SDSoC Environment User Guide

Platforms and Libraries

UG1146 (v2015.2) July 20, 2015

XILINX
ALL PROGRAMMABLE™

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
07/20/2015	2015.2	First version of the document.

Table of Contents

Revision History 2
Table of Contents 3
Chapter 1: Introduction 4
Chapter 2: SDSoC Platforms 5
Hardware Requirements 7
Software Requirements 8
Metadata Files 9
Vivado Design Suite Project 16
Library Header Files 16
Pre-built Hardware 17
Linux Boot Files 19
Using Petalinux to Create Linux Boot Files 21
Standalone Boot Files 22
FreeRTOS Configuration/Version Change 23
Chapter 3: C-Callable Libraries 25
Header File 26
Static Library 26
Creating a Library 29
Testing a Library 30
C-Callable Library Example: Vivado FIR Compiler IP 31
C-Callable Library Example: HDL IP 32
Chapter 4: Tutorial: Creating an SDSoC Platform 33
Example: Exporting Direct I/O in an SDSoC Platform 33
Example: Software Control of Platform IP 39
Example: Sharing a Processing System 7 IP AXI Port 42
Appendix A: Additional Resources and Legal Notices 46
Xilinx Resources 46
Solution Centers 46
References 46
Please Read: Important Legal Notices 47

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. The SDSoC system compiler generates an application-specific system on chip from application code written in C/C++, by extending a target platform. The SDSoC environment includes a number of platforms for application development and others 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 connecting 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.

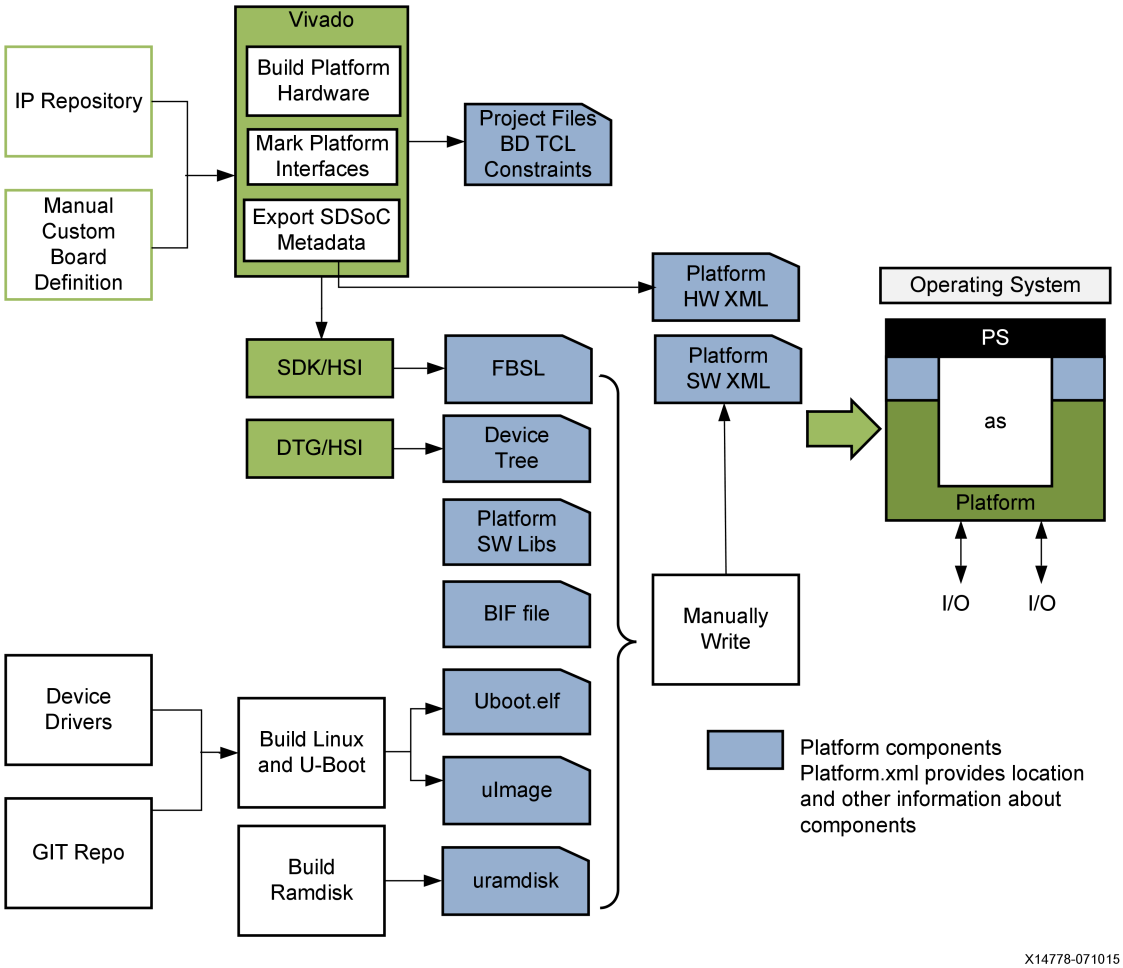
This document describes how to create a custom SDSoC platform from a hardware system built using the Vivado® Design Suite, and a software run-time environment.

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

SDSoC Platforms

An SDSoC platform consists of a Vivado® Design Suite hardware project, an operating system, boot files, and optional software libraries. In addition, an SDSoC platform includes XML metadata files that describe the hardware and software interfaces. A platform provider uses Vivado Design Suite and IP integrator to create the platform hardware, and Tcl commands within the Vivado tools to tag SDSoC environment hardware interfaces. The HSI utility generates the hardware metadata file. The platform creator must also provide boot loaders and operating system required to boot the platform and, if desired, optional software libraries that can be linked by SDSoC environment applications. Currently, the software platform metadata file must be created manually.

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

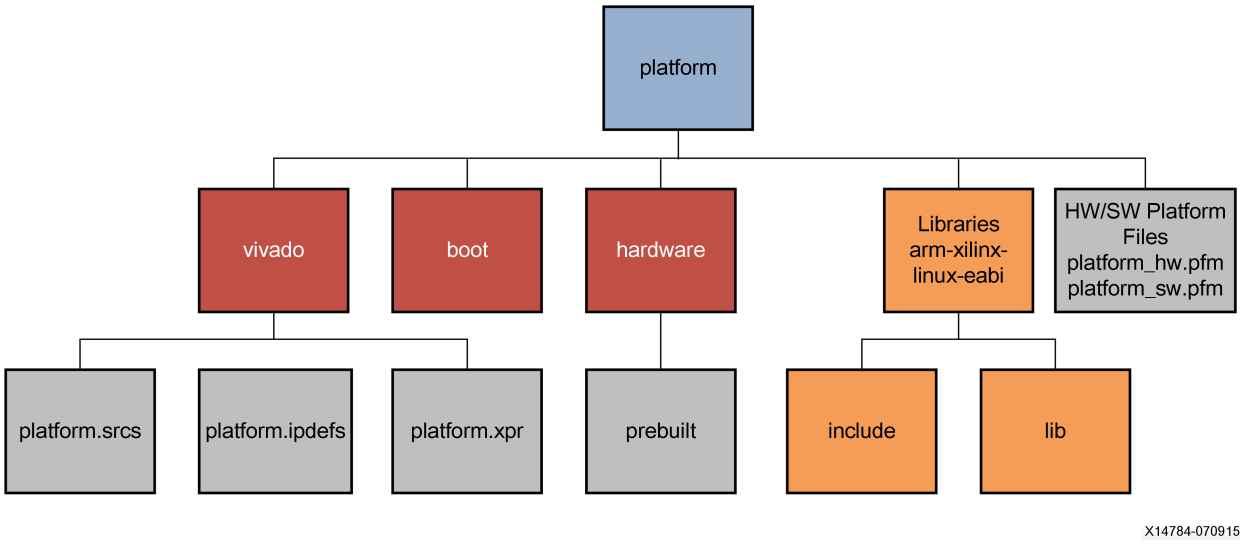


As shown, the hardware component of an SDSoC platform is built using the Vivado Design Suite. If a platform supports a target Linux operating system, you can build the kernel and U-boot bootloader at the command line or using the PetaLinux tool suite. You can use the PetaLinux tools, SDSoC environment or the Xilinx SDK to build platform libraries.

An SDSoC platform consists of the following elements:

- Metadata files
  - Platform hardware description file (`<platform>_hw.pfm`)
  - Platform software description file (`<platform>_sw.pfm`)
- 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 many of the guidelines enumerated in this document.

In addition, this spreadsheet contains an embedded `platform_dm_test.zip` file with a basic liveness test for every SDSoC environment data mover. Unzip `platform_dm_test.zip` into a workarea, 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.

In addition, a platform should provide tests for every custom interface so that users have examples of how to access these interfaces.

Hardware Requirements

This section describes requirements on the hardware design component of an SDSoC platform.

A Zynq®-7000 All Programmable SoC hardware system must be built in the Vivado® Design Suite using the IP integrator tool. There are several rules that the hardware design must observe.

- Any platform IP that is not part of the standard Vivado IP catalog must be local to the Vivado Design Suite project. References to external IP repository paths are not allowed.
- An SDSoC platform hardware port interface consists of AXI, AXI4-Stream, clock, reset, and interrupt interfaces only. Any custom bus types or hardware interfaces must remain internal to the platform.

  All AXI and AXI4-Stream interfaces are connected to a single data motion clock by the SDSoC environment, so the platform must handle all clock-domain crossing as needed.
- AXI4-Stream interfaces require `TLAST`, `TKEEP` sideband signals to comply with the Vivado tools IP employed by SDSoC environment data movers.
- Every platform must contain a Processing System 7 IP block from the Vivado IP catalog.

  Every unused AXI port on the Processing System 7 IP block automatically becomes part of the exported SDSoC platform interface
- 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.
- If a platform exports an unterminated AXI port on an interconnect, every SDSoC environment application must use this port or the SDSoC system compiler issues an error when invoking the Vivado Design Suite tools. The SDSoC environment automatically terminates AXI ports only on the Processing System 7 IP block or on an AXI Interconnect block, by setting IP port enablement parameters in the platform hardware XML file as described in Platform Hardware Description File.
- Every platform must export at least one general purpose master AXI port from the Processing System 7 IP. If a platform requires both `M_AXI_GP0` and `M_AXI_GP1`, then it must export a master port on the AXI Interconnect IP block connected to one or both of them.
- SDSoC environment exported reset signals must be driven by a Processor System Reset IP block from the Vivado IP catalog, synchronized to an exported SDSoC environment clock signal. Each clock in the platform interface must include the associated `proc_sys_reset`, `peripheral_reset`, `peripheral_aresetn`, and `interconnect_reset` ports in the platform interface.
- 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. The SDSoC platform interrupt interface consists of any remaining unused interrupts. If a platform does not use any interrupts, the Concat block has an unterminated input, but the SDSoC environment automatically terminates inputs to the interrupt Concat block as needed.

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 does not have to support all of them.

If 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 3.17 version of the linux kernel
To build the kernel:

Clone/pull from the master branch of the Xilinx/linux-xlnx tree at github,
and checkout the xlnx_3.17 branch
 git checkout xlnx_3.17

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=y
CONFIG_CMA_SIZE_MBYTES=256
CONFIG_LOCALVERSION="-xilinx-apf"
# The following configs are optional, and remove some debug settings
CONFIG_PRINTK_TIME=n
CONFIG_LOCKUP_DETECTOR=n
CONFIG_DEBUG_RT_MUTEXES=n
CONFIG_DEBUG_WW_MUTEX_SLOWPATH=n
CONFIG_PROVE_LOCKING=n
CONFIG_DEBUG_ATOMIC_SLEEP=n
CONFIG_PROVE_RCU=n
CONFIG_DMA_API_DEBUG=n
One way to do this is to
 cp arch/arm/configs/xilinx_zynq_defconfig arch/arm/configs/tmp_defconfig
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

Build the kernel using
make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi- UIMAGE_LOADADDR=0x8000 uImage
```

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

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, setting attributes on blocks, block parameters, and block ports in the IP integrator block diagram using Tcl APIs in the Tcl Console (or in a script if using a batch flow) and invoking the HSI utility to generate the SDSoC platform hardware description. You can also create and edit the file manually in a text editor.

The SDSoC platform port interface consists of a set of available unused AXI or AXI4-Stream bus interfaces on platform IP within an IP integrator block diagram, a set of unused interrupts on the Processing System 7 IP block (`processing_system7`), clock ports, and synchronized resets provided by `proc_sys_reset` IP from the Vivado IP catalog.

The `processing_system7` IP block in the Vivado IP catalog receives special treatment; every unused AXI interface becomes part of the SDSoC environment interface. For all other IP in the IP integrator system, the platform builder must explicitly set a Tcl property on an AXI or AXI4-Stream interface in order for the interface to be included in the SDSoC environment platform interface.

Vivado IP Integrator Tcl Commands

This section describes the Vivado® IP integrator Tcl commands to specify the hardware interface of an SDSoC™ platform.

The following command is required to enable IP integrator export of the SDSoC environment hardware platform description.

```
set_param project.enablePlatformHandoff true
```

File name and location: `<platform>/<platform>_hw.pfm`

Example: `platforms/zc702/zc702_hw.pfm`

The following describes the attributes that can be applied within a block diagram using Vivado Design Suite Tcl APIs.

To declare a platform port and bus interface respectively, use:

```
set_property SDSOC_PFM.MARK_SDSOC [get_bd_pins <pin>]
set_property SDSOC_PFM.MARK_SDSOC [get_bd_intf_pins <bus_interface>]
```

To declare the default platform clock source, use:

```
set_property SDSOC_PFM.PFM_CLOCK TRUE [get_bd_pins /ps7/FCLK_CLK2]
```

To declare a platform clock ID (non-negative integer), use:

```
set_property SDSOC_PFM.CLOCK_ID 2 [get_bd_pins /ps7/FCLK_CLK2]
```

To declare an instance to be a Linux UIO platform device, use:

```
set_property SDSOC_PFM.UIO TRUE [get_bd_cells /axi_gpio_0]
```

To declare parameters for a platform IP instance, use:

```
set_property SDSOC_PFM.PFM_PARAMS {NUM_SI} [get_bd_cells /axi_interconnect_s_axi_acp]
```

The list within the {} should be a colon-separated list of parameter names for the platform IP instance, in this case `axi_interconnect_s_axi_acp`.

To declare a parameter for a platform IP instance, use:

```
set_property SDSOC_PFM.NUM_SI
{count($designComponent/xd:connection/xd:busInterface[@xd:instanceRef=$instance
 and @xd:name='axi_interconnect_s_axi_acp_S00_AXI'])+1}
[get_bd_cells/axi_interconnect_s_axi_acp]
[get_bd_cells /axi_interconnect_s_axi_acp]
```

The parameter must be listed in SDSOC_PARAMS list for this instance. The additive "+1" term in the expression reflects the number of interconnect ports currently used within the platform, in this case, one.

IMPORTANT: *In this release of the SDSoC environment, the exported hardware description file `<platform>_hw.pfm` might be missing some required metadata, which you must add manually. Inspect the `vivado.log` file or the output of the Tcl Console in the Vivado tools for* unknown attribute *warnings, which indicate that a particular attribute is not yet supported.*

Known issues include:

- Platform AXI stream bus interfaces containing `TLAST`, `TKEEP` sideband signals must have the `xd:hasTlast` attribute (with value true) in the corresponding `xd:busInterface` element, but these are not automatically detected. You must add these attributes manually (see Example: Software Control of Platform IP).
- Platform AXI interfaces on the `axi_interconnect` IP block are missing the following attributes, which must be added manually (see Example: Sharing a Processing System 7 IP AXI Port).
  - `xd:coherent` – required for an interconnect interface that is connected to the S_AXI_ACP, optional otherwise
  - `xd:numBits` - number of available ID bits for this interface. Each AXI interface has a fixed number of ID bits, and each interconnect reserves `ceil(log2(#interfaces))` bits. Because platform interfaces are shared with the SDSoC environment by cascading `axi_interconnects`, the SDSoC environment requires this number to avoid overloading an interface.
  - `xd:memport` – tag an `axi_interconnect` bus interface as a channel to external memory through a Vivado tools MIG IP, Processing System 7 M_AXI_GP[01], S_AXI_ACP, or S_AXI_HP[0-3] bus interface, with values selected from one of ["MIG"|"M_AXI_GP"|"S_AXI_ACP"|"S_AXI_HP"] tags an interface as a channel to external memory.

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 as described in Resets.

The following Tcl command declares the `FCLK_CLK1` port on the Processing System 7 IP instance to be part of the SDSoC environment interface.

```
set_property SDSOC_PFM.MARK_SDSOC true [get_bd_pins /ps7/FCLK_CLK1]
```

You must specify a non-negative integer valued ID for each clock with the following Tcl command.

```
set_property SDSOC_PFM.PFM_CLOCK_ID <id> [get_bd_pins <clock port>]
```

For example, the following Tcl command sets the ID for `FCLK_CLK1` to be 1.

```
set_property SDSOC_PFM.CLOCK_ID 1 [get_bd_pins /ps7/FCLK_CLK1]
```

For any clock source that is not a `processing_system7` port, you must manually add to the generated XML hardware description file the following element. For example, to export the `clk_out1` port for the `instance clkwiz_0` insert the following in the platform hardware description file.

```
<xd:busInterface
 xd:busInterfaceRef="clk_out1"
 xd:busTypeRef="clock"
 xd:instanceRef="clkwiz_0"
 xd:mode="master"
 xd:name="clkwiz_0_clk_out1"/>
```

Every platform must declare a default clock for the SDSoC environment to use when no explicit clock has been specified. For example, the following Tcl command declares the `FCLK_CLK2` port on the Processing System 7 IP block as the default platform clock.

```
set_property SDSOC_PFM.CLOCK TRUE [get_bd_pins /ps7/FCLK_CLK2]
```

If the clock ID for `FCLK_CLK2` is 2, the XML element for that captures platform default clock is `<xd:systemClocks xd:defaultClock="2">`. Forgetting to declare a default platform clock leads the SDSoC system compiler to issue errors during system generation.

Resets

The SDSoC™ environment requires a platform to synchronize resets to specific clocks using the Vivado® Design Suite `proc_sys_reset` IP and to export the reset interfaces with the following commands:

```
set_property SDSOC_PFM.MARK_SDSOC true [get_bd_pins <proc_sys_reset>/interconnect_aresetn]
set_property SDSOC_PFM.MARK_SDSOC true [get_bd_pins <proc_sys_reset>/peripheral_aresetn]
set_property SDSOC_PFM.MARK_SDSOC true [get_bd_pins <proc_sys_reset>/peripheral_reset]
```

Interrupts

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. You do not need to declare interrupts explicitly with Tcl APIs; the Vivado® tools automatically export available interrupts as part of the SDSoC platform interface.

Creating SDSoC Platform Hardware Description File

To export the system through the Vivado IDE or using Tcl:

```
write_hwdef -file "[file join <platform>.sdk <platform>_wrapper.hdf]"
```

To generate the platform description file from the Vivado tools Tcl Console, invoke the Vivado tools Hardware/Software Interface (HSI) utility as follows.

```
load_features hsi
hsi::open_hw_design <platform>.sdk/<platform>_wrapper.hdf
hsi::generate_target {sdsoc} [hsi::current_hw_design] -dir <target_directory>
```

This generates the hardware platform description file in `<target_directory>`, along with a journal Tcl file that contains the Tcl commands you invoked to declare the platform.

IMPORTANT: *`hsi` generates a file called `<platform>.pfm`, which you must rename to `<platform>_hw.pfm`.*

IMPORTANT: *As described in Vivado IP Integrator Tcl Commands, you might need to add additional attributes to the hardware description file. See Tutorial: Creating an SDSoC Platform, for examples.*

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, devicetree and root file system, specify `xd:linuxImage="boot/image.ub"` while omitting `xd:devicetree` and `xd:ramdisk`).

```
<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"/>
```

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"
/>
```

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>
}
```

The SDSoC system compiler inflates this template during system generation to create an application-specific BIF file it passes 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>
 }
```

The SDSoC system compiler inflates this template during system generation to create an application-specific BIF file it passes 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 might 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"/>
```

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.
 xd:libDir        Directory passed to the linker using -L .
                  Separate multiple paths with a colon ':' character.
 xd:libName       Library name passed to the linker using -l.
                  Separate multiple library names with a colon ':' character.
                  If this is specified, sdscc/sds++ automatically adds the -l option
                  when linking the ELF, otherwise the user adds the -l option.
/>
```

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 XML File

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_hdmi/vivado/zc702_hdmi.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_hdmi/arm-xilinx-linux-gnueabi/include/zc702hdmi/hwi_export.h
```

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

```
#include "zc702hdmi/hwi_export.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_hdmi/arm-xilinx-linux-gnueabi/lib/libzc702hdmi.a
```

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

```
-lzc702hdmi
```

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 employed 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 a board into a Linux prompt and execute 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® IP integrator, click the **File > Export > Export_to_SDK** menu options to export the Hardware to Xilinx® SDK and then open up Xilinx SDK. Using this hardware platform, select the new project menu in Xilinx SDK to create a new Xilinx application, and then select the FSBL application from the list. This creates an FSBL executable.

For more detailed information on using the Xilinx SDK, see the SDK Help System.

When the platform provider generates the FSBL through Xilinx SDK, they 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_hdmi/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_hdmi/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 devicetree 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: `platforms/zc702_hdmi/boot/devicetree.dtb`

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 `<sds_install_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, devicetree 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: `platforms/zc702_hdmi/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, devicetree and root file system, do not define the `xd:ramdisk` attribute.

Sample xml description:

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

Location: `platforms/zc702_hdmi/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 a few additional steps for generating Linux boot files for use with the SDSoC environment.

1. Open the `<project-root>/subsystems/linux/config` file and append the following lines of code:

```
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-xilinx-linux-gnueabi-"
CONFIG_LOCALVERSION="-xilinx-apf"
CONFIG_PRINTK_TIME=n
CONFIG_DEBUG_KERNEL=n
CONFIG_HAVE_DEBUG_KMEMLEAK=n
CONFIG_LOCKUP_DETECTOR=n
CONFIG_DEBUG_RT_MUTEXES=n
CONFIG_DEBUG_WW_MUTEX_SLOWPATH=n
CONFIG_PROVE_LOCKING=n
CONFIG_DEBUG_ATOMIC_SLEEP=n
CONFIG_PROVE_RCU=n
CONFIG_DMA_API_DEBUG=n
```

2. Open the `<project-root>/subsystems/linux/configs/device-tree/system-top.dts` file and append the following lines of code:

```
&clkc {
 fclk-enable = <0x0>;
};
/ {
 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, devicetree, 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® IP integrator, click the **File > Export > Export_to_SDK** menu options to export the Hardware to Xilinx® SDK and then open up Xilinx SDK. Using this hardware platform, select the new project menu in Xilinx SDK to create a new Xilinx application, and then select the FSBL application from the list. This creates an FSBL executable.

For more detailed information on using the Xilinx SDK, see the SDK Help System.

When the platform provider generates the FSBL through Xilinx SDK, they 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_hdmi/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.

FreeRTOS Configuration/Version Change

The SDSoC™ environment FreeRTOS support uses a pre-built library using the default `FreeRTOSConfig.h` file included with the v8.2.1 software distribution, along with a predefined linker script.

To change the FreeRTOS v8.2.1 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/2015.2/tps/FreeRTOS` includes the source files used to build the pre-configured FreeRTOS v8.2.1 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.

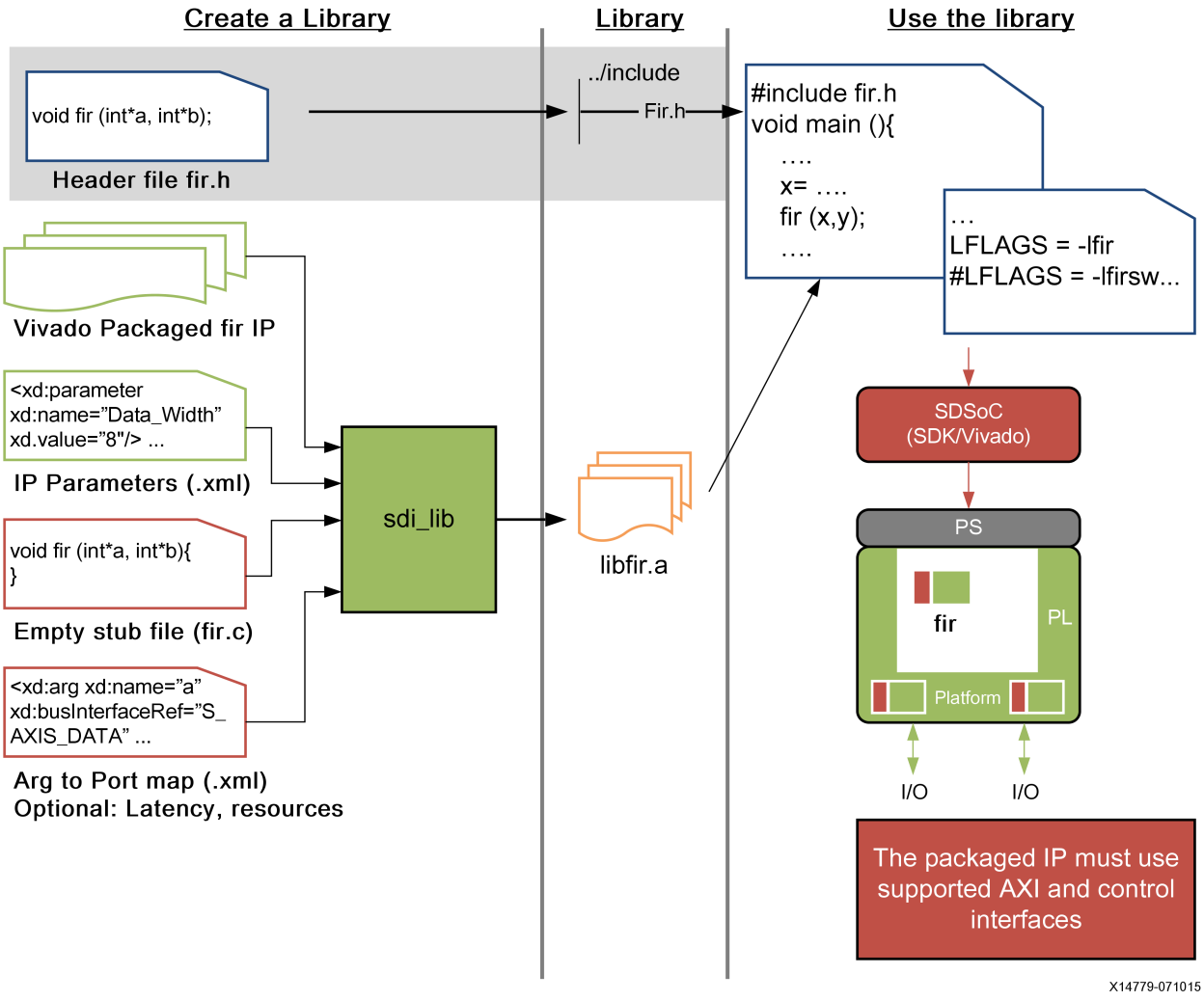
   a. Open a command shell.
   b. Run the SDSoC environment `<path_to_install>/SDSoC/2015.2/settings64` script to set up the environment to run command line tools (including the ARM GNU toolchain for the Zynq®-7000 AP SoC).
   c. Copy the folder to a local folder.
   d. Modify `FreeRTOSConfig.h`.
   e. Run the make command.

If you are not using FreeRTOS v8.2.1, 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 v8.2.1 library can be used.

C-Callable Libraries

This section describes how to create a C-callable library that enables the SDSoC system compiler to link application programs to function libraries that are implemented in programmable logic with IP blocks written in hardware description languages like VHDL or Verilog, like any other static library. A C-callable library can also provide `sdscc`-compiled applications access to IP blocks within a platform (see Example: Exporting 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

The function prototype in provides in a header file as it is done for any C/C++ static library. The header file defines the function interface. This header file is to be included in the user's source code.

For example:

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

Where array X is the filter input and Y is the filter output.

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 an entry point into the library. This is the function (or set of functions) that can be called from the user code.

The contents of the function are not required because the SDSoC environment replaces the function body with API calls that execute the data transfer to/from the IP block. These calls are dependent on the data motion network created by the SDSoC environment.

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: The API calls used by the SDSoC environment require the use of `stdlib.h` and `stdio.h` to be included in this file.

IP Core

The IP core is an HDL IP compatible with the Vivado® tools that is used as a hardware implementation for the library function or functions. 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. If you right-click on the block in IP integrator, choose **Edit IP Meta Data** to access the IP Customization Parameters and view the correct names.

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: Exporting 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

Option	Description
`-ip-repo <path>`	Add HDL IP repository search path
`-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.

SDSoC Hardware Functions for Platform IP

If the IP blocks that implement a library are contained within a hardware platform, the associated function mappings must also be included in the SDSoC environment hardware platform description file There is no way to associate the function maps to the IP blocks in Vivado® Design Suite 2015.2, so you must manually add them the hardware platform description file.

After creating the function maps and `<IP>.xml`, copy the `<xd:fcnMap>` and `<xd:component>` elements into the hardware platform description file `<platform>_hw.pfm` at the end, following the platform component and before the `</xd:repository>` closing tag.

In addition, you must provide a component declaration derived from the IP-XACT `component.xml` within the IP. You create this component declaration by running the following command within an SDSoC environment command shell.

```
xsltproc --stringparam P_XD_AUTOESL_COMP TRUE --stringparam \
     P_XD_COMP_TYPE accelerator -o <IP>.xml \
     <sds_root>/scripts/xsd/ipxact2xdcomp.xsl \
     <platform_root>/vivado/<platform>.ipdefs/repo/<IP_name>/<IP_name>/component.xml
```

The table below describes the command-line parameters:

Parameter	Description
<sds_root>	Root of the SDSoC install
<platform>	Platform name
<platform_root>	Platform directory
<IP>	IP name
<IP_name_version>	IP directory in the Vivado project

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 Using C-Callable IP Libraries 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. In keeping 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 Using C-Callable IP Libraries 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 that 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 over an AXI master interface in the IP
- `arraycopy_axis` - array transfers using AXI4-Stream interfacesThe 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 applicationsA simple test example that employs both IPs is available in the `samples/rtl_lib/arraycopy/use` directory. In the terminal shell, run `make` to create a Linux application that exercises both hardware functions.

See also Using C-Callable IP Libraries for using the library within the SDSoC IDE.

XILINX
ALL PROGRAMMABLE™

Chapter 4

Tutorial: Creating an SDSoC Platform

In this tutorial, you create simple example platforms in the SDSoC™ environment starting from hardware systems built using the Vivado® Design Suite.

Recall that a platform hardware description defines a connectivity interface for the design built in the Vivado tools, consisting of AXI and AXI stream, clock, reset, and interrupt ports to which the SDSoC environment can connect hardware functions and data mover channels. This interface encapsulates the platform hardware system as a single "component" seen by the SDSoC compiler that essentially corresponds to an IP integrator block diagram (BD) for the base platform hardware. You specify the interface through attributes on the BD and IP blocks within the BD.

The process of creating an SDSoC hardware platform specification consists of specifying the platform interface in IP integrator using Tcl commands in the Vivado tools Tcl Console, and using the Vivado tools to export the hardware description to SDSoC. Rather than explicit "external ports," each port on the interface is a reference to a specific port on an IP within the IP integrator block diagram.

Essentially any Vivado IP integrator system that targets the Zynq® SoC can be the basis of an SDSoC environment platform. The SDSoC environment automatically sets parameters on the Processing System 7 IP to enable AXI interfaces as needed, and adds additional DMA interrupts to the system, but otherwise does not modify any of the IP in the platform system.

It might be instructive to look at the platforms that are included in the SDSoC environment in the `<sdsoc_root>/platforms` directory. In addition, the following simple examples demonstrate different specific aspects of building a platform.

- `pf_axis` - Exporting direct I/O in an SDSoC platform
- `zc702_led` - Software control of IP within a platform
- `zc702_acp` - Sharing an AXI bus interface between platform and `sdscc`

Example: Exporting Direct I/O in an SDSoC Platform

This simple example is derived from the ZC702 platform included with the SDSoC™ environment, introducing an IP block that provides an AXI4-Stream master to proxy direct input to the Zynq® device Programmable Logic (PL), and another that provides an AXI4-Stream slave to proxy direct output from the PL.

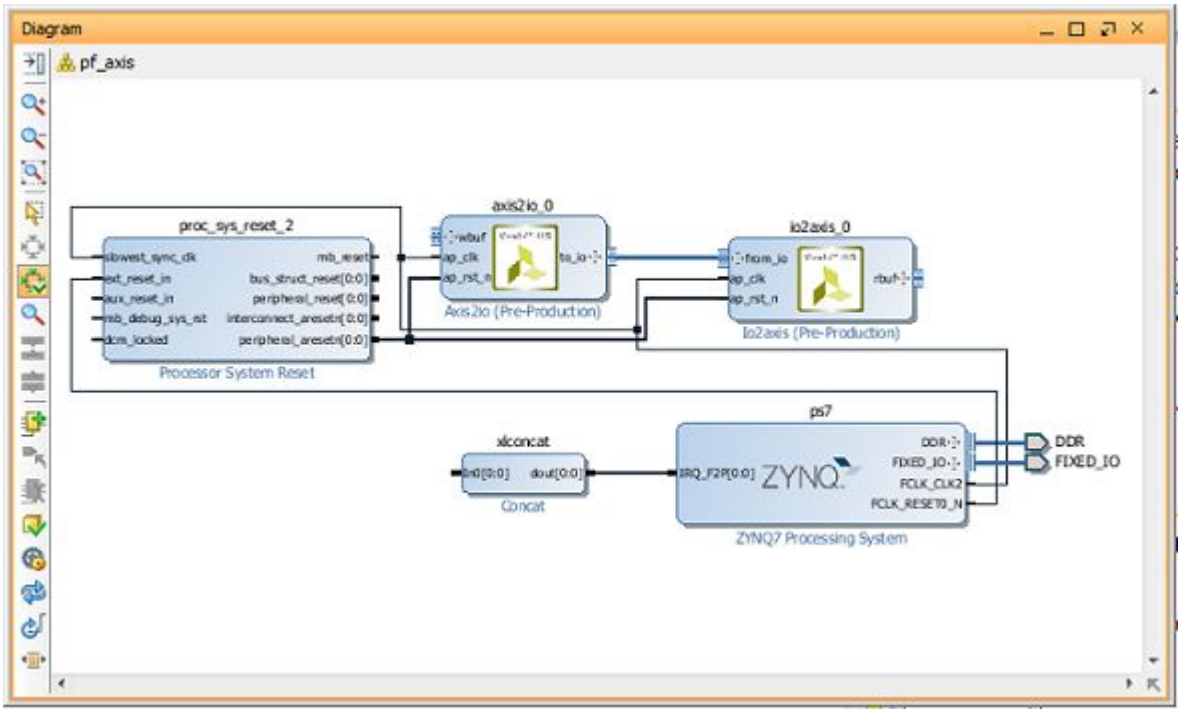
In a real platform, these blocks will be replaced by platform-specific IP that deliver input and output to the PL accessible by the SDSoC environment.

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

The Vivado® design is captured as a Tcl script `pf_axis.bd.tcl`. To build the hardware platform, open an SDSoC environment terminal shell, copy `<sds_root>/samples/platforms/pf_axis` into a new directory, and `cd` into this directory. The command, `make vivado` builds the hardware platform. After the script finishes, open the resulting Vivado project using the command, `$vivado pf_axis.xpr`

Open the block diagram, which should look similar to what is shown in the following figure.

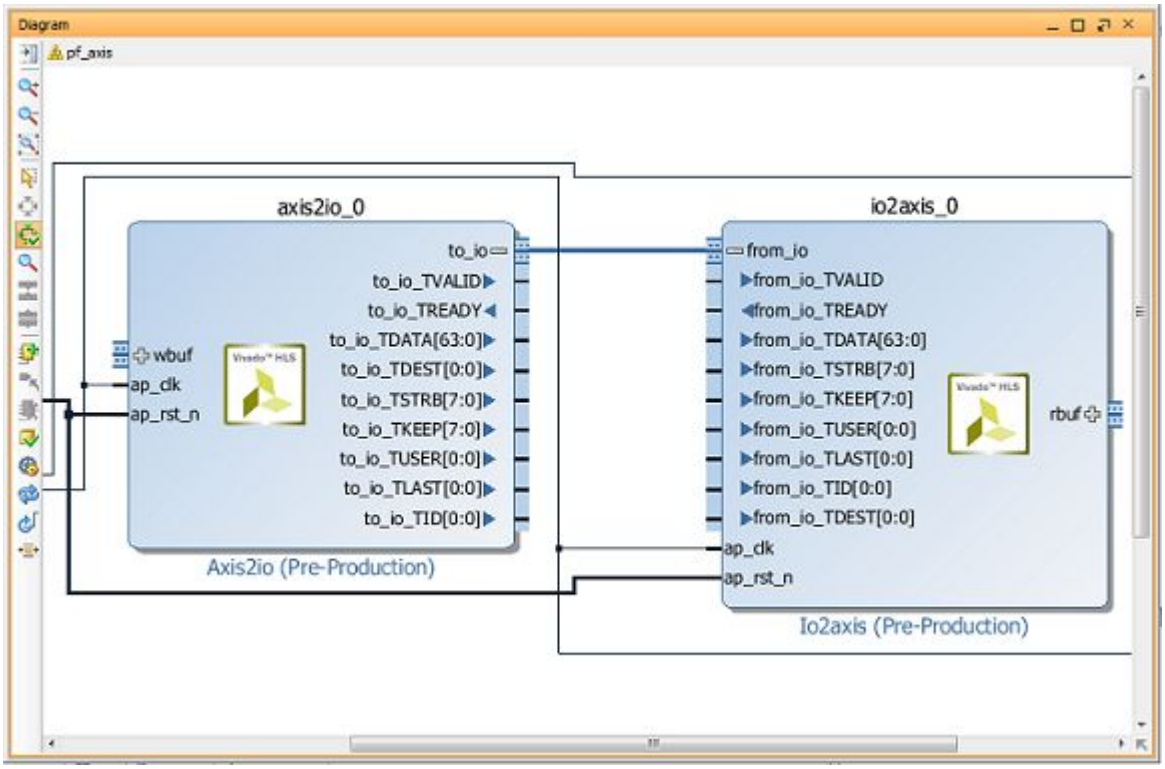
Figure 4–1: **Block Diagram**



The platform direct input (output) port is the unterminated `rbuf` (`wbuf`) port on the `io2axis_0` (`pf_write_0`) IP. The `to_io` output on the `axis2io_0` block is connected to the `from_io` input on the `io2axis_0` block so that the platform can be tested, but in a real platform, such ports are connected to other IPs and ultimately to PL pins on the Zynq device.

At the SDSoC platform interface, AXI4-Stream interfaces require `TLAST`, `TKEEP` sideband signals to comply with the IP underlying SDSoC environment data movers. You can confirm the existence of these sideband signals within IP integrator by expanding the ports as shown in the following figure.

Figure 4–2: **AXI4-Stream Bus Connections**



AXI4-Stream buses entirely within the platform do not require these sideband signals.

RECOMMENDED: *Currently, 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 AXI Data FIFO IP within the Vivado IP catalog for clock domain crossing. Make sure you configure the IP to generate the `TLAST` and `TKEEP` sideband signals.*

SDSoC Platform Tcl Commands in the Vivado Design Suite

The hardware port interface to an SDSoC™ environment platform consists of a set of unused AXI or AXI4-Stream bus interfaces on platform IP (for example, the Processing System 7 IP core), unused interrupts on the Processing System 7 IP block, clock ports, and synchronized resets,for example, provided by the Processor System Reset IP in the Vivado® IP catalog.

1. In the SDSoC environment terminal, open `pf_axis.bd.tcl` in a text editor. Most of this file was generated by the IP integrator `write_bd_tcl` command.
2. Scroll down to near the end of the Tcl file (starting on line 411) to inspect the API calls described in Vivado IP Integrator Tcl Commands to mark the SDSoC platform interfaces.
3. Use the following command to enable IP integrator to export additional metadata to create an SDSoC hardware platform description:

```
set_param project.enablePlatformHandoff true
```

4. Use the following command to declare the platform default clock:

```
set_property SDSOC_PFM.PFM_CLOCK TRUE [get_bd_pins /ps7/FCLK_CLK2]
```

5. Declare platform clocks and their associated ID with the following commands:

```
set_property SDSOC_PFM.CLOCK_ID 2 [get_bd_pins /ps7/FCLK_CLK2]
set_property SDSOC_PFM.MARK_SDSOC true [get_bd_pins /ps7/FCLK_CLK2]
```

6. You must synchronize platform resets to a specific clock using `proc_sys_reset` IP blocks.

   The following commands specify a platform reset bundle:

```
set_property SDSOC_PFM.MARK_SDSOC true [get_bd_pins /proc_sys_reset_2/interconnect_aresetn]
set_property SDSOC_PFM.MARK_SDSOC true [get_bd_pins /proc_sys_reset_2/peripheral_aresetn]
set_property SDSOC_PFM.MARK_SDSOC true [get_bd_pins /proc_sys_reset_2/peripheral_reset]
```

   The remaining `set_property` commands tag resets similarly.

7. Declare the `io2axis` master and `axis2io` slave AXI4-Stream bus interfaces that proxy the direct I/O with the following commands:

```
set_property SDSOC_PFM.MARK_SDSOC true [get_bd_intf_pins /io2axis_0/rbuf]
set_property SDSOC_PFM.MARK_SDSOC true [get_bd_intf_pins /axis2io_0/wbuf]
```

All other commands in this script are standard Vivado tools commands. The following commands export a hardware handoff file from the Vivado Design Suite to the SDSoC environment and invoke the HSI utility to generate the platform description:

```
write_hwdef -file "[file join ${pf}.sdk ${pf}_wrapper.hdf]"
load_features hsi
hsi::open_hw_design <platform>.sdk/<platform>_wrapper.hdf
hsi::generate_target {sdsoc} [hsi::current_hw_design] -dir <target_directory>
```

This generates the hardware platform description file `hsi/pf_axis.pfm`. When you run the `make pf_axis` target, this file is copied into the platform directory, renamed to `pf_axis_hw.pfm`. Open the platform description file in a text editor and look for the clocks, reset, and bus interfaces that were specified in `pf_axis.bd.tcl`.

IMPORTANT:

Not all required metadata are automatically generated from the Vivado tools using Tcl APIs, so you must add this metadata manually to the generated XML file `<platform>_hw.pfm`.

Known issues include: Platform AXI4-Stream bus interfaces containing `TLAST`, `TKEEP` sideband signals must have the `xd:hasTlast` attribute (with value "true") in the corresponding `xd:busInterface` element, but these are not automatically detected.

SDSoC Platform Software Libraries

The SDSoC system compilers (`sdscc/sds++`) map application program data flow between hardware functions into connections between the IPs included in the platform and accelerator functions implemented in programmable logic fabric.

Consequently, 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.

The 'all' make target in `pf_lib/Makefile` uses the SDSoC `sdslib` utility to create a static C-callable library for the platform as described in Creating a Library. It is worth noting that you can also use `sdslib` to wrap HDL IPs as accelerator functions that can be linked by `sdscc` within an application.

1. Define the C-callable interfaces for I/O IPs. In the SDSoC tool command shell, `cd` into the `pf_lib` directory.

   There are two platform IPs that require C-callable functions, `pf_read` and `pf_write`. The hardware functions are defined in `pf_read.cpp` and `pf_write.cpp`, as follows:

   ```
   void pf_read(u64 rbuf[N]) {}
   void pf_write(u64 wbuf[N]) {}
   ```

   The function bodies are empty; The `sdscc` compiler fills in stub function bodies with the appropriate code to move data. 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 mappings from the function interfaces to the respective IP ports. 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` and `pf_write` IPs are captured in `pf_read.fcnmap.xml` and `pf_write.fcnmap.xml`. Open `pf_read.fcnmap.xml`.

   ```
   <xd:fcnMap xd:fcnName="pf_read" xd:componentRef="pf_axis">
       <xd:arg
           xd:name="read_buf"
           xd:direction="out"
           xd:busInterfaceRef="io2axis_0_rbuf"
           xd:portInterfaceType="axis"
           xd:arraySize="128"
           xd:dataWidth="64"/>
       <xd:latencyEstimates xd:best-case="258" xd:worst-case="258" xd:average-case="258"/>
       <xd:resourceEstimates xd:LUT="62" xd:FF="47" xd:BRAM="0" xd:DSP="0"/>
   </xd:fcnMap>
   ```

   Each function argument requires name, direction, IP bus interface name, interface type, and data width. Array arguments must specify array size. Scalar arguments must specify a register offset. The SDSoC environment uses the latency estimates during high level scheduling.

IMPORTANT: *The `fcnMap` associates the platform function `pf_read` with the platform bus interface `pf_read_0_rbuf` on the platform component `pf_axis`, not the bus interface on the actual IP within the `pf_axis` platform that implements the function. In `pf_axis_hw.pfm` the `pf_axis` bus interface ("port") named `io2axis_0_rbuf` (`axis2io_0_wbuf`) contains the actual mapping to the IP in the `xd:instanceRef` attribute (`io2axis_0`).*

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 `pf_read.params.xml` and `pf_write.params.xml` are particularly simple. To see a more interesting example, open `<sdsroot>/samples/fir_lib/build/fir_compiler.{fcnmap,params}.xml` in the SDSoC install tree. This example maps multiple functions onto the Vivado FIR Compiler IP core.

4. Build the library.

   The `sdslib` commands are as follows.

```
sdslib -lib libpf_axis.a \
 pf_read pf_read.cpp \
 -vlnv xilinx.com:sds:pf_read:1.0 \
 -ip-map pf_read.fcnmap.xml \
 -ip-repo ./xilinx_com_sds_pf_read_1_0 \
 -ip-params pf_read.params.xml

sdslib -lib libpf_axis.a \
 pf_write pf_write.cpp \
 -vlnv xilinx.com:sds:pf_write:1.0 \
 -ip-map pf_write.fcnmap.xml \
 -ip-repo ./xilinx_com_sds_pf_write_1_0 \
 -ip-params pf_write.params.xml
```

You can call `sdslib` repeatedly for a library. Each call adds additional functions into the library.

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 `pf_axis` platform reuses all of the ZC702 boot files.

1. Open the platform software description, `pf_axis/pf_axis/pf_axis_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="arm-xilinx-linux-gnueabi/include"
    xd:libDir="arm-xilinx-linux-gnueabi/lib"
    xd:libName=”pf_axis”
/>
```

   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"
/>
```

Building the pf_axis Platform

1. To build the SDSoC™ environment platform from the SDSoC terminal:

```
$ make pf_axis
```

The `pf_axis` build target pulls together all the components of the SDSoC environment platform. It creates a `pf_axis` root directory for the platform, expands an archive of the Vivado project into a `vivado` subdirectory. It then calls `make -C pf_lib` to create the software library, copying the library and header into an `arm-xilinx-linux-gnueabi` subdirectory and copying the `pf_axis_hw.pfm` and `pf_axis_sw.pfm` metadata files into the platform directory.

Test the pf_axis Platform

This example provides a very simple C++ application in `pf_axis/pf_test/pf_test.cpp`. Key points in this function are the ways in which buffers are allocated using `sds_alloc`,

```
u64 *wbuf = (u64 *) sds_alloc(N * sizeof(u64));
u64 *rbuf = (u64 *) sds_alloc(N * sizeof(u64));
```

and the way that the platform functions are invoked to either read from platform inputs or write to platform outputs.

```
pf_write(wbuf); // write to platform output
pf_read(rbuf); // read from platform input
```

1. `cd` into the `pf_axis/pf_test` directory and execute `make all`
2. After the SDSoC environment creates an sd_card image, copy onto an SD card, boot, and run `pf_test.elf`.

```
sh-4.3# ./pf_axis.elf
registering devices
generating device nodes...
Test PASSED!
sh-4.3#
```

Example: Software Control of Platform IP

This example is a single clock, ZC702-based, platform with a general purpose I/O (AXI GPIO) IP block (`axi_gpio`) implemented in programmable logic and connected to the LEDs on the board. The AXI GPIO IP has an associated Linux kernel driver, but this example demonstrates how to provide software control of platform IP using the Linux UIO (userspace I/O) framework to communicate with the GPIO directly from a test application. In this example, you learn how to create a platform software library outside the SDSoC™ environment then make it available to applications within the SDSoC environment.

RECOMMENDED: *As you read through this tutorial, you should work through the example provided in `<sds_root>/samples/platforms/zc702_led/`. See the `readme.txt` file for instructions to build and test the platform.*

Build System and SDSoC Platform Hardware Description

1. Make a local working copy of `samples/platforms/zc702_led`, and from an SDSoC™ environment terminal shell, `cd` into this directory.

   There is a makefile in this directory to build the platform as well as a Tcl script that defines the platform.

2. From the terminal shell, run the following command: `$ make vivado`

   This command invokes the Vivado tools on the `zc702_led.bd.tcl` script to build the base system.

This script invokes Tcl APIs to specify the default platform clock, the corresponding clock ID, and the reset interfaces.

```
set_property SDSOC_PFM.PFM_CLOCK TRUE [get_bd_pins /ps7/FCLK_CLK2]
set_property SDSOC_PFM.PFM_CLOCK_ID 2 [get_bd_pins /ps7/FCLK_CLK2]
set_property SDSOC_PFM.UIO true [get_bd_cells /axi_gpio_0]
set_property SDSOC_PFM.MARK_SDSOC true [get_bd_pins /ps7/FCLK_CLK2]
set_property SDSOC_PFM.MARK_SDSOC true [get_bd_pins /proc_sys_reset_2/interconnect_aresetn]
set_property SDSOC_PFM.MARK_SDSOC true [get_bd_pins /proc_sys_reset_2/peripheral_aresetn]
set_property SDSOC_PFM.MARK_SDSOC true [get_bd_pins /proc_sys_reset_2/peripheral_reset]
```

The SDSoC environment employs the Linux UIO framework for hardware functions and consequently must be informed of any UIO platform devices within the platform. For the `zc702_led` platform, this is accomplished by the following API call in the script:

```
set_property SDSOC_PFM.UIO true [get_bd_cells /axi_gpio_0]
```

This part of the script declares the `axi_gpio_0` instance to be a UIO device.

The script then builds the design and uses the HSI utility to generate the hardware description.

```
load_features hsi
hsi::open_hw_design "[file join vivado ${pf}.sdk ${pf}_wrapper.hdf]"
hsi::generate_target {sdsoc} [hsi::current_hw_design] -dir hsi
```

This generates the platform hardware description `hsi/zc702_led.pfm`.

Build the SDSoC Platform

1. To build the SDSoC™ environment platform, from the terminal shell run the following command: `make zc702_led`

This build target unzips an archived version of the Vivado® tools project built in the previous step, and then builds a platform software library to access the GPIO block driving the LEDs on the ZC702 board using the `make -C lib` command.

The boot environment for the `zc702_led` 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.

Linux UIO Framework for Userspace Control of Platform IP

The default `lib` build target creates a software library containing the UIO driver for the `axi_gpio` block consisting of the files `uio_axi_gpio.[ch]`. This driver provides a simple API for controlling the GPIO IP. See `test/pf_axis.cpp` for example usage of the API.

The devicetree 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 devicetree source) using the dtc compiler

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

IMPORTANT: *the dtc compiler must be built and run on a Linux host machine or Virtual Machine (VM). You can find the source to build the dtc at `https://github.com/Xilinx/linux-xlnx/tree/master/scripts/dtc`.*

To register the `axi_gpio_0` platform peripheral with Linux, add the following devicetree blob to the devicetree as required by the UIO framework:

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

This blob was manually created and inserted into the device tree as the lexically first occurring `generic-uio` device within the `amba` record in the devicetree.

The name 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 call the initialization function before accessing the peripheral and must release the resource when it is finished. The test application in `zc702_led/test` provides a sample usage.

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 SDSoC Platform

1. To test the platform, from the terminal shell run the following command, `$ make -C test`.

The test application contains a simple arraycopy hardware function using the SDSoC™ development environment `axi_lite` data mover, invoked within a loop. The LEDs on the ZC702 are lit to match the binary representation of the loop index.

Although quite simple, this design demonstrates how you can employ the Linux UIO framework to control platform peripherals and provide a software library as part of your platform for use in SDSoC environment applications.

Example: Sharing a Processing System 7 IP AXI Port

This example demonstrates how to share a Processing System 7 (`processing_system7`) IP AXI port between a platform and an IP generated in the SDSoC™ environment. The AXI port must be connected to an AXI Interconnect IP block within the platform, and the platform exports a corresponding master or slave port on the interconnect. During system generation, the SDSoC environment maps data channels to the platform by cascading an interconnect connected to this exported port.

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

IMPORTANT: *Whenever a platform exports an AXI or AXI4-Stream bus interface that is not part of the Processing System 7 IP, every application targeting the platform must use every such bus interface. Otherwise, the system errors out in the Vivado Design Suite due to unterminated interfaces. For an AXI master or slave on an AXI Interconnect IP, you can set the `NUM_SI` and `NUM_MI` IP parameters to automatically enable the correct number of slaves or masters. See Step 3 below for an example.*

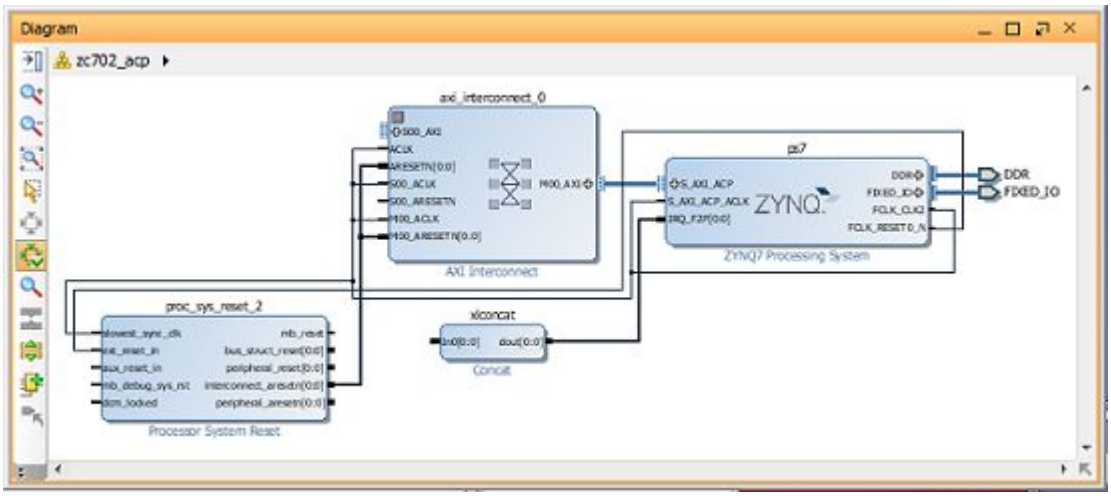
Build Hardware and SDSoC Platform Hardware Description

1. Make a local working copy of `samples/platforms/zc702_acp`, and from an SDSoC™ environment terminal shell, `cd` into this directory.

   There is a makefile in this directory to build the platform as well as a Tcl script that defines the platform.

2. From the terminal shell, run the following command: `$ make vivado`

This command invokes the Vivado tools on the `zc702_acp.bd.tcl` script to build the base system. The IP integrator block diagram for the generated system is shown in the following figure.



This design is particularly simple; the platform does not actually even use the ACP other than to export a port on an AXI Interconnect IP (`axi_interconnect`). A real platform will use some of the interconnect slave ports, which must be the least significant `Snm_AXI` ports (no gaps) and then export the least significant indexed unused port.

This script invokes Tcl APIs (see Vivado IP Integrator Tcl Commands) to specify the default platform clock, the corresponding clock ID, and the reset interfaces.

```
set_property SDSOC_PFM.PFM_CLOCK TRUE [get_bd_pins /ps7/FCLK_CLK2]
set_property SDSOC_PFM.CLOCK_ID 2 [get_bd_pins /ps7/FCLK_CLK2]
set_property SDSOC_PFM.MARK_SDSOC true [get_bd_pins /ps7/FCLK_CLK2]
set_property SDSOC_PFM.MARK_SDSOC true [get_bd_pins /proc_sys_reset_2/interconnect_aresetn]
set_property SDSOC_PFM.MARK_SDSOC true [get_bd_pins /proc_sys_reset_2/peripheral_aresetn]
set_property SDSOC_PFM.MARK_SDSOC true [get_bd_pins /proc_sys_reset_2/peripheral_reset]
```

Every Processing System 7 IP AXI interface is connected to an AXI Interconnect IP. To share such an interface with the SDSoC environment, the platform exports the least significant unused slave interconnect on the IP that masters the Processing System 7 IP `S_AXI_ACP`. Because, in this example, there are no AXI masters within the platform, the platform exports `S00_AXI` with the following commands.

```
set_property SDSOC_PFM.MARK_SDSOC true [get_bd_intf_pins /axi_interconnect_0/S00_AXI]
set_property SDSOC_PFM.MARK_SDSOC true [get_bd_pins /axi_interconnect_0/S00_ARESETN]
set_property SDSOC_PFM.MARK_SDSOC true [get_bd_pins /axi_interconnect_0/S00_ACLK]
```

3. You must mark the clock and reset pins corresponding to an AXI interface. The AXI Interconnect IP requires customization by the SDSoC environment whenever it is used in the application. To accomplish this customization, set the parameters in the generated system, using metadata captured in the platform as shown in the following code:

```
set_property SDSOC_PFM.PFM_PARAMS {NUM_SI} [get_bd_cells /axi_interconnect_0]
set_property SDSOC_PFM.NUM_SI {count($designComponent/xd:connection/xd:busInterface
  [@xd:instanceRef=$instance and @xd:name='axi_interconnect_0_S00_AXI'])+0}
  [get_bd_cells /axi_interconnect_0] [get_bd_cells /axi_interconnect_0]
# NOTE: set_property line breaks added for readability (remove in practice)
```

   The script then builds the design and uses the HSI utility to generate the hardware description.

```
load_features hsi
hsi::open_hw_design "[file join vivado ${pf}.sdk ${pf}_wrapper.hdf]"
hsi::generate_target {sdsoc} [hsi::current_hw_design] -dir hsi
```

   This process generates the platform hardware description `hsi/zc702_acp.pfm`.

In the current Vivado tools release, some of the necessary platform metadata for this example is missing or incorrect, and must be added manually. The `S00_AXI` element has an incorrect `xd:clockRef` attribute value and is missing metadata. It should be as shown in the following:

```
<xd:busInterface
    xd:busInterfaceRef="S00_AXI"
    xd:busTypeRef="aximm"
    xd:clockRef="axi_interconnect_0_S00_ACLK"
    xd:resetRef="axi_interconnect_0_S00_ARESETN"
    xd:instanceRef="axi_interconnect_0"
    xd:mode="slave"
    xd:name="axi_interconnect_0_S00_AXI"
    xd:idBits="3"
    xd:coherent="true"
    xd:memport="S_AXI_ACP"
/>
```

The `xd:idBits` value should represent the number of available ID bits on the ACP port. The `xd:coherent` attribute is required for any interconnect port that provides access to the ACP. The `xd:memport` attribute is required for any AXI slave port that provides access to DDR. Make these changes manually in a text editor.

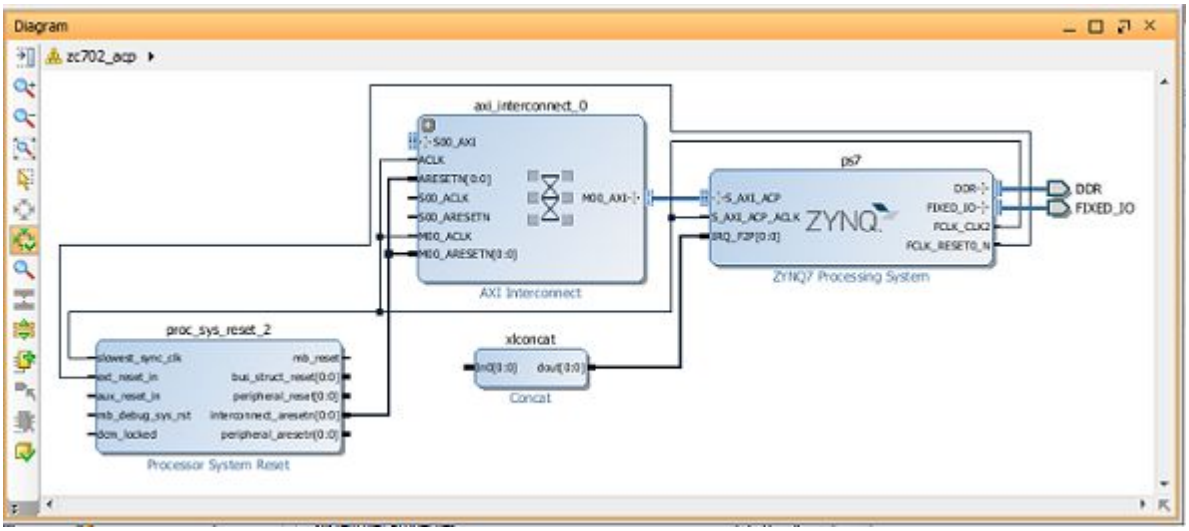
Build the SDSoC Platform

1. To build the SDSoC™ platform, from the terminal shell run the following command: `make zc702_acp`.

   This build target unzips an archived version of the Vivado® tools project built by the previous step. In general, a platform clock can be driven by a clock source other than the Processing System 7 IP block (for example, a clocking wizard), and can be completely independent of the IPs generated by the SDSoC environment. For this reason, the clock port for each AXI interconnect must be left unterminated in the platform Vivado tools project. The SDSoC environment automatically connects this to the correct clock when generating the final system.

2. Open the Vivado tools project `zc702/vivado/zc702_acp.xpr` and disconnect `/axi_interconnect_0/S00_ACLK` from the clock net. The SDSoC environment connects this clock port to the same clock net as the AXI mastering this bus connection. You need to reconnect all of the other IP clock pins after deleting the connection to the `axi_interconnect`.

   The `zc702_acp` platform system is shown in the following figure:



IMPORTANT: *You must manually delete the connection to the `axi_interconnect` slave clock port so that the SDSoC environment can connect this port to whatever platform clock is driving this AXI slave. Otherwise, users of your platform will be unable to generate correct logic whenever the SDSoC environment clock and the platform clock are different.*

The boot environment for the `zc702_led` is identical to the `zc702`.

Test the SDSoC Platform

1. To test the platform, from the terminal shell run: `make -C pf_test`
2. The test application contains a simple arraycopy hardware function using the SDSoC environment `zero_copy` (accelerator mastered AXI bus). Load the contents of `pf_test/sd_card` into an SD card and boot.

```
$ /mnt/zc702_acp_test.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: Getting Started* (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