SDSoC Environment User Guide

Platforms and Libraries

UG1146 (v2016.2) July 13, 2016

XILINX
ALL PROGRAMMABLE™

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
07/13/2016	2016.2	• Updated Direct I/O tutorial in Chapter 4. • Updates to reflect changes to software.
05/11/2016	2016.1	Updates to reflect changes to software.

Table of Contents

XILINX
ALL PROGRAMMABLE™

Chapter 1

Introduction

The SDSoC™ (Software-Defined System On Chip) 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 SDSoC system compiler generates an application-specific system-on-chip by compiling application code written in C/C++ into hardware and software that extends a target platform. The SDSoC environment 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 SDSoC environment targets a specific platform, and you employ the tools within the SDSoC IDE to customize the platform with application-specific hardware accelerators and data motion networks that connect accelerators to the platform. 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

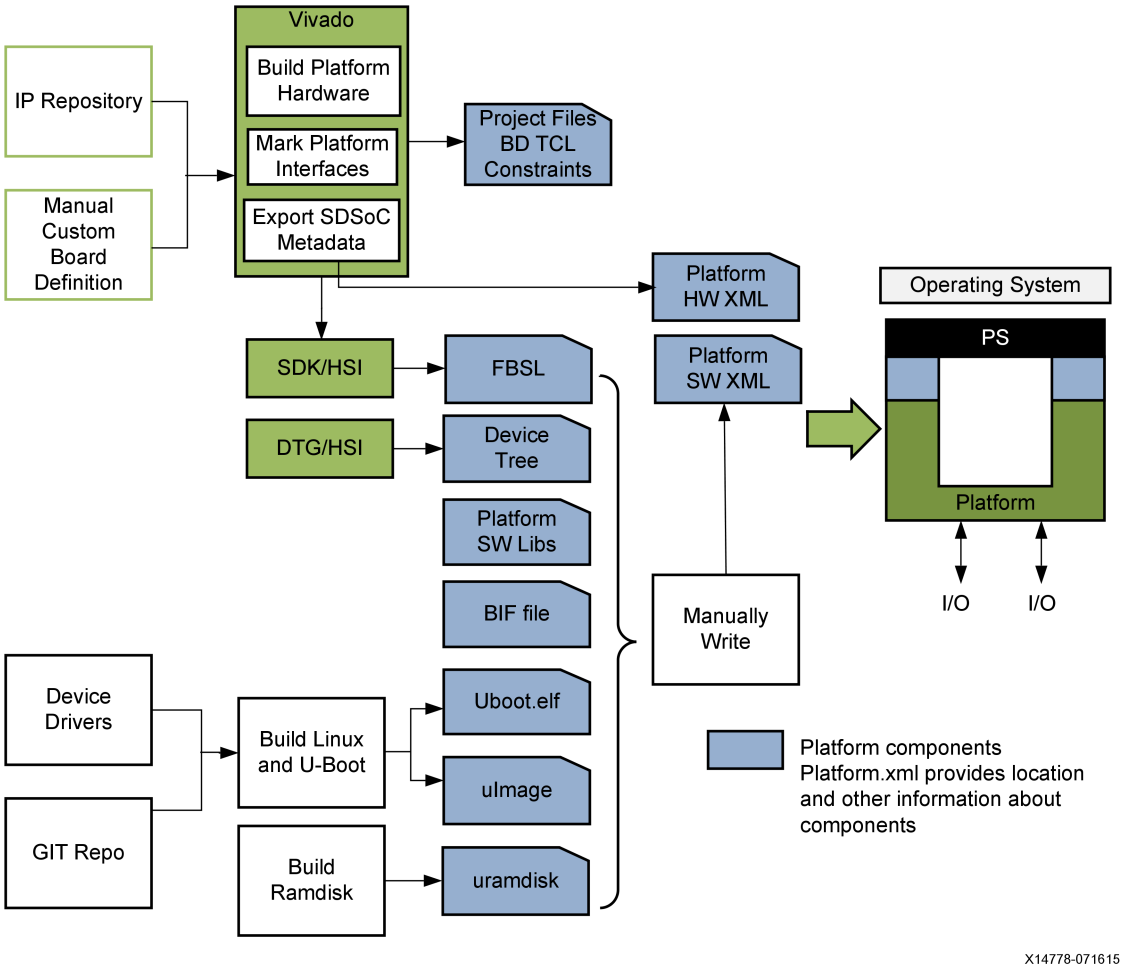
SDSoC Platforms

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 builds 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, SDSoC environment or the Xilinx SDK to build platform libraries. Currently, the software platform metadata file must be created manually.

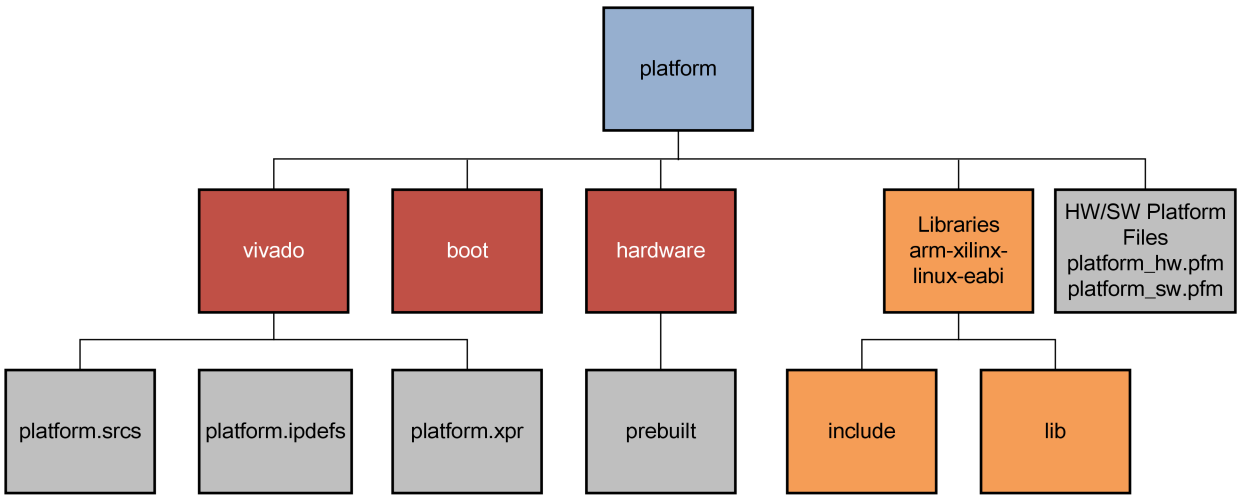
Figure 2–1: **Primary Components of an SDSoC Platform**



An SDSoC platform consists of the following elements:

- Metadata files
  - Platform hardware description file (<platform>_hw.pfm) generated using Vivado tools
  - Platform software description file (<platform>_sw.pfm) written by hand
- Vivado Design Suite project
  - Sources
  - Constraints
  - IP blocks
- Software files
  - Library header files (optional)
  - Static libraries (optional)
  - Linux related objects (device-tree, u-boot, Linux-kernel, ramdisk)
- Pre-built hardware files (optional)
  - Bitstream
  - Exported hardware files for SDK
  - Pre-generated device registration and port information software files
  - Pre-generated hardware and software interface files

Figure 2–2: **Directory Structure for a Typical SDSoC Platform**



In general, only the platform builder can ensure that a platform is "correct" for use within the SDSoC environment. However, you can find a Platform Checklist in `<sdsoc_root>/docs/SDSoC_platform_checklist.xlsx`, which contains an embedded `platform_dm_test.zip` archive containing a basic liveness test for every data mover used by the SDSoC system compiler. Unzip `platform_dm_test.zip` into a work area, and from within an SDSoC environment Terminal shell, execute the following.

```
$ make PLATFORM=<platform_path> axidma_simple
$ make PLATFORM=<platform_path> axidma_sg
$ make PLATFORM=<platform_path> axidma_2d
$ make PLATFORM=<platform_path> axififo
$ make PLATFORM=<platform_path> zero_copy
$ make PLATFORM=<platform_path> xd_adapter
```

Each of these tests should build cleanly, and should be tested on the board.

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.

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 SoC 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. Load the SDSoC Vivado Tcl APIs.
3. Execute Tcl APIs in the Vivado Tcl Console to accomplish the following steps:
    a. Declare the hardware platform name
    b. Declare a brief platform description
    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. Generate the platform hardware description metadata file

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

1. The Vivado project name must match the platform name. If the Vivado project contains more than one block diagram, the block diagram used is the one that has the same name as the platform.
2. Every platform IP that is not part of the standard Vivado IP catalog must be local to the platform Vivado Design Suite project. References to external IP repository paths are not allowed.
3. Every platform must contain a Processing System IP block from the Vivado IP catalog.
4. An SDSoC platform hardware port interface must be an AXI, AXI4-Stream, clock, reset, or interrupt interface only. Custom bus types or hardware interfaces must remain internal to the platform.
5. Every platform must declare at least one general purpose AXI master port from a 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.
6. 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.
7. 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
8. Every platform AXI interface will be connected to a single data motion clock by the SDSoC environment.

   NOTE: Accelerator functions generated by the SDSoC compilers might run on a different clock that is provided by the platform.
9. 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.
10. Every exported platform clock must have an accompanying Processor System Reset IP block from the Vivado IP catalog.
11. Platform interrupt inputs must be exported by a Concat (`xlconcat`) block 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.

Software Requirements

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

The SDSoC environment 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 SDSoC environment 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 SDSoC environment specific drivers which are available with the `linux-xlnx` kernel sources in the `drivers/staging/apf` directory. The base platforms included with the SDSoC environment provide instructions, for example, `platforms/zc702/boot/how-to-build-this-linux-kernel.txt`.

This linux kernel (uImage) and the associated device tree (`devicetree.dtb`) are based on the 4.4 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-v2016.2` tag.

   ```
   git checkout -b sdsoc_release_tag xilinx-v2016.2
   ```

2. Add the following CONFIGs 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. After building it, run the Vivado® hardware export command to create the hardware handoff file. Using this newly generated hardware handoff file, use SDSoC 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 the section Create the Standalone Platform Software. The platform can bundle the libraries and header files in the platform using the procedure listed in the section Platform Software Description File below.

Metadata Files

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

- Platform hardware description file
- Platform software description file

Platform Hardware Description File

A platform hardware description file `<platform>_hw.pfm` is an XML metadata file that describes the hardware system to the SDSoC environment, including available clock frequencies, interrupts, and the hardware interfaces that the SDSoC environment can use to communicate with hardware functions.

As shown in the figure in SDSoC Platforms, you create this file by building a base hardware platform design using the Vivado Design Suite. Using a Vivado Tcl API, you declare the SDSoC Platform port interface and generate the SDSoC platform hardware description. The tutorials in this document provide example usages of these Vivado Tcl APIs.

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 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 Tcl console.

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

2. Execute Tcl APIs in Vivado to accomplish the following steps:
   a. Declare the hardware platform name
   b. Declare a brief platform description
   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. Generate the platform hardware description metadata file

The following describes the TCL API to be used within a block diagram.

Create hardware platform description

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

```
sdsoc::create_pfm <platform>_hw.pfm
```

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_hw.pfm]

sdsoc::pfm_name        $pfm "xilinx.com" "xd" "zc702" "1.0"

sdsoc::pfm_description $pfm "Zynq ZC702 Board"
```

Clocks

You can export any clock source with the platform, but for each 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> <port> <instance> <id> <is_default> <proc_sys_reset>
```

Arguments:

Argument	Description
pfm	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
```

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
```

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
```

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
}
```

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
```

Write 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>
```

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 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
```

Platform Software Description File

As described in SDSoC Platforms, an SDSoC platform has a software component that includes operating system, boot loaders, and libraries. The platform software description file contains metadata about the software runtime needed by the SDSoC system compilers to generate application-specific systems-on-chip built upon a platform.

Boot Files

By default, the SDSoC environment creates an SD card image to boot a board into a Linux prompt or execute a standalone program.

Describe the files for Linux using the following format. If you are using a unified boot image or `.ub` file containing a kernel image, device tree and root file system, specify `xd:linuxImage="boot/image.ub"` while omitting `xd:devicetree` and `xd:ramdisk`. The optional `xd:sdcard` folder contains folders and files that will be added to the root of the SD card image. The optional `xd:sdcardMountPath` specifies the SD card mount path, which defaults to `/mnt` if not specified (in the example below, the PetaLinux mount path `/media/card` is shown).

```
<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/ramdisk.image.gz"
 xd:sdcard="boot/sdcard"
 xd:sdcardMountPath="/media/card"/>
```

For standalone, where no OS is used, the description is:

```
<xd:bootFiles
 xd:os="standalone"
 xd:bif="boot/standalone.bif"
 xd:readme="boot/generic.readme"
 xd:sdcard="boot/sdcard"
/>
```

NOTE: Note that these elements refer to a Boot Image File (BIF). The BIF file must exist in the location specified.

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
 }
```

Library Files

A platform can optionally include libraries. If you describe the library files using the following format, the SDSoC environment automatically adds the appropriate include and library paths (using the -I and -L switches) when calling the compiler.

```
  <xd:libraryFiles
   xd:os="linux"
   xd:includeDir="arm-xilinx-linux-gnueabi/include"
   xd:libDir="arm-xilinx-linux-gnueabi/lib"/>
 <xd:libraryFiles
  xd:os="standalone"
  xd:includeDir="arm-xilinx-eabi/include"
  xd:libDir="arm-xilinx-eabi/lib"
  xd:bspconfig="arm-xilinx-eabi/system.mss"
  xd:bsprepo="arm-xilinx-eabi/bsprepo"/>
```

Description

The informal schema for `xd:libraryFiles` is:

```
<xd:libraryFiles
 xd:os            Operating system. Valid values: linux, standalone
 xd:includeDir    Directory passed to compiler using -I.
                  Separate multiple paths with a colon ':' character.
                  When sdscc/sds++ compiles source files, the include path
                  order is: (1) user paths, (2) platform paths, (3) SDSoC install paths
                  and (4) Vivado HLS paths (if required).
 xd:libDir        Directory paths passed to the linker using -L.
                  Separate multiple paths with a colon ':' character. Each path
                  must be a directory within the platform directory, containing libraries
                  that can be linked with the user application.  The library path
                  link order is: (1) user specified paths, (2) path to SDSoC-generated
                  BSP (standalone/FreeRTOS only), (3) platform paths, (4) SDSoC install
                  paths, and (5) SDSoC-generated project.  Do not place standalone
                  BSP library libxil.a, which sdscc/sds++ generates for a BSP configuration
                  file in a directory on the xd:libDir path.
 xd:libName       Library names passed to the linker using -l.
                  Separate multiple library names with a colon ':' character.
                  When specified, sdscc/sds++ automatically adds the -l option
                  when linking the ELF.
 xd:bspconfig     BSP configuration file (.mss) for standalone/FreeRTOS. When specified,
                  the platform must also specify an xd:includeDir containing BSP header files.
                  sdscc/sds++ uses this .mss instead of generating a default BSP
                  configuration file based on the both the platform and hardware
                  in the 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).
 xd:bsprepo       BSP repository folder. When specified,
                  xd:bspconfig must also be specified.  sdscc/sds++ adds this folder
                  to the BSP repository search path used to create a standalone BSP.
                  Refer to Generating
                  Basic Software Platforms (UG1138) for information
                  about BSP repositories.
/>
```

Pre-Built Hardware Files

A platform can optionally include pre-built hardware files, which the SDSoC environment clones into a project when an application has no hardware functions, rather than rebuilding the bitstream and boot image. This provides fast compilation to run an application software on the target. When a platform provides pre-built hardware files, you can force the bitstream compile using the `sdscc-rebuild-hardware` option to force the creation of hardware files.

The example below describes pre-built hardware included in the ZC702 platform:

```
<xd:hardware
 xd:system="prebuilt"
 xd:bitstream="prebuilt/bitstream.bit"
 xd:export="prebuilt/export"
 xd:hwcf="prebuilt/hwcf"
 xd:swcf="prebuilt/swcf"/>
```

Description

The informal schema for `xd:hardware` is:

```
<xd:hardware
 xd:system           Identifier associated with predefined hardware; when
                     the SDSoC environment searches for a pre-built bitstream, it looks
                     for the keyword "prebuilt"
 xd:bitstream        Path to the bitstream.bit file for the pre-built hardware
 xd:export           Path to the folder containing SDK-compatible files
                     created using the Vivado tools export_hardware command.
                     This folder contains the hardware handoff file <platform>.hdf,
                     for example, zc702.hdf.
 xd:hwcf             Path to the folder containing hardware system
                     information files. Files found in this folder
                     are partitions.xml and apsys_0.xml.
 xd:swcf             Path to the folder containing device registration and
                     port information files. Files found in this folder
                     are devreg.c, devreg.h, portinfo.c and portinfo.h.
 />
```

The pre-built platform files can be created using the SDSoC system compiler by building a "Hello world" program.

Examples are provided for every base platform in `<sdsoc_install_directory>/platforms/*/hardware/prebuilt`.

Testing the Platform Hardware Description File

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

```
sds-pf-check <platform>_hw.pfm
```

After putting the hardware platform description file (`<platform>_hw.pfm`) and software platform description file (`<platform>_sw.pfm`) in the platform directory, you can verify that the SDSoC environment can read the files correctly by executing the following command, which lists all the available platforms. If you see the platform you have created in the displayed list, then the SDSoC environment has found it.

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

To display more information about your platform, use this command:

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

Vivado Design Suite Project

The SDSoC™ environment 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 can be the basis for an SDSoC environment project, there are a few constraints described in Hardware Requirements.

File name and location: `platforms/<platform>/vivado/<platform>.xpr`

Example: `platforms/zc702/vivado/zc702.xpr`

NOTE: You must place the complete project in the same directory as the `xpr` file.

IMPORTANT: *You cannot simply copy the files in a Vivado tools project; the Vivado tools manage internal states in a way that might not be preserved through a simple file copy. To make a project clonable, use the Vivado command* ***File > Archive Project*** *to create a zip archive. Unzip this archive file into the SDSoC platform directory where the hardware platform resides.*

The Vivado tools require ***Upgrade IP*** *for every new version of the Vivado Design Suite. To migrate an SDSoC hardware platform, open the project in the new version of the tools, and then upgrade all IP. Archive the project and then unzip this archive into the SDSoC platform hardware project.*

If you encounter IP Locked errors when the SDSoC environment invokes the Vivado tools, it is a result of failing to make the platform clonable.

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 `xd:includeDir` attribute for the corresponding OS in the platform software description file.

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

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

Example:

For `xd:includeDir="arm-xilinx-linux-gnueabi/include"`:

```
<sdsoc_root>/samples/platforms/zc702_axis_io/arm-xilinx-linux-gnueabi/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

```
xd:includeDir="<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 `xd:libDir` attribute for the corresponding OS in the platform software description file.

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

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

Example:

For `xd:libDir="arm-xilinx-linux-gnueabi/lib"`:

```
<sdsoc_root>/samples/platforms/zc702_axis_io/arm-xilinx-linux-gnueabi/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,

```
xd:libDir="<relative_lib_path1>:<relative_lib_path2>"
```

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

```
<platform_root>/<relative_lib_path1>
<platform root>/<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.*

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 directory. Data in the subdirectory is pointed to by the `xd:bitstream`, `xd:export`, `xd:hwcf`, and `xd:swcf` attributes for the corresponding pre-built hardware.

For a given `xd:bitsteam="<relative_lib_path>/bitstream.bit"` in a platform xml, the location is:

```
platforms/<platform>/<relative_lib_path>/bitstream.bit
```

For a given `xd:export="<relative_export_path>"` in a platform xml, the location is:

```
platforms/<platform>/<relative_export_path>
```

For a given `xd:hwcf="<relative_hwcf_path>"` in a platform xml, the location is:

```
platforms/<platform>/<relative_hwcf_path>
```

For a given `xd:swcf="<relative_swcf_path>"` in a platform xml, the location is:

```
platforms/<platform>/<relative_swcf_path>
```

Example:

For `xd:bitstream="prebuilt/bitstream.bit"`:

```
platforms/zc702/hardware/prebuilt/bitstream.bit
```

For `xd:export="prebuilt/export"`:

```
platforms/zc702/hardware/prebuilt/export
```

contains `zc702.hdf`

For `xd:hwcf="prebuilt/hwcf"`:

```
platforms/zc702/hardware/prebuilt/hwcf
```

containing `partitions.xml` and `apsys_0.xml`.

For `xd:swcf="prebuilt/swcf"`:

```
platforms/zc702/hardware/prebuilt/swcf
```

containing `devreg.c`, `devreg.h`, `portinfo.c` and `portinfo.h`.

Pre-built hardware files are automatically used by the SDSoC 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>/hardware/prebuilt` 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`
- export

  Files found in `_sds/p0/ipi/<platform>.sdk` (`<platform>.hdf`)
- hwcf

  Files found in `_sds/.llvm` (`partitions.xml`, `apsys_0.xml`)
- swcf

  Files found in `_sds/swstubs` (`devreg.c`, `devreg.h`, `portinfo.c`, `portinfo.h`)

Linux Boot Files

The SDSoC™ environment 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 SDSoC environment requires several objects as part of the platform including:

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

The SDSoC environment uses the Xilinx® bootgen utility program to combine the necessary files with the bitstream into a `BOOT.BIN` file in a folder called `sd_card`. The end-user copies the contents of this folder into the root of an SD card to boot the platform.

IMPORTANT: *For detailed instructions on how to build the boot files, refer to the Xilinx Wiki at http://wiki.xilinx.com.*

First Stage Boot Loader (FSBL)

The first stage boot loader 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 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 SDSoC environment flow.

For the SDSoC system compiler to use an FSBL, a BIF file must point to it (see Boot Files). The file must reside in the `<platform_root>/boot/fsbl.elf` folder.

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

Example:

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

U-Boot

Das U-Boot is an open source boot loader. Follow the instructions at wiki.xilinx.com to download U-Boot and configure it for your platform.

For the SDSoC environment to use a U-Boot, a BIF file must point to it (see Boot Files). The file must reside in the `<platform_root>/boot/fsbl.elf` folder.

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

Example: `samples/platforms/zc702_axis_io/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. Follow the device-tree related instructions 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.

The file name and location are defined in the platform xml. Use the `xd:devicetree` attribute in an `xd:bootFiles` element. If you are using a unified boot image (.ub file) containing the kernel, devicetree and root file system, do not define the `xd:devicetree` attribute.

Sample xml description:

```
xd:devicetree="boot/devicetree.dtb"
```

Location: `samples/platforms/zc702_axis_io/boot/devicetree.dtb`

NOTE: When a platform exports a clock sourced by a `processing_system7 FCLK_CLK` port, you must modify the standard PetaLinux-generated device tree to enable the clock at boot time, in order to support hardware debugging with the Vivado Internal Logic Analyzer IP core. Use the following command to modify the device tree for your platform to enable the clocks.

```
fclk-enable = <0xf>;
```

Linux Image

A Linux image is required to boot. 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 to enable the SDSoC environment APF drivers and the Contiguous Memory Allocator (CMA) when configuring Linux for your platform. Linux kernel build instructions for SDSoC platforms are described in `<sdsoc_root>/<platform>/boot/how-to-build-this-linux-kernel.txt`.

The file name and location are defined in the platform xml. Use the `xd:linuxImage` attribute in an `xd:bootFiles` element. If you are using a unified boot image (.ub file) containing the kernel, device tree and root file system, define the `xd:linuxImage` attribute and specify the location of the .ub file, for example `xd:linuxImage="boot/image.ub"`.

Sample xml description:

```
xd:linuxImage="boot/uImage"
```

Location: `samples/platforms/zc702_axis_io/boot/uImage`

Ramdisk Image

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

The file name and location are defined in the platform xml. Use the `xd:ramdisk` attribute in an `xd:bootFiles` element. If you are using a unified boot image (.ub file) containing the kernel, device tree and root file system, do not define the `xd:ramdisk` attribute.

Sample xml description:

```
xd:ramdisk="boot/uramdisk.image.gz"
```

Location: `samples/platforms/zc702_axis_io/boot/uramdisk.image.gz`

Using PetaLinux to Create Linux Boot Files

It is possible to generate all the Linux boot files using PetaLinux as shown in PetaLinux Tools Documentation: Workflow Tutorial (UG1156). The overall workflow while using PetaLinux is the same, but there are a few additional steps for generating Linux boot files for use with the SDSoC environment. Because of this, we have provided a BSP for ZC702 that is configured for use with the SDSoC environment.

If your platform clock sources include an `FCLK_CLK` port from a `processing_system7` IP block, you must modify the device tree as described in Device Tree.

To build a PetaLinux image for ZC702 that can be used with the SDSoC environment, follow these steps.

1. Create a new PetaLinux project using the supplied BSP.

   ```
   $ petalinux-create -t project /path/to/Xilinx-ZC702-SDSoC-2016.2.bsp
   ```

2. Build the project.

   ```
   $ petalinux-build
   ```

3. Wrap the generated kernel and rootfs with a U-Boot header.

   ```
   $ petalinux-package --image -c kernel --format uImage
   ```

4. Rename device tree BLOB and ramdisk.

   ```
   $ mv images/linux/system.dtb images/linux/devicetree.dtb
   $ mv images/linux/urootfs.cpio.gz images/linux/uramdisk.image.gz
   ```

5. The final output products are located under `./images/linux` and are ready to be copied to an SD card.

The ZC702 BSP was derived from the default BSP provided with PetaLinux by making the following changes.

1. Run `petalinux-config -c kernel` to launch the menuconfig system.
   a. Select **Kernel Features > Contiguous Memory Allocator [ON]**.
   b. Select **Device Drivers > Generic Driver Options > DMA Contiguous Memory Allocator [ON]**.
   c. Select **Device Drivers > Generic Driver Options > Size in Mega Bytes [256]**.
   d. Select **Device Drivers > Staging drivers [ON] > Xilinx APF Accelerator driver [ON] > Xilinx APF DMA engines support [ON]**.
2. Open the `<project-root>/subsystems/linux/configs/device-tree/system-top.dts` file and append the following lines of code:

```
&clkc {
   fclk-enable = <0xf>;
};
/ {
   xlnk {
      compatible = "xlnx,xlnk-1.0";
      clock-names = "xclk0", "xclk1", "xclk2", "xclk3";
      clocks = <&clkc 15>, <&clkc 16>, <&clkc 17>, <&clkc 18>;
   };
};
```

3. Run `petalinux-config -c rootfs`, to launch the menuconfig system
4. Select **Filesystem Packages**.
5. Select **base**.
   a. Select **external-xilinx-toolchain > libstdc++6**
   b. Select **tcf-agent > tcf-agent**
6. Run the `petalinux-build` command, which builds the project and generates the file named `image.ub` inside the `<project-root>/images/linux` folder, which has the kernel, device tree, and file system packaged inside.

Standalone Boot Files

If no OS is required, the end-user can create a boot image that automatically executes the generated executable.

First Stage Boot Loader (FSBL)

The first stage boot loader 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 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 SDSoC environment flow.

For the SDSoC system compiler to use an FSBL, a BIF file must point to it (see Boot Files). The file must reside in the `<platform_root>/boot/fsbl.elf` folder.

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

Example:

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

Executable

For the SDSoC environment to use an executable in a boot image, a BIF file must point to it (see Boot Files).

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

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

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. The file that describes the applications to SDSoC 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.

```
<?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 format of the file to be xml and is mandatory:

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

A `<manifest:Manifest>` xml element is required as a container for all application templates:

```
<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

A `<template>` element can have multipe attributes:

Table 2–6: **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 have multiple other xml sub-elements:

Table 2–7: **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 SDSoC 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>
```

Include paths 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 SDSoC adding the flags `-I"../src/myinclude" -I"../src/dir/include"` to the compiler:

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

Library paths 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 SDSoC 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 SDSoC not making a copy of directories `MyDir` and `MyOtherDir` when creating the new project. It will also not make a copy of files `MyFile.txt` and `MyOtherFile.txt`. 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 SDSoC 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 SDSoC 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 SDSoC 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 SDSoC 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 SDSoC Environment C/C++ Build Settings dialog as linker Miscellaneous options.

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

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

Full template.xml Example

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>
```

FreeRTOS Configuration/Version Change

The SDSoC™ environment 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>/SDSoC/<version>/platforms/zc702` to a local folder.

2. To just modify the default linker script, modify the file `<path_to_your_platform>/zc702/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/freertos/include`.
   c. Add the library `libfreertos.a` to `<path_to_your_platform>/zc702/freertos/li`.
   d. Change the paths in `<path_to_your_platform>/zc702/zc702_sw.pfm` for the section containing the line `("xd:os="freertos" (xd:includeDir="freertos/include" and xd:libDir="freertos/lib")`.
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 SDSoC environment folder `<path_to_install>/SDSoC/2016.2/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.

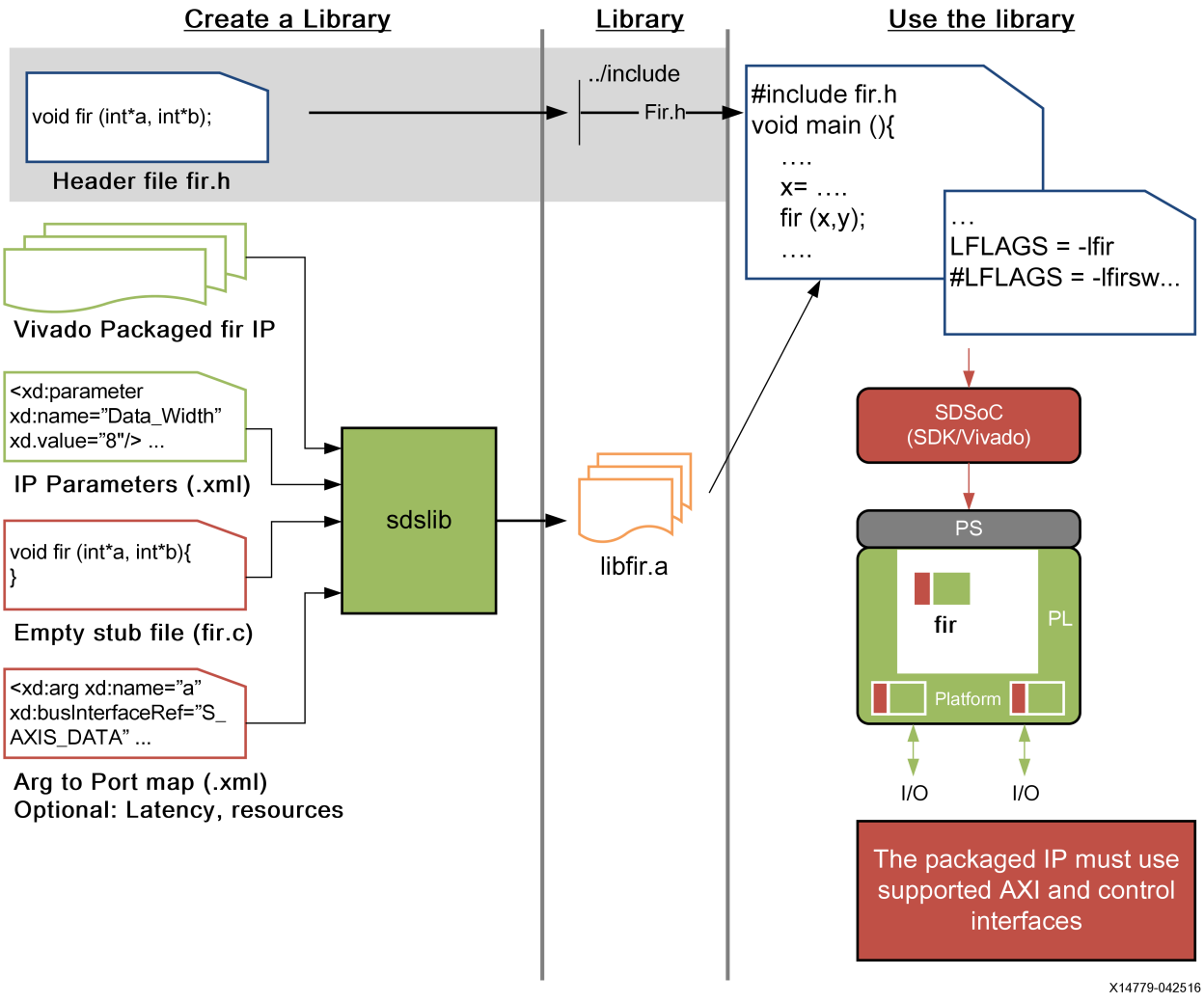
   a. Open a command shell.
   b. Run the SDSoC environment `<path_to_install>/SDSoC/2016.2/settings64` script to set up the environment to run command line tools (including the ARM GNU toolchain for the Zynq®-7000 AP SoC).
   c. Copy the folder to a local folder.
   d. Modify `FreeRTOSConfig.h`.
   e. 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.

C-Callable Libraries

This section describes how to create a C-callable library for IP blocks written in a hardware description language like VHDL or Verilog. User applications can statically link with such libraries using the SDSoC system compilers, and the IP blocks will be instantiated into the generated hardware system. A C-callable library can also provide `sdscc`-compiled applications access to IP blocks within a platform (see Example: Direct I/O in an SDSoC Platform).

Figure 3–1: **Create and Use a C-Callable Library**



The following is the list of elements that are part of an SDSoC platform software callable library:

- Header File

  Function prototype
- Static Library
  - Function definition
  - IP core
  - IP configuration parameters
  - Function argument mapping

Header File

A library must declare function prototypes that map onto the IP block in a header file that can be #included in user application source files. These functions define the function call interface for accessing the IP through software application code.

For example:

```
 // FILE: fir.h
#define N 256
void fir(signed char X[N], short Y[N]);
```

Static Library

An SDSoC environment static library contains several elements that allow a software function to be executed on programmable resources.

Function Definition

The function interface defines the entry points into the library, as a function or set of functions that can be called in user code to target the IP. The function definitions can contain empty function bodies since the SDSoC compilers will replace them with API calls to execute data transfers to/from the IP block. The implementation of these calls depend upon the data motion network created by the SDSoC system compilers.

For example:

```
// FILE: fir.c
#include "fir.h"
#include <stdlib.h>
#include <stdio.h>
void fir(signed char X[N], short Y[N])
{
    // SDSoC replaces function body with API calls for data transfer
}
```

NOTE: Application code that links to the library must also #include stdlib.h and stdio.h, which are required by the API calls in the stubs generated by the SDSoC system compilers.

IP Core

An HDL IP core for a C-callable library must be packaged using the Vivado® tools. This IP core can be located in the Vivado tools IP repository or in any other location. When the library is used, the corresponding IP core is instantiated in the hardware system.

You must package the IP for the Vivado Design Suite as described in the Vivado Design Suite User Guide: Designing with IP (UG896). The Vivado IP Packager tool creates a directory structure for the HDL and other source files, and an IP Definition file (`component.xml`) that conforms to the IEEE-1685 IP-XACT standard. In addition, the packager creates an archive zip file that contains the directory and its contents required by Vivado Design Suite.

The IP can export AXI4, AXI4-Lite, and AXI4 Stream interfaces. The IP control register must exist at address offset `0x0`, and must conform to the following specification, which coincides with the native `axilite` control interface for an IP generated by Vivado HLS.

The control signals are generally self-explanatory. The `ap_start` signal initiates the IP execution, `ap_done` indicates IP task completion, and `ap_ready` indicates that the IP is can be started. For more details, see the Vivado HLS documentation for the `ap_ctrl_hs` bus definition.

```
// 0x00 : Control signals
// bit 0 - ap_start (Read/Write/COH)
// bit 1 - ap_done (Read/COR)
// bit 2 - ap_idle (Read)
// bit 3 - ap_ready (Read)
// bit 7 - auto_restart (Read/Write)
// others - reserved
// (COR = Clear on Read, COH = Clear on Handshake)
```

IMPORTANT: *For details on how to integrate HDL IP into the Vivado Design Suite, see Vivado Design Suite User Guide: Creating and Packaging Custom IP (UG1118).*

IP Configuration Parameters

Most HDL IP cores are customizable at synthesis time. This customization is done through IP parameters that define the IP core's behavior. The SDSoC environment uses this information at the time the core is instantiated in a generated system. This information is captured in an XML file.

The `xd:component` name is the same as the `spirit:component` name, and each `xd:parameter` name must be a parameter name for the IP. To view the parameter names in IP Integrator, right-click on the block and select **Edit IP Meta Data** to access the IP Customization Parameters.

For example:

```
 <!-- FILE: fir.params.xml --
><?xml version="1.0" encoding="UTF-8"?>
<xd:component xmlns:xd="http://www.xilinx.com/xd" xd:name="fir_compiler">
<xd:parameter xd:name="DATA_Has_TLAST" xd:value="Packet_Framing"/>
<xd:parameter xd:name="M_DATA_Has_TREADY" xd:value="true"/>
<xd:parameter xd:name="Coefficient_Width" xd:value="8"/>
<xd:parameter xd:name="Data_Width" xd:value="8"/>
<xd:parameter xd:name="Quantization" xd:value="Integer_Coefficients"/>
<xd:parameter xd:name="Output_Rounding_Mode" xd:value="Full_Precision"/>
<xd:parameter xd:name="CoefficientVector"
 xd:value="6,0,-4,-3,5,6,-6,-13,7,44,64,44,7,-13,-6,6,5,-3,-4,0,6"/></xd:component>
```

Function Argument Map

The SDSoC system compiler requires a mapping from any function prototypes in the library onto the hardware interface defined by the IP block that implements the function. This information is captured in a "function map" XML file.

The information includes the following.

- Function name – the name of the function mapped onto a component
- Component reference – the IP type name from the IP-XACT Vendor-Name-Library-Version identifier.

  If the function is associated with a platform, then the component reference is the platform name. For example, see Example: Direct I/O in an SDSoC Platform.
- C argument name – an address expression for a function argument, for example `x` (pass scalar by value) or `*p` (pass by pointer).

  NOTE: argument names in the function map must be identical to the argument in the function definition, and they must occur in precisely the same order.
- Function argument direction – either `in` (an input argument to the function) or `out` (an output argument to the function). Currently the SDSoC environment does not support `inout` function arguments.
- Bus interface – the name of the IP port corresponding to a function argument. For a platform component, this name is the platform interface `xd:name`, not the actual port name on the corresponding platform IP.
- Port interface type – the corresponding IP port interface type, which currently must be either `aximm` (slave only), `axis`.
- Address offset – hex address, for example, 0x40, required for arguments mapping onto `aximm` slave ports.
- Data width – number of bits per datum.
- Array size – number of elements in an array argument.

The function mapping for a configuration of the Vivado FIR Filter Compiler IP from `samples/fir_lib/build` is shown below.

```
<!-- FILE: fir.fcnmap.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<xd:repository xmlns:xd="http://www.xilinx.com/xd">
    <xd:fcnMap xd:fcnName="fir" xd:componentRef="fir_compiler">
        <xd:arg xd:name="X"
            xd:direction="in"
            xd:portInterfaceType="axis"
            xd:dataWidth="8"
            xd:busInterfaceRef="S_AXIS_DATA"
            xd:arraySize="32"/>
        <xd:arg xd:name="Y"
            xd:direction="out"
            xd:portInterfaceType="axis"
            xd:dataWidth="16"
            xd:busInterfaceRef="M_AXIS_DATA"
            xd:arraySize="32"/>
        <xd:latencyEstimates xd:worst-case="20" xd:average-case="20" xd:best-case="20"/>
        <xd:resourceEstimates xd:BRAM="0" xd:DSP="1 xd:FF="200" xd:LUT="200"/>
    </xd:fcnMap>
</xd:repository>
```

Creating a Library

Xilinx provides a utility called `sdslib` that allows the creation of SDSoC libraries.

Usage

```
sdslib [arguments] [options]
```

Arguments (mandatory)

Argument	Description
`-lib <libname>`	Library name to create or append to
`<function_name file_name>+`	One or more `<function, file>` pairs. For example: `fir fir.c`
`-vlnv <v>:<l>:<n>:<v>`	Use IP core specified by this vlnv. For example, `-vlnv xilinx.com:ip:fir_compiler:7.1`
`-ip-map <file>`	Use specified <file> as IP function map
`-ip-params <file>`	Use specified <file> as IP parameters
`-pfunc`	IP core is a platform function

Option	Description
`-ip-repo <path>`	Add HDL IP repository search path
`-target-os <name>`	Specify target Operating System • linux (default) • standalone (bare-metal)
`--help`	Display this information

As an example, to create an SDSoC library for a `fir filter` IP core, call:

```
> sdslib -lib libfir.a \
    fir fir.c \
    fir_reload fir_reload.c \
    fir_config fir_config.c \
    -vlnv xilinx.com:ip:fir_compiler:7.1 \
    -ip-map fir_compiler.fcnmap.xml \
    -ip-params fir_compiler.params.xml
```

In the above example, `sdslib` uses the functions `fir` (in file `fir.c`), `fir_reload` (in file `fir_reload.c`) and `fir_config` (in file `fir_config.c`) and archives them into the `libfir.a` static library. The `fir_compiler` IP core is specified using `-vlnv` and the function map and IP parameters are specified with `-ip-map` and `-ip-params` respectively.

Testing a Library

To test a library, create a program that uses the library. Include the appropriate header file in your source code. When compiling the code that calls a library function, provide the path to the header file using the `-I` switch.

```
> sdscc -c -I<path to header> -o main.o main.c
```

To link against a library, use the `-L` and `-l` switches.

```
> sdscc -sds-pf zc702 ${OBJECTS} -L<path to library> -lfir -o
 fir.elf
```

In the example above, the compiler uses the library `libfir.a` located at `<path to library>`. See also SDSoC Environment User Guide (UG1027) for testing the library using the SDSoC IDE.

C-Callable Library Example: Vivado FIR Compiler IP

You can find an example on how to build a library in the SDSoC environment installation under the `samples/fir_lib/build` directory. This example employs a single-channel reloadable filter configuration of the FIR Compiler IP within the Vivado® Design Suite. Consistent with the design of the IP, all communication and control is accomplished over AXI4-Stream channels.

You can also find an example on how to use a library in the SDSoC environment installation under the `samples/fir_lib/use` directory. See also SDSoC Environment User Guide (UG1027) for using the library within the SDSoC IDE.

C-Callable Library Example: HDL IP

You can find an example of a Vivado tools-packaged RTL IP in the `samples/rtl_lib/arraycopy/build` directory. This example includes two IP cores, each of which copies M elements of an array from its input to its output, where M is a scalar parameter that can vary with each function call.

- `arraycopy_aximm` - array transfers using an AXI master interface in the IP.
- `arraycopy_axis` - array transfers using AXI4-Stream interfaces.

The register mappings for the IPs are as follows.

```
// arraycopy_aximm
// 0x00 : Control signals
// bit 0 - ap_start (Read/Write/COH)
// bit 1 - ap_done (Read/COR)
// bit 2 - ap_idle (Read)
// bit 3 - ap_ready (Read)
// bit 7 - auto_restart (Read/Write)
// others - reserved
// 0x10 : Data signal of ap_return
// bit 31~0 - ap_return[31:0] (Read)
// 0x18 : Data signal of a
// bit 31~0 - a[31:0] (Read/Write)
// 0x1c : reserved
// 0x20 : Data signal of b
// bit 31~0 - b[31:0] (Read/Write)
// 0x24 : reserved
// 0x28 : Data signal of M
// bit 31~0 - M[31:0] (Read/Write)
// 0x2c : reserved
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH = Clear on Handshake)

// arraycopy_axis
// 0x00 : Control signals
// bit 0 - ap_start (Read/Write/COH)
// bit 1 - ap_done (Read/COR)
// bit 2 - ap_idle (Read)
// bit 3 - ap_ready (Read)
// bit 7 - auto_restart (Read/Write)
// others - reserved
// 0x10 : Data signal of ap_return
// bit 31~0 - ap_return[31:0] (Read)
// 0x18 : Data signal of M
// bit 31~0 - M[31:0] (Read/Write)
// 0x1c : reserved
// (SC = Self Clear, COR = Clear on Read, TOW = Toggle on Write, COH = Clear on Handshake)
```

The makefile indicates how to use `stdlib` to create the library. To build the library, open a terminal shell in the SDSoC IDE, and from within the build directory, run

- `make librtl_arraycopy.a` - to build a library for Linux applications
- `make standalone/lib_rtl_arraycopy.a` - to build a library for standalone applications

A simple test example that employs both IPs is available in the `samples/rtl_lib/arraycopy/use` directory. In an SDSoC terminal shell, run `make` to create a Linux application that exercises both hardware functions.

See also SDSoC Environment User Guide (UG1027) for using the library within the SDSoC IDE.

Tutorial: Creating an SDSoC Platform

SDSoC™ platforms are by definition reusable, allowing you to target many applications to a single base hardware design and software context including bootloaders, operating system, file system, and libraries. This tutorial chapter presents several simple platforms, each designed to demonstrate a useful capability without introducing the complexity of a real-world hardware system. Each of these target the ZC702 board available from Xilinx, but could easily be retargeted to another hardware board. Only the `zc702_led` platform uses board-specific resources (LEDs).

- `zc702_axis_io` - Accessing direct I/O from FPGA pins in an SDSoC platform
- `zc702_led` - Software control of IP cores within a platform
- `zc702_acp` - Sharing an AXI bus interface between the platform and the `sdscc` system compiler

In each example, you first create the base hardware platform using the Vivado® Design Suite, and then export it for use within the SDSoC™ environment. Each example is structured as follows:

- Description of the platform and what it demonstrates
- Generation of the SDSoC hardware platform description
- Creation of platform software libraries, if required
- Creation of the SDSoC software platform description
- Testing the platform

In completing the examples, you make use of information from previous chapters describing the commands used to generate the hardware platform description XML file, the elements of the software platform description XML file, the creation of C-callable libraries, and the platform directory structure.

For example, consider the steps to generate the SDSoC hardware platform description. Recall that a platform hardware description defines a connectivity interface for the design built in the Vivado tools, consisting of AXI4 and AXI4-Stream, clock, reset, and interrupt ports to which the SDSoC environment can connect hardware functions and data mover channels. You declare this connectivity interface from within the Vivado Tcl Console through a set of Tcl APIs described in SDSoC Vivado Tcl Commands according to the following steps.

1. Build and verify the hardware system using the Vivado Design Suite.
2. Open the hardware project in the Vivado Design Suite GUI. (You can also script this process.)
3. Execute Tcl APIs in the Vivado Design Suite to accomplish the following steps:
    a. Declare the hardware platform name.
    b. Declare a brief platform description.
    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. Generate the platform hardware description metadata file.

After working through the platform examples in this tutorial, it would be worthwhile inspecting the platforms that are included in the SDSoC environment in the `<sdsoc_root>/platforms` directory.

Example: Direct I/O in an SDSoC Platform

The SDSoC environment is well-suited to creating hardware accelerator networks that communicate directly with input and output subsystems, e.g., analog-to-digital and digital-to-analog converters, or video I/O. Support for such connectivity is accomplished by converting raw physical data streams into AXI4-Stream interfaces that are exported as part of the platform interface specification. This tutorial example steps through the construction of such a platform. SDSoC Environment User Guide: An Introduction to the SDSoC Environment (UG1028) includes a tutorial using this platform to 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.

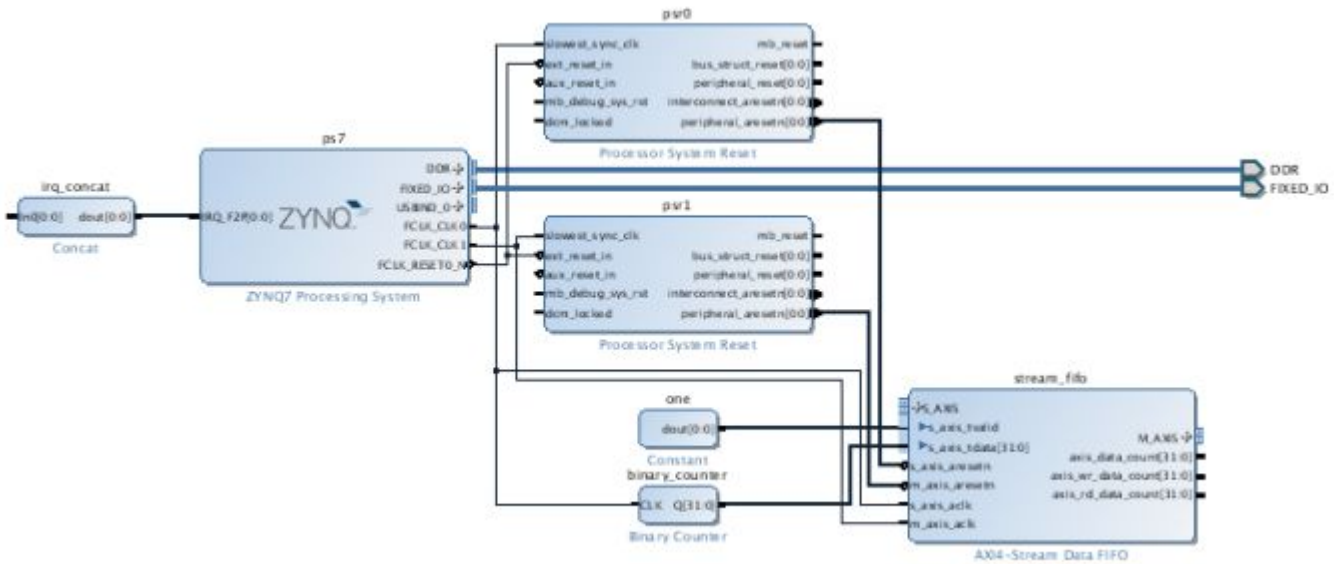
RECOMMENDED: *As you read through this tutorial, you should work through the example provided in `<sdsoc_root>/samples/platforms/zc702_axis_io`. A similar platform targeting an XC7Z010 Zynq-7000 AP SoC is included in `<sdsoc_root>/samples/xc7z010/zybo_axis_io`.*

Open an SDSoC environment terminal shell, copy `<sdsoc_root>/samples/platforms/zc702_axis_io` into a new directory and `cd` into this directory. This platform is fully functional, but in this lab you will reconstruct it. Execute the following commands in the terminal to save the files that you will recreate.

```
mkdir myplatforms

cp -rf <sdsoc_root>/samples/platforms/zc702_axis_io myplatforms

cd myplatforms/zc702_axis_io

mkdir solution

mv zc702_axis_io_hw.pfm solution

mv aarch32-linux/include/zc702_axis_io.h solution

mv aarch32-linux/lib/libzc702_axis_io.a solution
```

The hardware component of an SDSoC platform includes a Vivado project, which you will not recreate in this tutorial. In the terminal, `cd` into the `vivado` subdirectory. Open the Vivado project using the command, `vivado zc702_axis_io.xpr` and open the block diagram.

Figure 4–1: **zc702_axis_io Block Diagram**



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, 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 of course vary according to system requirements.

It is worth pointing out that this data stream is unpacketized, i.e., there is no TLAST signal in the AXI4 stream. This means that any application that consumes the data stream must be capable of handling unpacketized streams. In particular, because all data mover IP cores supported by SDSoC require packetized streams, for 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. One noteworthy aspect of this tutorial example is its demonstration of bubble-free consumption of unpacketized data streams.

RECOMMENDED: *In this release of SDSoC, 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 SDSoC environment platform clocks, you can use the AXI4-Stream Data FIFO IP within the Vivado IP catalog for clock domain crossing.*

Generate the SDSoC Hardware Platform Description

Execute the following steps. For reference, the Tcl commands are also contained in the platform file `vivado/zc702_axis_io_pfm.tcl` (which you have moved into `solution/zc702_axis_io_pfm.tcl`).

1. In the Vivado Tcl console, enter the following command to create a hardware platform object.

```
set pfm [sdsoc::create_pfm zc702_axis_io_hw.pfm]
```

2. Enter the following commands to 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. Enter the following commands to 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 that are required of every platform clock.

```
sdsoc::pfm_clock       $pfm FCLK_CLK2 ps7 0 false  psr0
sdsoc::pfm_clock       $pfm FCLK_CLK2 ps7 1 true  psr1
```

4. Enter the following to 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. Enter the following command to declare 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. Enter the following commands to declare the interrupt inputs:

```
for {set i 0} {$i < 16} {incr i} {

  sdsoc::pfm_irq       $pfm In$i xlconcat

}
```

7. Now that you have declared all of the interfaces, create the SDSoC platform hardware description file zc702_axis_io_hw.pfm in the platform root directory with the following commands.

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

Exit Vivado and from the SDSoC terminal in the vivado directory, validate the generated platform hardware description, move it into the platform root directory, and delete unnecessary project files.

```
sds-pf-check zc702_axis_io_hw.pfm

mv -f zc702_axis_io_hw.pfm ..

rm -rf zc702_axis_io.cache

rm -rf zc702_axis_io.hw

rm -rf zc702_axis_io.runs

rm -rf zc702_axis_io.sdk

rm -rf zc702_axis_io.sim

rm -rf vivado*
```

SDSoC Platform Software Libraries

Every platform IP that exports a direct I/O interface must have a C-callable library that an application can call to connect to the exported interface. In this section you will use the SDSoC `sdslib` utility to create a static C-callable library for the platform as described in Creating a Library.

1. Define the C-callable interfaces for I/O IPs. In the SDSoC tool command shell, `cd` into the `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_stream.cpp`, as follows. The function declaration is included in the file, `zc702_axis_io.h`.

   ```
   void pf_read_stream(unsigned int *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. Define the mapping from the function interface to the respective IP port. 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_hw.pfm` the platform bus interface ("port") named `stream_fifo_M_AXIS` contains the mapping to the IP in the `xd:instanceRef` attribute.*

3. Parameterize the IP.

   IP customization parameters must be set at compile time in an XML file. In this lab, the platform IP has no parameters, so the file `zc702_axis_io.params.xml` is particularly simple. To see a more interesting example, open `<sdsroot>/samples/fir_lib/build/fir_compiler.params.xml` in the SDSoC install tree.

4. Build the library.

   The `sdslib` commands are as follows.

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

Copy `libzc702_axis_io.a` into `../aarch32-linux/lib` and copy `zc702_axis_io.h` into `../aarch32-linux/include`, to make the library available to applications that target the platform.

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 the application on the hardware platform. There is currently no automation for this step.

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

Open the platform software description, `zc702_axis_io_sw.pfm`.

The following element instructs the SDSoC environment where to find the platform software libraries created in the platform directory.

```
<xd:libraryFiles
    xd:os="linux"
    xd:includeDir="aarch32-linux/include"
    xd:libDir="aarch32-linux/lib"
    xd:libName=”zc702_axis_io”
/>
```

Similarly, the boot files are specified as follows:

```
<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"
/>
```

Test the zc702_axis_io Platform

To test the platform, create a new SDSoC project in the IDE and select **Other** for the platform, navigating to the platform location (ensure that the folder and platform names are the same). The platform contains a `samples` subdirectory with several test applications, each of which is of specific interest. The `samples/template.xml` file registers sample applications with the SDSoC IDE.

```
<template location="aximm" name="Unpacketized AXI4-Stream to DDR"
   description="Shows how to copy unpacketized AXI-Stream data directly to DDR.">
    <supports>
        <and>
            <os name="Linux"/>
        </and>
    </supports>
    <accelerator name="copy_data" location="main.cpp"/>
</template>
<template location="stream" name="Packetize an AXI4-Stream"
   description="Shows how to packetize an unpacketized AXI4-Stream.">
    <supports>
        <and>
            <os name="Linux"/>
        </and>
    </supports>
    <accelerator name="packetize" location="packetize.cpp"/>
</template>
<template location="pull_packet" name="Packetization and bubble-free I-O"
   description="Illustrates a technique to enable bubble-free access to a free-running input source.">
    <supports>
        <and>
            <or>
                <os name="Linux"/>
            </or>
        </and>
    </supports>
    <accelerator name="PullPacket" location="main.cpp"/>
</template>
```

1. Create a new project and select `Unpacketized AXI4-Stream to DDR`. 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.
2. From the **C/C++ Build Settings**, add the `zc702_axis_io` library that contains the platform I/O functions to the `-l` libraries linker options.
3. 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 int *bufs[4];
     for (int i=0; i<4; i++)
         bufs[i] = (unsigned int *) sds_alloc(N * sizeof(unsigned int));
     // Flush the platform FIFO of start-up noise
     s2mm_data_copy(bufs[0]);
     for (int i=0; i<4; 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)
     {
         unsigned int tmp[1];
         pf_read_stream(tmp);   // read from platform
         s2mm_data_copy(tmp, buf);   // copy data into output buffer
     }
     ```

4. Build the application. When the build completes, the `SDDebug` folder contains an `sd_card` folder with the boot image and application ELF. Alternatively, to build the project from the command line, in the SDSoC terminal, `cd` into the `samples/arraycopy` directory and execute `make all`.
5. After the build finishes, copy the contents of the `sd_card` directory onto an SD card, boot, and run `zc702_axis_io.elf`.

```
sh-4.3# ./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.

RECOMMENDED: *As you read through this tutorial, you should work through the example provided in `<sdsoc_root>/samples/platforms/zc702_led/`.*

Create SDSoC Platform Hardware Description

Execute the following steps

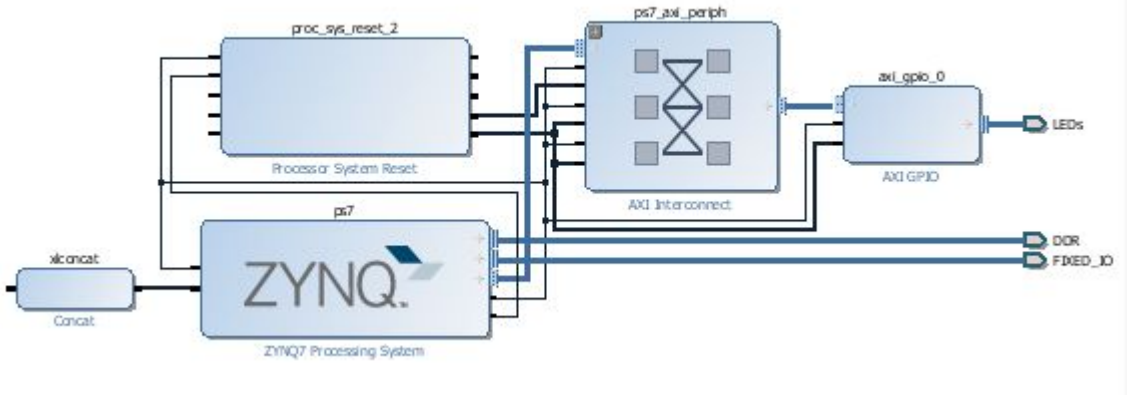
1. Make a local working copy of `samples/platforms/zc702_led`, and from an SDSoC™ environment terminal shell and `cd` into this new directory. This platform is fully functional, but in this lab you will reconstruct it. Execute the following commands in the terminal to save the files that you will recreate (the commands below save the files to a folder named `solution`, which you can compare to the files you create).

```
mkdir myplatforms

cp -rf <sdsoc_root>/samples/platforms/zc702_led myplatforms

cd myplatforms/zc702_led

mkdir solution

mv zc702_led_hw.pfm solution

mv aarch32-linux/include/uio_axi_gpio.h solution

mv aarch32-linux/lib/libzc702_led.a solution

mkdir solution/aarch32-none

mv aarch32-none/include solution/aarch32-none
```

2. `cd` to the `vivado` directory, execute `vivado zc702_led.xpr` and 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 4–3: **zc702_led Block Diagram**



3. In the Vivado Tcl console, enter the following command to load the SDSoC Vivado Tcl APIs and create a hardware platform object.

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

set pfm [sdsoc::create_pfm zc702_led_hw.pfm]
```

4. Enter the following commands to 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_led" "1.0"

sdsoc::pfm_description $pfm "Zynq ZC702 Board With Software Control of Platform IP"
```

5. Enter the following command to declare 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
```

6. Enter the following to 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
```

7. To support Linux applications, the platform provides a userspace library to write to the LEDs built on the Linux UIO framework. In the Tcl commands listed above, the following API call declares the `axi_gpio_0` IP block as a UIO device. This declaration is necessary so that the SDSoC compilers can correctly configure other UIO devices in application code:

```
sdsoc::pfm_iodev       $pfm S_AXI axi_gpio_0 uio
```

8. Enter the following commands to declare the interrupt inputs:

```
for {set i 0} {$i < 16} {incr i} {

  sdsoc::pfm_irq       $pfm In$i xlconcat

}
```

9. Now that you have declared all of the interfaces, create the SDSoC platform hardware description file `zc702_led_hw.pfm` in the platform root directory with the following commands.

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

Exit Vivado and from the SDSoC terminal window in the `vivado` directory, validate the platform hardware description, move the file into the platform directory, and remove unnecessary files.

```
sds-pf-check zc702_led_hw.pfm

mv -f zc702_led_hw.pfm ..

rm -rf zc702_led.cache

rm -rf zc702_led.hw

rm -rf zc702_led.runs

rm -rf zc702_led.sim

rm -rf vivado*
```

Create the zc702_led Platform Software Description

- For standalone applications, an SDSoC platform requires a linker script and header files. From these, the SDSoC compilers create 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 hardware export facility.
  2. Create a hardware platform specification project from the exported hardware system as you would normally do using the Xilinx SDK.
  3. Create a board support package (BSP) project as you would normally do using the Xilinx SDK.
  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.

Create the Standalone Platform Software

Before you create the software component of the SDSoC platform for standalone applications, you must build the hardware component as described in Create SDSoC Platform Hardware Description. In that section, you already ran the Vivado hardware export command (with the block diagram open, use **File > Export > Export Hardware**) to create the hardware handoff file `zc702_led.sdk/zc702_led_wrapper.hdf`. Several steps in this section are executed in a similar manner as in the Xilinx SDK tool.

1. From the SDSoC IDE, select **File > New > Project > Xilinx > Hardware Platform Specification**, selecting the `zc702_led.sdk/zc702_led_wrapper.hdf` handoff file you just created to create a hardware platform project.
2. From the SDSoC IDE, select **File > New > Project > Xilinx > Board Support Project**, selecting the hardware platform project you just created, to create a BSP . When prompted in the pop-up window, add `xilffs` library to your BSP project.
3. In an SDSoC terminal window, copy the header files from your BSP project directory into the SDSoC platform project `zc702_led/arm-xilinx-eabi/include` directory. These header files will be used for the `zc702_led` platform for standalone applications.
4. The BSP project contains a `system.mss` BSP configuration file that specifies driver versions, options, and other settings. Since your BSP was created using default settings, the `.mss` file is not needed in the platform. After generating an application-specific hardware system, `sdscc/sds++` automatically creates an `.mss` file when it generates the BSP and links the application ELF.
5. From the SDSoC IDE, select **File > New > Application Project** to create an application project for this hardware specification and BSP. From the Available Templates list, select `Hello World` to create an application software project.
6. In the SDSoC 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**.

Open the linker script `src/lscript.ld` in a text editor and modify the heapsize as follows:

```
_HEAP_SIZE = DEFINED(_HEAP_SIZE) ? _HEAP_SIZE : 0x8000000;
```

Then copy the linker script into the platform directory `zc702_led/aarch32-none`, and in a text editor, open `zc702_led/zc702_led_sw.pfm` adding the following element.

```
<xd:libraryFiles

  xd:os="standalone"

  xd:includeDir="aarch32-none/include"

  xd:ldscript="aarch32-none/lscript.ld"

/>
```

When the SDSoC compilers compile and link the application, they automatically create a standalone BSP for the platform, linking the application ELF using the linker script you have just created.

Create the Linux Software Platform

IMPORTANT: *This tutorial aims to provide a simple example of software control of memory mapped platform IP in an embedded Linux target platform. It 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.*

Open an SDSoC terminal shell in the `zc702_led/src` directory, and in a text editor, open the `Makefile`. Observe that 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.

To build the library and copy into the platform directory, execute the following in the terminal shell.

```
make
cp -f libzc702_led.a ../aarch32-linux/lib
cp -f uio_axi_gpio.h ../aarch32-linux/include
```

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 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.

To convert the device tree back to binary format required by the Linux kernel, again employ the dtc device tree compiler.

```
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.

Open `zc702_led_sw.pfm` in a text editor and add 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.

Test the zc702_led Platform

To test the platform, create a new project in the SDSoC IDE. Select `Other` for the platform, and navigate to the platform location (ensure that the folder and platform names are the same). The platform contains a samples subdirectory with a single test application called `arraycopy`. The test applications contain 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 SDSoC 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>
```

- To test the Linux platform from the command line, do the following.
  a. In the SDSoC terminal, `cd` into the `samples/arraycopy` directory and execute `make`.
  b. When the build completes, copy the contents of the `sd_card` directory onto an SD card, insert in a ZC702 board and turn on the power. From a serial terminal connected to the board after Linux boots, execute `/mnt/arraycopy.elf`.
- To test the standalone platform from the command line, do the following
  a. in the SDSoC terminal, `cd` into the `samples/arraycopy_sa` directory and execute `make`.
  b. When the build completes, copy the contents of the `sd_card` directory onto an SD card, insert in a ZC702 board, connect a serial terminal to monitor `stdout` and turn on the power.

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.

Recall that a platform hardware description defines a connectivity interface for the design built in the Vivado tools, consisting of AXI4 and AXI4-Stream, clock, reset, and interrupt ports to which the SDSoC environment can connect hardware functions and data mover channels. You declare this connectivity interface from within the Vivado Tcl Console through a set of Tcl APIs described in SDSoC Vivado Tcl Commands according to the following steps.

1. Build and verify the hardware system using the Vivado Design Suite.
2. Open the hardware project in the Vivado Design Suite GUI (note: you can also script this process)
3. Load the SDSoC Vivado Tcl API
4. Execute Tcl APIs in Vivado to accomplish the following steps
    a. Declare the hardware platform name
    b. Declare a brief platform description
    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. Generate the platform hardware description metadata file

RECOMMENDED: *As you read through this tutorial, you should work through the example provided in* `<sdsoc_root>/samples/platforms/zc702_acp/`. *Refer to the* `readme.txt` *file for instructions to build and test the platform.*

Create an SDSoC Platform Hardware Description

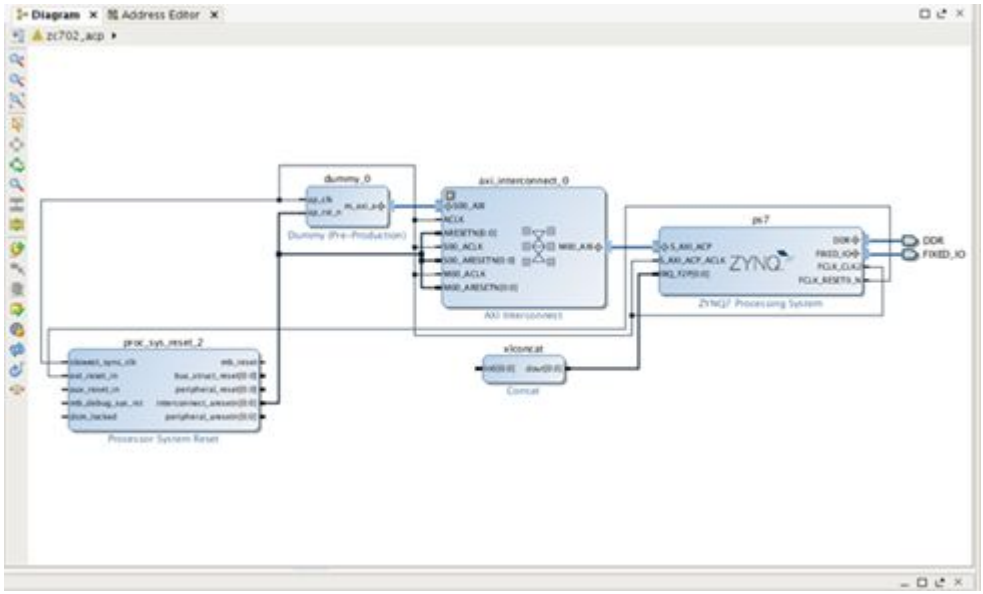
1. Make a local working copy of `samples/platforms/zc702_acp`, and from an SDSoC™ environment terminal shell, `cd` into the `zc702_acp/vivado` directory. This platform is fully functional, but in this lab you will reconstruct it. Execute the following commands in the terminal to save the files that you will recreate.

```
mkdir myplatforms

cp -rf <sdsoc_root>/samples/platforms/zc702_acp myplatforms

cd myplatforms/zc702_acp

mkdir solution

mv zc702_acp_hw.pfm solution
```

2. Open `zc702_acp.xpr` with the Vivado tool, and open the block diagram.

   The block diagram will look something like the following.

Figure 4–4: **ZC702_acp Block Diagram**



3. In the Vivado Tcl console, enter the following command to load the SDSoC Vivado Tcl APIs: and create a hardware platform object.

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

set pfm [sdsoc::create_pfm zc702_acp_hw.pfm]
```

4. Enter the following commands to declare the platform name and provide a brief description that will be displayed when a user executes `sdscc -sds-pf-info zc702_acp`.

```
sdsoc::pfm_name        $pfm "xilinx.com" "xd" "zc702_acp" "1.0"

sdsoc::pfm_description $pfm "Zynq ZC702 board with a shared ACP port"
```

5. Enter the following command to declare 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
```

6. Enter the following to 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 API call for the `S01_AXI` port declares the interconnect port as part of the platform interface, providing access to a hardware coherent `S_AXI_ACP` port on the `processing_system7` IP block within the platform. Observe in the Vivado block diagram that the `s00_AXI` port (i.e., the least significant indexed port(s)) is used 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

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

7. Enter the following commands to declare the interrupt inputs:

```
for {set i 0} {$i < 16} {incr i} {

  sdsoc::pfm_irq       $pfm In$i xlconcat

}
```

8. Now that you have declared all of the interfaces, create the SDSoC platform hardware description file `zc702_acp_hw.pfm` in the platform root directory with the following commands.

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

Exit Vivado and from the SDSoC terminal in the `vivado` directory, validate the hardware description, move the file into the platform directory, and remove unnecessary project files.

```
sds-pf-check zc702_acp_hw.pfm

mv -f zc702_acp_hw.pfm ..

rm -rf zc702_acp.cache

rm -rf zc702_acp.hw

rm -rf zc702_acp.runs

rm -rf zc702_acp.sdk

rm -rf zc702_acp.sim

rm -rf vivado*
```

Test the zc702_acp Platform

To test the platform, create a new SDSoC project in the IDE and select `Other` for the platform, navigating to the platform location (ensure that the folder and platform names are the same). The platform contains a samples subdirectory with a single test application call `Arraycopy`. The `template.xml` file in this directory registers sample applications with the SDSoC IDE.

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

To test the platform in the SDSoC GUI, execute the following steps.

1. Create a new SDSoC project named `zc702_acp` by selecting **File > New > SDSoC Project**. To select the platform, click the **Other** button and navigate to `<sdsoc_root>/samples/platforms/zc702_acp`. Click **Next**.
2. Select the **Array Copy** template and click **Finish**.
3. In the Project Explorer, expand `zc702_acp/src/arraycopy.cpp`, right-click on `arraycopy` function and **Toggle HW/SW** to select the function for hardware.
4. Build the project, and when the build completes, copy the contents of the `SDDebug/sd_card` directory into an SD card
5. Boot the ZC702 board and execute the following:

```
$ /mnt/zc702_acp.elf
```

To test the platform from the command line, in the SDSoC terminal, `cd` into the `samples/arraycopy` directory.

1. To test the platform, from the terminal shell run: `make`
2. The test application contains a simple arraycopy hardware function using the SDSoC environment `axi_dma_simple` datamover. Load the contents of `samples/arraycopy/sd_card` into an SD card and boot.

```
$ /mnt/zc702_acp.elf
```

This example demonstrates how Processing System 7 IP ports can be shared between a platform and SDSoC environment generated logic.

XILINX
ALL PROGRAMMABLE™

Appendix A

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. *SDSoC Environment User Guide: An Introduction to the SDSoC Environment* (UG1028), also available in the docs folder of the SDSoC environment.
2. *SDSoC Environment User Guide* (UG1027), also available in the docs folder of the SDSoC environment.
3. *SDSoC Environment User Guide: Platforms and Libraries* (UG1146), also available in the docs folder of the SDSoC environment.
4. *UltraFast Embedded Design Methodology Guide* (UG1046)
5. *ZC702 Evaluation Board for the Zynq-7000 XC7Z020 All Programmable SoC User Guide* (UG850)
6. *Vivado Design Suite User Guide: High-Level Synthesis* (UG902)
7. *PetaLinux Tools Documentation: Workflow Tutorial* (UG1156)
8. Vivado® Design Suite Documentation
9. *Vivado Design Suite User Guide: Creating and Packaging Custom IP* (UG1118)

Please Read: Important Legal Notices