware Description

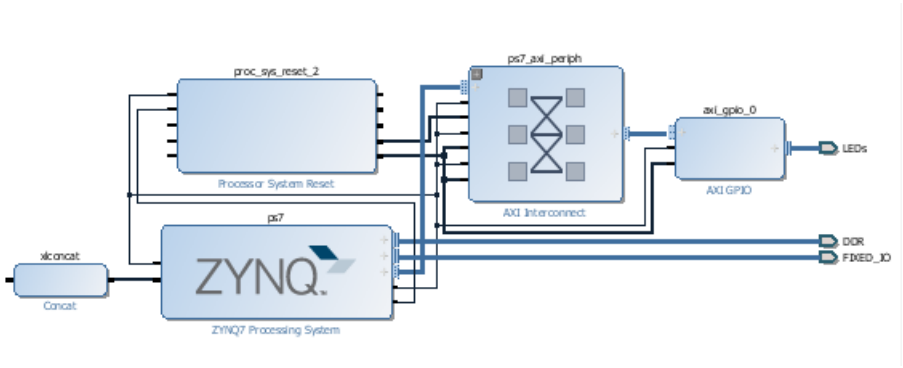
In the `zc702_led` platform, the AXI GPIO IP used to control the LEDs on the ZC702 board is accessible to SDSoC applications only through its platform software interface. The first step is to create the platform hardware metadata file `zc702_led.hpfm`. The following steps describe the commands used to create the hardware platform metadata XML file.

To view the hardware system, within an SDx Terminal, open `<sdx_root>/samples/platforms/zc702_led/hw/vivado/zc702_led.xpr` in Vivado.

1. After the design opens in the Vivado IDE, select **Open Block Diagram**.

   As shown in the following figure, the design uses the Vivado `AXI_GPIO` IP block to connect to the LEDs on the ZC702 board.

   Figure 4: zc702_led Block Diagram

2. The following Tcl command creates a hardware platform object within Vivado.

   ```
   set pfm [sdsoc::create_pfm zc702_led.hpfm]
   ```

3. 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_led`'.

   ```
   sdsoc::pfm_name        $pfm "xilinx.com" "xd" "zc702_led" "1.0"
   sdsoc::pfm_description $pfm "Zynq ZC702 Board With Software Control of
   Platform IP"
   ```

4. The following command declares the default platform clock to have id 2:. The 'true' argument indicates that this clock is the platform default. Note also the declaration of the associated `proc_sys_reset_2` IP instance that is required of every platform clock.

   ```
   sdsoc::pfm_clock       $pfm FCLK_CLK2 ps7 2 true  proc_sys_reset_2
   ```

5. 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. Observe that this platform does not declare the `M_AXI_GP0` port that is used within the platform to write the the LEDs.

   ```
   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
```

6. The following commands declare the interrupt inputs:

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

7. The following command creates the SDSoC platform hardware description file `zc702_led.hpfm`.

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

Platform Software Description

- For standalone applications, an SDSoC platform requires a linker script and header files. From these, the `sdscc` system compiler creates an application-specific board support package (BSP) and link the application code against a platform-specific `libXil.a` library. You create the linker script as follows.
  1. Open the hardware system in the Vivado IDE and use the **File→Export→Export Hardware** command.
  2. Create a hardware platform specification project from the exported hardware system as you would normally do using the Xilinx SDK. Refer to "Exporting a Hardware Definition to SDK" in *Vivado Design Suite User Guide: Designing IP Subsystems with IP Integrator* (UG994) for more information.
  3. Create a board support package (BSP) project as you would normally do using the Xilinx SDK. Refer to *Generating Basic Software Platforms Reference Guide* (UG1138).
  4. Create a "Hello World" application project using the hardware specification and BSP from the two previous steps.

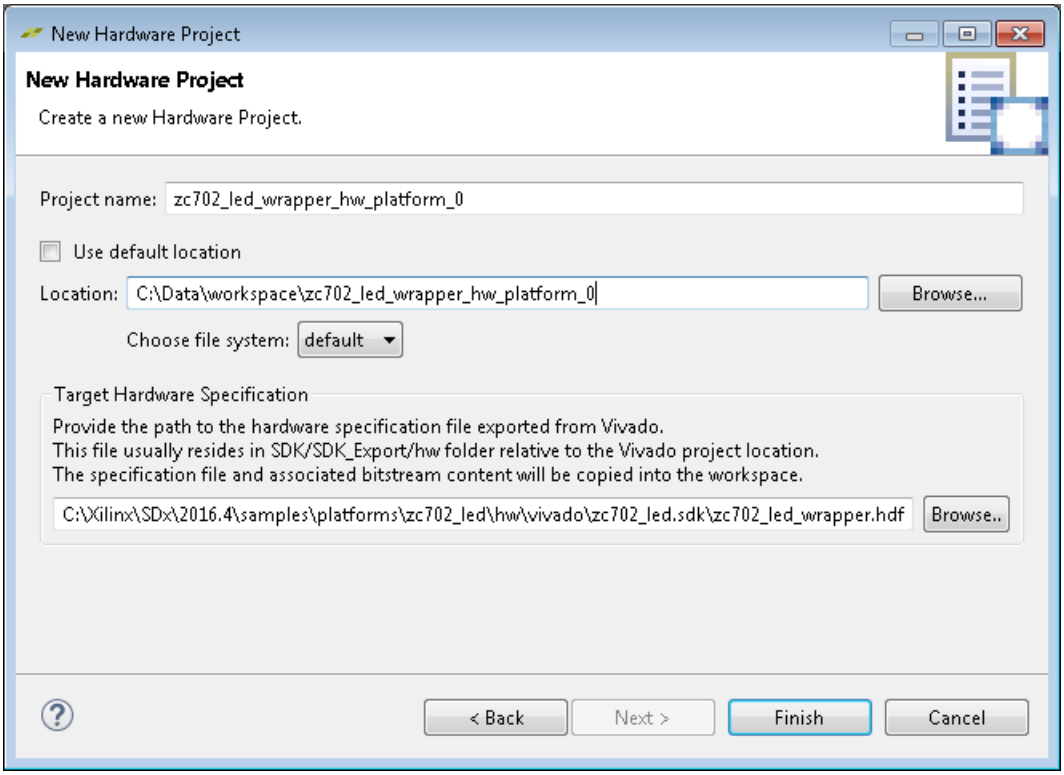
  The linker script created for the "Hello World" project and the header files from the BSP provide the SDSoC platform software component.
- For Linux applications, the software component of the SDSoC platform provides a Linux boot environment for the `zc702_led` that is identical to the ZC702 platform that is provided as part of the SDSoC environment except for the `devicetree.dtb`, which is required to register the AXI GPIO platform peripheral. The `zc702_led` platform also includes a software library to access the AXI GPIO peripheral via the Linux UIO driver framework.

Standalone Platform Software

For SDSoC platforms that support standalone applications, the platform software is built using the same hardware handoff mechanism that is used in the Xilinx SDK. In the Vivado environment, the hardware export command **File→Export→Export Hardware** creates the hardware handoff file `<sdx_root>/samples/platforms/zc702_led/hw/vivado/`

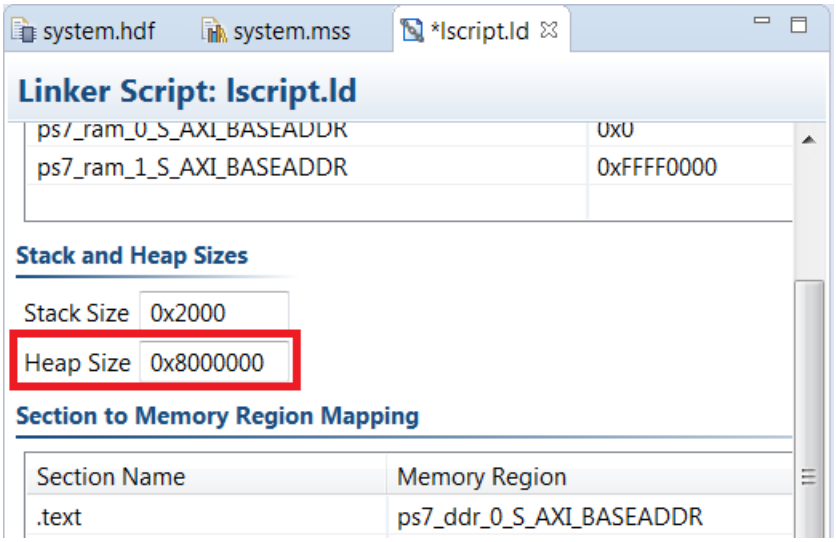
`zc702_led.sdk/zc702_led_wrapper.hdf`. This section briefly summarizes the steps to create a linker script for the platform.

1. To create an "SDK-style" hardware platform specification within the SDx IDE, select **File→New→Project**.
2. In the New Project dialog box, expand the **Xilinx** folder, select **Hardware Platform Specification** and click **Next**.
3. In the New Hardware Project dialog box, click on the **Browse** button in the Target Hardware Specification section, navigate to the `<sdx_root>/samples/platforms/zc702_led/zc702_led.sdk` folder, select the `zc702_led_wrapper.hdf` file and click **Open**.



4. The **Finish** command completes this step.
5. The next step is to create an SDK-style board support package. This can be done within the SDx IDE by selecting **File→New→Project**.
6. In the New Project dialog box, expand the **Xilinx** folder, select **Board Support Package**. The **Next** command proceeds.
7. The **Project name** field has standalong_bsp_0 entered by default. The **Hardware Platform** is selected by default. The **Board Support Package OS** is selected as standalone by default. The **Finish** command completes this step.
8. The Board Support Package Settings dialog box opens. For SDSoC platforms, in the Supported Libraries section, you must check the box next to xilffs to select the `xilffs` library.
9. The SDSoC platform for standalone applications requires a linker script, which can be created as a by-product of building a 'Hello, World" application. In the SDx IDE, select **File→New→Application Project** to create an application project for this hardware specification and BSP.
10. In the Board Support Package click **Use Existing** and select standalone_bsp_0 created earlier, and in the next step,

11. Select `Hello World` to create an application software project. The **Finish** command completes this step.
12. The Open Associated Perspective dialog box opens. Click **Yes**.
13. In the SDx IDE, select the `Hello World` project in the Project Explorer view and build the project selecting the "hammer" icon in the taskbar or by right-clicking in the Project Explorer window and selecting **Build Project**.
14. The default heap size is too small for most SDSoC applications. Open the linker script under `Hello_World/src/lscript.ld` by double clicking on it in the Project Explorer window and change the Heap Size to 0x8000000



15. Save the linker script by selecting **File→Save** from the menu.
16. You can inspect the linker script in `<sdx_root>/samples/platforms/zc702_led/sw/aarch32-none`.

Linux Software Platform

IMPORTANT: *This section provides a simple example of software control of memory mapped platform IP in an embedded Linux target platform, but is not a self-contained primer on embedded Linux, user space drivers, or device trees. If concepts are unfamiliar, there are many references that can provide more information, and these should be consulted before working through this example.*

The makefile for the software library that controls the AXI GPIO in Linux, is located in `<sdx_root>/samples/platforms/zc702_led/src/Makefile`. The default build target creates a software library containing the UIO driver for the `axi_gpio` block consisting of the files `uio_axi_gpio.[ch]`. This user space driver provides a simple API for controlling the GPIO IP that could be applied to other memory-mapped IP blocks within a platform.

The header file and library reside in the `sw/aarch32-linux/{include,lib}` subdirectories in the platform.

For the SDSoC platform to support Linux applications, you must update the Linux device tree provided with the `zc702` platform that is used to boot the platform. The device tree provided as part of the `zc702_led` platform was manually created by modifying the `devicetree.dtb` from the ZC702 platform. First, the `zc702 devicetree.dtb` was converted to a text format (`.dts` or device tree source) using the dtc compiler

```
dtc -I dtb -O dts -o devicetree.dts boot/devicetree.dtb
```

Two changes to the device tree file `devicetree.dts` are needed to register the `axi_gpio_0` platform peripheral with Linux. First, add `uio_pdrv_genirq.of_id=generic-uio` to `bootargs` as follows:

```
bootargs = "console=ttyPS0,115200 root=/dev/ram rw earlyprintk
uio_pdrv_genirq.of=generic-uio";
```

Then add the following device tree blob by inserting it into the device tree as the lexically first occurring `generic-uio` device within the `amba` record in the device tree:

```
gpio@41200000 {
    compatible = "generic-uio";
    reg = <0x41200000 0x10000>;
};
```

The name on the second devicetree blob mentioned above must be unique. Xilinx has adopted a convention of using the base address for the peripheral computed by the Vivado tools during system generation as a guarantee. The value of the `reg` member must be the base address for the peripheral and the number of byte addresses in the corresponding address segment for the IP. Both of these are visible in the Vivado IP integrator Address Editor.

The `dtc` utility will convert the device tree back to binary format required by the Linux kernel.

```
dtc -I dts -O dtb -o devicetree.dtb boot/devicetree.dts
```

The UIO driver in the `zc702_led/lib` directory provides the required hooks for the UIO framework:

```
int axi_gpio_init(axi_gpio *inst, const char* instnm);
int axi_gpio_release(axi_gpio *inst);
```

Any application that accesses the peripheral must first call the initialization function before accessing the peripheral and must release the resource when it is finished. The SDSoC test program in `samples/arraycopy.cpp` demonstrates example usage.

The `<sdx_root>/samples/platforms/zc702_led/sw/zc702_led.spfm` file declares the platform paths in the following element

```
    <xd:libraryFiles
      xd:os="linux"
      xd:libName="zc702_led"
      xd:libDir="aarch32-linux/lib"
      xd:includeDir="aarch32-linux/include"
    />
```

For more information on device trees and the Linux UIO framework, Xilinx recommends training material available on the Web, for example: http://www.free-electrons.com/docs.

Platform Sample Designs

The platform contains a samples subdirectory with a single test application called `arraycopy`. The test application contains a simple `arraycopy` hardware function invoked within a loop. In addition to copying the array input to the hardware function output, the application code lights the LEDs on the ZC702 board to match the binary representation of the loop index.

The `template.xml` file in this directory registers sample applications with the SDx IDE.

```
<template location="arraycopy" name="Array copy" description="Linux test
application">
    <supports>
        <and>
            <os name="Linux"/>
        </and>
    </supports>
</template>
<template location="arraycopy_sa" name="Array copy" description="Standalone
test application">
    <supports>
        <and>
            <os name="standalone"/>
        </and>
    </supports>
</template>
```

Although quite simple, the `zc702_led` platform demonstrates how to access platform peripherals outside of the SDSoC software run-time. Standalone applications include direct calls to the peripheral device driver APIs, and Linux applications can employ the Linux UIO framework (memory-mapped read/write) to control platform peripherals, accessible via an SDSoC platform software library.

Example: Sharing a Platform IP AXI Port

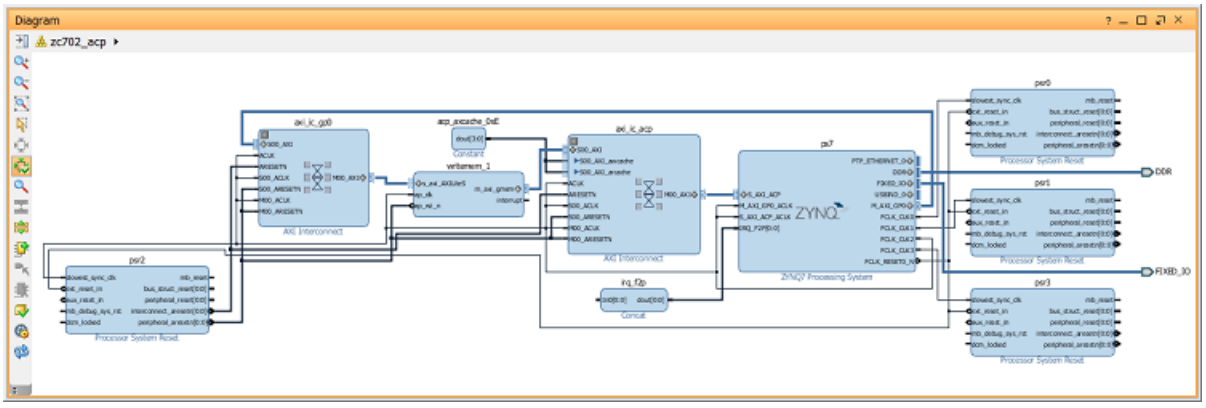
To share an AXI master (slave) interface between a platform IP and the accelerator and data motion IPs generated by the SDSoC compilers, you employ the SDSoC Tcl API to declare the first unused AXI master (slave) port (in index order) on the AXI interconnect IP block connected to the shared interface. 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/platforms/zc702_acp/hw/vivado/zc702_acp.xpr`.

1. The block diagram looks something like the following.

Figure 5: ZC702_acp Block Diagram



2. The hardware platform interface is defined in the file `<sdx_root>/samples/platforms/zc702_acp/zc702_acp_pfm.tcl`. The following commands create a Vivado platform object, give it a name and 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. The following command declares the default platform clock to have id 2. The 'true' argument indicates that this clock is the platform default. Note also the declaration of the associated `proc_sys_reset_2` IP instance that is required of every platform clock.

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

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

```
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
}
```

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 description file.

```
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 C

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.