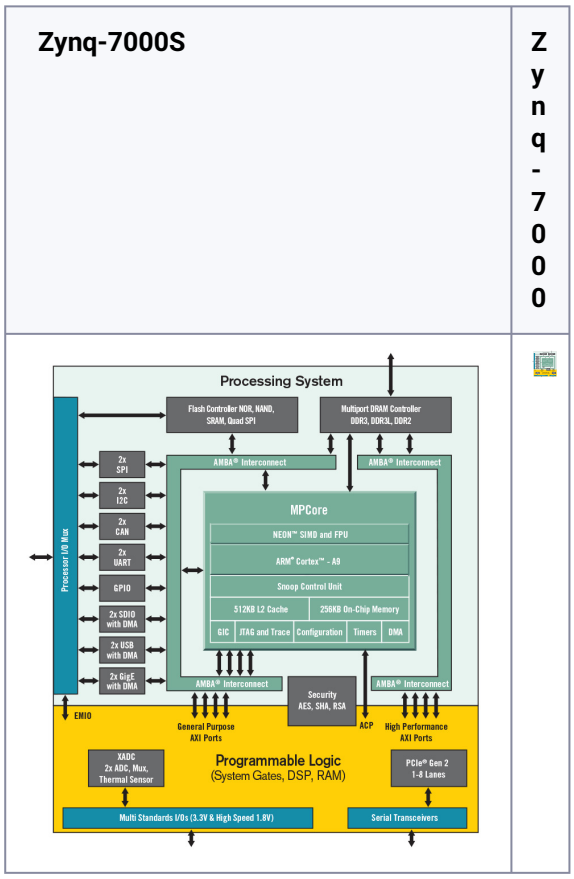
Zynq-7000

Xilinx Wiki

Exported on 01/04/2024

Table of Contents



This page provides a list of resources to help you get started using the Xilinx Zynq-7000 SoC, including pre-built images for Xilinx development boards, tutorials, and example designs. More detailed information can be found by following the links provided on this page.

Whether you're an expert or novice user, the easiest way to get started with a Xilinx development board is to start with a pre-built Linux image for your board. If you're new the Xilinx embedded design flow, the Embedded Design Tutorial is the recommended way to learn the tools and design flow. To build a custom Linux image, it's recommended that you start with a Petalinux BSP for one of the Xilinx boards, and then customize the configuration to suit your needs.

Table of Contents

1 Pre-Built Release Images

The pre-built images referenced here are for the Xilinx development boards. These can loaded on to SD Cards on the Xilinx development boards and you can boot Linux. The Pre-Built Releases Images page includes images for Zynq UltraScale+ MPSoC, Zynq UltraScale+ RFSoC and Zynq-7000.

2 Evaluation Boards and BSP

PetaLinux Board Support Packages (BSP) and Reference Examples include pre-built boot loaders, system images and bitstreams. Built-in tools allow a single command to deploy and boot these elements to either physical hardware, or to the included full QEMU system emulator. With PetaLinux, developers can have their Xilinx-based hardware booted and running within about 5 minutes after installation; ready for application, library and driver development.

2.1 Xilinx Boards

Xilinx provides two development board for the Zynq-7000 devices. For more information, the links below take you to board-specific pages at Xilinx.com

- ZC702 - Z7020 device
- ZC706 - Z7045 device

Each board also comes with a PetaLinux BSP that includes an image, documentation to recreate that image and a design that can be used as a starting point for the hardware user. There is one BSP for each board above. They are called PetaLinux BSPs since the Xilinx tool PetaLInux is used to create these images. The links below take you to the PetaLinux Download page at Xilinx.com. Please note that you will need a Xilinx.com login to download these files.

- ZC702 PetaLinux BSP
- ZC706 PetaLinux BSP

2.2 Partner Boards

Xilinx has many partners that design boards using Zynq-7000 SoCs and here are a few.

- Avnet - Zedboard.com
    - MicroZed
    - PicoZed
    - MiniZed (Zynq-7000S)
    - Zed Board
    - ZED Board PetaLinux BSP from Xilinx
- Digilent
    - ZedBoard
    - Arty Z7
    - PYNQ-Z1
    - Cora Z7 (Zynq-7000S or Zynq-7000)

3 Example designs

Xilinx provides a variety of example designs on their development boards for the users. These range OS, power management and graphic examples. An example design is a snapshot in time. What this means is that the design is done on a specific Xilinx tool release and not necessarily updated to other tool releases or the current release. The user can take these and update them on their own.

- Zynq-7000 Example Designs

4 Targeted Reference Designs (TRD)

Xilinx also provides a smaller set of Targeted Reference Designs or TRDs. There are not as many TRDs as Example Designs. However these TRDs are updated on each major tool release for a set amount of time. The TRDs are fully supported by Xilinx.

- Zynq -7000 TRDs

5 Embedded Design Tutorial (EDT)

The Embedded Design Tutorial provides an introduction to using the Xilinx® Vivado® Design Suite flow for using the Zynq-7000 device. The examples are targeted for the Xilinx ZC702 evaluation boards. The latest versions of the EDT use the Vitis™ Unified Software Platform.

- UG1165 - Zynq-7000 MPSoC Embedded Design Tutorial

6 Overview of the Embedded Software Stack on a Zynq-7000

The following is an overview of the embedded software stack for a Zynq-7000 SoC.

The processor system boot is a two-stage process. The first stage is an internal BootROM which stores the stage-0 boot code. The BootROM executes on CPU 0 and CPU 1 executes the wait-for-event (WFE) instruction. The BootROM also configures the necessary peripherals to start fetching the First Stage Bootloader (FSBL) boot code from one of the boot devices. The programmable logic (PL) is not configured by the BootROM. The second state is the FSBL boot code. This is typically stored in one of the flash memories, the SD card, or can be downloaded through JTAG. The BootROM code copies the FSBL boot code from the chosen non-volatile memory to on-chip memory (OCM). The size of the FSBL loaded into OCM is limited to 192 kilobyte. The full 256 kilobyte is available after the FSBL begins executing. There is an optional second stage boot loader that is optional and user-designed. A common second stage boot loader is U-boot.

6.1 FSBL

The First Stage Bootloader (FSBL) for Zynq-7000 configures the FPGA with the hardware bitstream (if it exists) and loads the Operating System (OS) Image, Standalone (SA), or 2nd Stage Boot Loader image from the non-volatile memory (NAND/SD/eMMC/QSPI) to Memory (DDR/TCM/OCM). It supports multiple partitions and each partition can be a code image, bitstream, or generic data. Each of these partitions, if required, can be authenticated and/or decrypted.

For more information, go to the Zynq-7000 FSBL page.

6.2 U-Boot

U-Boot, short for Universal Boot Loader, is an open source, primary boot loader used in embedded devices to boot the device's operating system kernel that is frequently used in the Linux community. Xilinx uses U-Boot as a second stage boot loader in the Zynq-7000 devices. For more information about U-Boot visit their page at https://www.denx.de/wiki/U-Boot.

For more information about U-Boot on Zynq-7000 devices go to the U-Boot page on this wiki.

6.3 Linux

Since Linux is the primary OS that people start with on the Zynq-7000 devices there is more information on it at the Linux page. This includes the two different build tools used to create customer distributions. These two tools include Xilinx's PetaLinux and Yocto, an open source project that is part of the Linux Foundation. The Linux page also describes how to build your own Linux from the source and links to information about the Linux drivers that Xilinx provides.

7 Power Management

For Zynq-7000 Power Management there is a are several wiki pages dedicated to this but a good starting point is the Zynq-7000 Power Management page.

8 Security

Zynq-7000 provides hardware accelerators to implement integrity, confidentiality, and authentication in system. For a list of resources on Zynq-7000 Security visit the following page: Zynq-7000 AP SoC Security. There is also a section in the Zynq-7000 All Programmable SoC Software Developers Guide - (UG821) about security and secure boot.

9 Documentation/Resources

The following link is a list of all the documentation for Zynq-7000 from Xilinx.com. This information is hosted on the web but is also available with an installation of the Xilinx tool DocNav.

- Zynq 7000 SoC Design Hub

The Xilinx Community Forums are places to get some answers on questions or search to see what issues others have had with Xilinx devices.

- https://forums.xilinx.com/

The Xilinx Community Portal showcases Xilinx in the Open Source Space and highlights projects that have been done with Xilinx products.

- https://www.xilinx.com/community.html

The Zynq-7000 SoC Solution Center is available to address all questions related to the Zynq-7000 SoC. Whether you are starting a new design with Zynq-7000 SoC or troubleshooting a problem, use the Zynq-7000 SoC solution center to guide you to the right information.

- Zynq-7000 SoC Solution Center Answer Record

10 Tools

The Xilinx tools provide all required tool chains to compile and link applications for Xilinx supported platforms, create and configure hardware designs and create bitstreams.

Installing the Xilinx Tools

11 Wiki Articles

- Zynq-7000 Example Designs
- Zynq-7000 Targeted Reference Deisgns
- Zynq-7000 FSBL
- Zynq-7000 Power Management
- Zynq-7000 BIST Guide
- Zynq Qt and Qwt Base Libraries-Build Instructions
- Zynq-7000 AP SoC SATA part 1 – Ready to Run Design Example Setup
- Zynq-7000 Analog Data Acquistion using AXI_XADC
- XAPP1231 - Partial Reconfiguration of a Hardware Accelerator with Vivado Design Suite
- Zynq 7000 Partial Reconfiguration Reference Design
- Data Movers
- Programming QSPI from U-boot ZC702
- Zynq Ethernet Performance
- Zynq 7000 Tips and Tricks

12 Zynq-7000 Example Designs

This page has the list and points to Zynq-7000 example designs. An example design is a design that is in a point in time. Meaning done on a Xilinx tool release and not necessarially updated. If the user wants this design example they can use it on the tool release it was created on or take on porting to the desired tool release on their own.

1. Zynq-7000 AP SoC - 32 Bit DDR Access with ECC Tech Tip
2. Zynq-7000 AP SoC - Base TRD execution from 32 Bit ECC Proxy System Tech Tip
3. Zynq-7000 AP SoC - Implementing a Host PC GUI for Communication with Zynq Tech Tip
4. Zynq-7000 AP SoC - Installing the Ubuntu Desktop on PetaLinux and Demo Tech Tip
5. Zynq-7000 AP SoC - Performance - Ethernet Packet Inspection - Bare Metal - Redirecting Packets to PL Tech Tip
6. Zynq-7000 AP SoC - Performance - Ethernet Packet Inspection - Bare Metal - Redirecting Headers to PL and Cache Tech Tip
7. Zynq-7000 AP SoC - Performance - Ethernet Packet Inspection - Linux - Redirecting Packets to PL and Cache Tech Tip
8. Zynq-7000 AP SoC - Precision Timing with IEEE1588 v2 Protocol Tech Tip
9. Zynq-7000 AP SoC - Protocol Communication Between a Host PC and ZC702 Board Tech Tip
10. Zynq-7000 AP SoC - Read and Write to the Zynq OCM from The PL
11. Zynq-7000 AP SoC - RealTime - InterruptLatency Reference Design and Demo Tech Tip
12. Zynq-7000 AP SoC - Using BRAM for Additional On-Chip Memory Tech Tip
13. Zynq-7000 AP SoC - Zynq BFM Simulation of Packet Processing Unit in PL Tech Tip
14. Zynq-7000 AP SoC Benchmark - LMBench Tech Tip
15. Zynq-7000 AP SoC Benchmark - SPEC CPU 2000 Tech Tip
16. Zynq-7000 AP SoC Benchmarking & debugging - Ethernet TechTip
17. Zynq-7000 AP SoC Boot - Booting and Running Without External Memory Tech Tip
18. Zynq-7000 AP SoC Boot - Locking and Executing out of L2 Cache Tech Tip
19. Zynq-7000 AP SoC Boot - Multiboot Tech Tip
20. Zynq-7000 AP SoC Boot - Programmable Logic Configuration via Ethernet Tech Tip
21. Zynq-7000 AP SoC Boot - Rebooting to a Different Boot Image and Bitstream from Linux Tech Tip
22. Zynq-7000 AP SoC Low Power Techniques part 1 - Installing and Running the Power Demo Tech Tip
23. Zynq-7000 AP SoC Low Power Techniques part 2 - Measuring ZC702 Power using TI Fusion Power Designer Tech Tip

24. Zynq-7000 AP SoC Low Power Techniques part 3 - Measuring ZC702 Power with a Standalone Application Tech Tip
25. Zynq-7000 AP SoC Low Power Techniques part 4 - Measuring ZC702 Power with a Linux Application Tech Tip
26. Zynq-7000 AP SoC Low Power Techniques part 5 - Linux Application Control of Processing System - Frequency Scaling & More Tech Tip
27. Zynq-7000 AP SoC Low Power Techniques part 6 - Linux Application Control of Programmable Logic Frequency & More Tech Tip
28. Zynq-7000 AP SoC Low Power Techniques part 7 - Ubuntu Application Control of Processing System - Frequency Scaling & More Tech Tip
29. Zynq-7000 AP SoC Performance – Gigabit Ethernet achieving the best performance
30. Zynq-7000 AP SoC SATA part 1 – Ready to Run Design Example Setup
31. Zynq-7000 AP SoC SATA part 2 – Ready to Run Design Example Benchmarking
32. Zynq-7000 AP SoC SATA part 3 – Ready to Run NAS Design Example Setup
33. Zynq-7000 AP SoC SATA part 4 – Ready to Run NAS Design Example Benchmarking
34. Zynq-7000 AP SoC SATA part 5 – Building the Design Example
35. Zynq-7000 AP SOC Secondary Boot Over PCIe Techtip
36. Zynq-7000 AP SoC Spectrum Analyzer part 1 - Accelerating Software & More - Installing and Running the Spectrum Analyzer Demo Tech Tip
37. Zynq-7000 AP SoC Spectrum Analyzer part 1 - Accelerating Software & More - Installing and Running the Spectrum Analyzer Demo Tech Tip 2014.3
38. Zynq-7000 AP SoC Spectrum Analyzer part 2 - Accelerating Software - Building ARM NEON Library Tech Tip
39. Zynq-7000 AP SoC Spectrum Analyzer part 2 - Accelerating Software - Building ARM NEON Library Tech Tip 2014.3
40. Zynq-7000 AP SoC Spectrum Analyzer part 3 - Accelerating Sfotware - Running ARM Library Tests Tech Tip
41. Zynq-7000 AP SoC Spectrum Analyzer part 3 - Accelerating Software - Running ARM Library Tests Tech Tip 2014.3
42. Zynq-7000 AP SoC Spectrum Analyzer part 4 - Accelerating Software - Building and Running an FFT Tech Tip
43. Zynq-7000 AP SoC Spectrum Analyzer part 4 - Accelerating Software - Building and Running an FFT Tech Tip 2014.3
44. Zynq-7000 AP SoC Spectrum Analyzer part 5 - Accelerating Software - Accelerating an FFT with ACP Coprocessor Tech Tip
45. Zynq-7000 AP SoC Spectrum Analyzer part 5 - Accelerating Software - Accelerating an FFT with ACP Coprocessor Tech Tip 2014.3

46. Zynq-7000 AP SoC Spectrum Analyzer part 6 - AMS - XADC Signal Acquisition and DMA to L2 Cache & Compete Design Tech Tip
47. Zynq-7000 AP SoC Spectrum Analyzer part 6 - AMS - XADC Signal Acquisition and DMA to L2 Cache & Complete Design Tech Tip 2014.3
48. Zynq-7000 AP SoC Spectrum Analyzer part 7 - Building and Running a QT based GUI Tech Tip 2014.3
49. Zynq-7000 AP SoC USB CDC Device Class Design Example Techtip
50. Zynq-7000 AP SoC USB Mass Storage Device Class Design Example Techtip
51. ZC702 Example Application in Linux
52. Zynq SoC - Programmable Logic Configuration via Ethernet
53. OpenWrt running on ZC702
54. HDMI FrameBuffer Example Design 2017.3
55. HDMI FrameBuffer Example Design 2018.1
56. HDMI FrameBuffer Example Design 2018.3

13 Zynq-7000 Targeted Reference Deisgns

This page has the list and points to Zynq-7000 Targeted Reference Designs or TRDs. These designed are updated on each major tool release for a set amount of time. The TRDs are fully supported by Xilinx.

Zynq UltraScale MPSoC Software Acceleration TRD 2016.2

Data Movers

K7 Embedded TRD 2013.2

Zynq Base TRD 14.1

Zynq Base TRD 14.2

Zynq Base TRD 14.3

Zynq Base TRD 14.4

Zynq Base TRD 14.5

Zynq Base TRD 2013.2

Zynq Base TRD 2013.3

Zynq Base TRD 2013.4

Zynq Base TRD 2014.2

Zynq Base TRD 2014.4

Zynq Base TRD 2015.2

Zynq Base TRD 2015.4

Zynq PCIe TRD 14.3

Zynq PCIe TRD 14.4

Zynq PCIe TRD 14.5

Zynq PCIe TRD 14.6

14 Zynq-7000 FSBL

This page provides details on building and customizing the FSBL for Zynq-7000, and important notes on the FSBL. All the information is presented in the format of FAQs.

14.1 Table of Contents

14.2 What is FSBL?

The First Stage Bootloader (FSBL) for ZYNQ-7000 configures the FPGA with hardware bitstream(if it exists) and loads second stage bootloader or bare-metal application code from the non-volatile memory(NAND/SD/QSPI) to memory (DDR/OCM) and takes A9 out of reset. It supports multiple partition can be a code image or bitstream. Each of these partitions, if required, will be authenticated and/or encrypted. FSBL is loaded into OCM and handed off by BootROM after authenticating and/or decrypting (as required) FSBL.

14.3 How to create FSBL from Vitis?

- Launch VITIS with the below command: vitis
- Provide path where VITIS workspace and project need to be created. With this VITIS workspace will be created
- (Optional step) To work with local repos, Select "Xilinx" (ALT - x) -> Repositories. Against Local Repositories, click on "New..." and provide path of the local repo
- Select File-->New-->Application Project to open "New Project" window, provide name for FSBL project
- In the “Platform” section, click on “Create a new platform from hardware (XSA)” and select pre-defined hardware platform for Zynq(e.g. zc702).
    - Alternatively, to create a new/custom platform from a .xsa file, click on “+”, browse and select the XSA file and a new hardware platform is created.
- In the "Domain" window, select the processor ps7_cortexa9_0/ps7_cortexa9_1, OS as standalone and Language as C
- Click Next and select "Zynq FSBL"
- Click "Finish" to generate the A9 FSBL. This populates the FSBL code and also builds it (along with BSP)
- Debug prints in FSBL are now disabled by default (except for FSBL banner). To enable debug prints, define symbol: FSBL_DEBUG_INFO
    - In VITIS this can be done by: right click on FSBL application project -> select “C/C++ Build Settings” -> “Tool Settings” tab ->Symbols (under ARM v7-A gcc compiler)
    - Click on Add (+) icon and Enter Value: FSBL_DEBUG_INFO, click on "OK" to close the "Enter Value" screen
- In case any of the source files (FSBL or BSP) need to be modified, browse the file, make the change and save the file, build the project.elf file will be present in the Debug/Release folder of FSBL project.

14.4 What are various levels of compilation flags in FSBL?

FSBL supports six levels of compilation flags:

Flag	Description
FSBL_DEBUG	Set this flag to enable the logs and message prints.
FSBL_DEBUG_INFO	Set this flag to obtain more detailed logs like register and partition header dumps.
FSBL_DEBUG_RSA	Set this flag to print more detailed intermediate values used in RSA functions.

Flag	Description
NON_PS_INSTANTIATE D_BITSTREAM	Set this flag when the bitstream does not have a PS component. Then the FSBL does not enable level shifters
RSA_SUPPORT	Set this flag to enable authentication feature in FSBL.
MMC_SUPPORT	Set this flag to enable MMC support in FSBL. When this flag is set, FSBL reads all the partitions from the eMMC device, instead of the primary boot device (which is set by the boot mode pins).

14.5 On what all processor cores can FSBL run on?

FSBL can only be run from A9_0 (AArch32)

14.6 What part of OCM is used by FSBL?

192 KB of OCM memory is accessible starting at address 0x0000_0000 for FSBL code memory. while the upper memory which is 64 KB of the OCM is accessible starting at address 0xFFFF_0000 for FSBL program memory.

14.7 Is there any xfsbl_translation_table.S different from translation_table.S of BSP in ZYNQ?

There is no xfsbl_translation_table.S, FSBL uses the default available translation_table.S of BSP.

14.8 What ECC initialization is done by FSBL?

DDR ECC Initialization: Initially it was added in FSBL source code, later moved to ps7_init and removed from FSBL.

14.9 Are there any other ways by which FSBL footprint can be further reduced?

Debug prints: By default only FSBL banner is printed. If more debug prints are enabled, time takes to print messages will add additional delay to boot time and adds extra code which result in increasing footprint.
Drivers Asserts: Asserts are used within all Xilinx drivers and can be turned off on a system-wide basis by

defining, at compile time, the NDEBUG identifier
(adding –DNDEBUG against extra_compiler_flags of drivers). This will help further reduce FSBL footprint.

14.10 What level of optimization used by FSBL?

For 2019.2, FSBL will be built by default to reduce the compilation time level optimization(-O0) in VITIS and we are able to debug FSBL in VItis.

14.11 What are the memory regions and registers reserved for FSBL?

Following are the memory regions and registers reserved for FSBL:
DDR region - 0x100000U (DDR_TEMP_START_ADDRESS) This is the address in DDR where bitstream will be copied temporarily.
OCM region - 0xFFFFFFF8U (BASEADDR_HOLDER) - The address holds the base address for the image boot ROM found.

14.12 Is there any order in which I have to specify bitstream in BIF file (for boot image creation)?

The first partition must be the FSBL ELF followed by the bitstream partition and then the application ELF. Bitstream is optional. FSBL does a handoff to the first application in the BIF order. The order within the BIF file is important. Bitstream must be the partition after FSBL. The bitstream is required only if the PL must be programmed.

14.13 Is USB boot mode supported in FSBL?

There is no USB boot mode support in FSBL.

14.14 What us the FSBL fallback feature and does FSBL support this?

To recover from an error condition, FSBL does a Fallback and enables BootROM to load another bootable image (the golden image that was originally present and in a known good state) if that image is present in the flash memory. FSBL updates a multiboot register and does a soft reset so that BootROM executes and loads the next present, valid image. Please refer UG821 for details on the fallback feature.

14.15 Is early handoff supported in FSBL?

There is no early handoff support in FSBL.

14.16 What are the various hooks provided in FSBL code?

Hooks are functions which can be defined by users. FSBL provides blank functions, and calls them from certain strategic locations. Below are the hooks available currently:

Hook purpose/location	Hook function name
Before PL bitstream download	FsblHookBeforeBitstreamDload()
After PL bitstream download	FsblHookAfterBitstreamDload()
Before handoff to the application	FsblHookBeforeHandoff()
During FSBL fallback	FsblHookFallback()

14.17 How boot time measurements can be done in FSBL?

Boot time (or performance) measurement is a way to measure time taken to complete certain time consuming activities (e.g. partition copy,authentication, decryption etc). In addition, overall time taken by FSBL (measured from after ps7_init completion to completion of all partitions/bitstream loading) is also provided. This feature can be enabled by defining macro FSBL_PERF (in fsbl.h). Debug prints should not be enabled when FSBL_PERF macro is defined.

14.18 What are QSPI modes support added in FSBL and reset requirements for large QSPI?

Zynq supports both linear and I/O mode QSPI. FSBL includes the ability to use linear addressing mode for Flash devices ≤128Mb and use IO mode for >128Mb devices. If a larger than 16MB QSPI flash is used, then in order to access data on the portion of the flash over 16MB, the software driver (standalone, u-boot, Linux) needs to extend the 3-bytes address storing the 4th byte into a vendor-specific QSPI register called "extended address register". Please refer AR# 64011 for QSPI reset requirements.

14.19 What are the boot image requirements when using larger than 16MB QSPI and RSA Authentication?

The BootROM uses the Linear mode to access the first 16MB of the QSPI flash to look for the boot image. There are limitations on where the boot image could be placed if larger than 16MB QSPI flash and RSA Authentication are used that the boot image cannot be placed at 0x0 offset in the flash for single x2 or x4, dual-stacked x4 and dual-parallel x4 configurations.

There are three possible work-arounds for this requirement:

- Erase the first 32KB of flash and program the boot image at 0x0 + 32KB offset (64KB in the case of dual parallel)
  The BootROM will fail booting from 0x0, will fallback and will boot from 0x0 +32KB offset (see UG585 Zynq-7000-TRM for Boot Partition Search).
- Program the boot image at 0x0 and Duplicate the Image Header at 0x0 + 16MB offset.
  The BootROM will use the Image Header at 0x0 + 16MB offset and then will boot with the boot image programmed at 0x0.
- Use only single x1 QSPI mode.

NOTE: If RSA is not used, the boot image can be placed at 0x0 even for larger than 16MB QSPI.

14.20 What are the changes required in FSBL for Execute-in-Place (XIP) option for QSPI flash memory?

Please refer to this tech-tip which intends to show how to make a DDR-Less system using Zynq-7000 SoC. It describes about executing boot loader code and application code without DDR. User can take this framework and use it for their specific applications.

Following points are covered in this tip.

- Shows FSBL XIP execution.
- Provide sample routine to do preloading of the data/code to L2-cache, in FSBL.
- Shows L2-cache Lockdown feature in Zynq-7000 SOC.
- Reference design for AXI Timer in PL.
- Shows Interrupt working concept.
- C and C++ application to run in L2-cache lock down mode.

14.21 What is Secondary Boot mode?

There is a provision to have two boot devices in the Zynq-7000 architecture. The primary boot mode is the boot mode used by BootROM to load FSBL.

The secondary boot mode is the boot device used by FSBL to load all the other partitions. Zynq-7000 AP devices support eMMC flash devices in MLC and SLC configuration as a secondary boot source. FSBL

supports loading the partitions from eMMC. This is possible only when the primary boot mode (set through the boot mode pins) is QSPI.

Use this option when there is a small QSPI flash and you would like to store all the other partitions on a larger flash memory like eMMC. In this case, place the FSBL on the QSPI flash and all the other partitions are on eMMC flash.

Secondary boot mode support from PCIe: Initially, BootROM and first stage boot loader (FSBL) are executed, and then u-boot. Loading u-boot from an external source like SD over QSPI is the primary boot mechanism.

The following application note in the link Zynq-7000 AP SOC Secondary Boot Over PCIe Techtip describes the implementation of secondary boot mechanism where the u-boot image will be transferred from an external host machine over PCI Express link to Processing System DDR memory.

14.22 What are the peripherals/IPs required for FSBL?

ADMA and any one instance of IPI are required IPs for FSBL. Hence these IPs are required in design and also should be not isolated from the processor for which FSBL is being created.

14.23 Revision History

14.23.1 What's new in 2019.2 release?

No changes.

14.23.2 What's new in 2019.1 release?

No changes.

14.23.3 What's new in 2018.3 release?

Added code to check EFUSE_SEC_EN bit and force encryption.

14.23.4 What's new in 2018.2 release?

No changes.

14.23.5 What's new in 2018.1 release?

No changes.

14.23.6 What's new in 2017.3 release?

No changes.

14.23.7 What's new in 2017.2 release?

No changes.

14.23.8 What's new in 2017.1 release?

No changes.

14.23.9 What's new in 2016.3 release?

Fabric Initialization sequence is modified to check the PL power before sequence starts and checking INIT_B reset status twice in case of failure.

14.23.10 What's new in 2016.2 release?

Added symbols to linker script to prevent linking failures in absence of spec file.

14.23.11 What's new in 2016.1 release?

PS UART code is now referred only when PS UART is present in design. This is since STDOUT_BASEADDRESS is defined even for coresight UART.
Added support for Macronix flash.
Removed the hard coded value of qspi read command and configured to pick from LQSPI_CFG register.
As xilrsa is not mandatory for zynq, remove xilrsa check while creating application in SDK.

15 Zynq-7000 Power Management

This page gives an overview of power management features and frameworks used in Zynq Linux solutions. Paths to files and documentation on this page are given relative to the Linux kernel source directory.

15.1 Table of Contents

15.2 cpufreq

The cpufreq framework is used to scale the CPU frequency. Full documentation is found in Documentation/cpu-freq/.

Summary
The framework exposes a control interface through sysfs in /sys/devices/system/cpu/cpu0/cpufreq/. This interface provides statistics (if enabled) and allows to choose a scaling governor and in case of the *userspace* governor is used to manually set a frequency.
To enable the cpufreq framework enable the following config options in your kernel configuration:

- CONFIG_THERMAL
- CONFIG_CPU_THERMAL
- CONFIG_CPU_FREQ
- CONFIG_CPU_FREQ_GOV_PERFORMANCE
- CONFIG_CPU_FREQ_GOV_POWERSAVE

- CONFIG_CPU_FREQ_GOV_USERSPACE
- CONFIG_CPU_FREQ_GOV_ONDEMAND
- CONFIG_CPU_FREQ_GOV_CONSERVATIVE
- CONFIG_CPU_FREQ_STAT
- CONFIG_CPU_FREQ_STAT_DETAILS
- CONFIG_CPUFREQ_DT

Zynq uses the generic cpufreq-dt driver is found in drivers/cpufreq/cpufreq-dt.c. The OPPs are specified in the device tree.

The default config for Zynq has the driver and its governors enabled. The default governor is *userpsace*, i.e. the frequency can be scaled using the sysfs interface and to leverage automatic frequency scaling a different governor has to be selected - either through sysfs or the kernel configuration option.

15.2.1 Known Issues

- When running at a too low frequency, USB breaks. The lowest tested and working frequency is 222.222 MHz. Highest frequency where it is known broken is 111.111 MHz
- Suspend breaks on reduced frequencies
- Due to dependencies between the CPU frequency and timers, the range of valid frequencies is limited/frequency changes may fail (see: https://github.com/Xilinx/linux-xlnx/commit/356cbded4131b067ec5fa4430c15c2fac867fa2b)
- cpufreq statistics don't work
- System clock runs slower proportionately to the difference between the default clock rate and that set by cpufreq if a timer other than the TTC (e.g. arm_global_timer) is used as clocksource

15.3 cpuidle

The cpuidle framework manages CPU idle levels. Full documentation is found in Documentation/cpuidle/.

Summary
The cpuidle drivers puts idle CPUs in a low power state when a CPU becomes idle, which means executing *wfi* on Zynq. The framework works fully autonomous and does not require the user to interact.
To enable the cpuidle framework enable the following config options in your kernel configuration:

- CONFIG_CPU_IDLE
- CONFIG_CPU_IDLE_GOV_LADDER
- CONFIG_CPU_IDLE_GOV_MENU

Zynq's cpuidle driver is found in arch/arm/mach-zynq/cpuidle.c.

15.4 Suspend

The *suspend* framework provides the interface to enter sleep states, like the well known 'suspend to disk/ RAM' on laptops. Documentation is found in Documentation/power/. In particular the files swsusp.txt and states.txt. And devices.txt for interaction with device drivers.

Summary
The architecture specific callbacks and initialization is located in arch/arm/mach-zynq/pm.c. In order to shut down the DDR subsystem the final suspend call — mach-zynq/suspend.S:zynq_sys_suspend() — is moved into OCM and executed from there. Below is a list of tested wakeup devices. Depending on the requirements of the wakeup device enabled, the IOPLL might not be bypassed. All Zynq peripheral drivers employ clock gating during suspend. The current implementation supports suspend to RAM only, which can be entered by executing

```
echo mem > /sys/power/state
```

To enable the suspend framework enable the following config options in your kernel configuration:

- CONFIG_PM_SLEEP

15.4.1 Wake on UART

Suspend works with and without the 'no_console_suspend' kernel parameter. The UART console can also be used as wakeup device

```
echo enabled > /sys/devices/amba.0/e0001000.serial/tty/ttyPS0/power/wakeup
```

15.4.2 Wake on GPIO

Export the wanted GPIOs

```
echo <n> > /sys/class/gpio/export
```

Use <n> = 12 and 14 for the push buttons on a zc702

Enable interrupts for the wanted GPIO pins

```
echo (rising|falling|both) > /sys/class/gpio/gpio<n>/edge
```

Make GPIO the wake-up device

```
echo enabled > /sys/devices/amba.0/e000a000.gpio/power/wakeup
```

15.4.3 Wake on Debugger

Suspend can be left through *xmd* by connecting, stopping and starting CPU0.

```
xmd% connect arm hw
xmd% stop
xmd% continue
```

15.4.4 Removable Media

Removable media (USB devices, sdcard, etc.) are usually ejected from the system before suspend is entered. Therefore it is highly discouraged to suspend the system while a file system on such a removable device is mounted. To make suspend work anyway when running with a rootfs from sdcard the config option MMC_UNSAFE_RESUME can be enabled. This prevents the removable MMC device from being ejected, but **NEVER** exchange removable devices during suspend with this config option set.

15.4.5 Known Issues

- suspend breaks when cpufreq is used to scale the CPU frequency
- the DDR PLL is bypassed without taking into account possible peripherals sourced by it
- HW running independent from Linux is not taken into account when DDR is shut down. I.e. additional means have to be employed to make sure it is safe to shut down DDR before suspend is triggered.

15.5 Hardware Monitoring

15.5.1 XADC

The Xilinx analog mixed signal module, referred to as the XADC, allows temperature and voltage monitoring. To compile a kernel with a driver for the XADC enable the following kernel config option:

- CONFIG_XILINX_XADC

The driver exposes an interface through sysfs in /sys/devices/amba.0/f8007100.ps7-xadc/iio:device0.
A to scale measurement is retrieved by the following calculation:

```
(foo_raw + foo_offset) * foo_scale
```

Further information is available in:
- device tree binding

15.5.2 UCD9248

TI's UCD9248 PWM controllers are commonly used on Xilinx platforms like the zc702. The Linux driver for these controllers allows voltage and current monitoring through a sysfs interface exposed in /sys/devices/amba.0/e0004000.ps7-i2c/i2c-0/i2c-8/8-003(4|5|6). The driver is documented in Documentation/hwmon/ucd9200
To compile a kernel with this driver enable the following kernel config option:
- CONFIG_PMBUS
- CONFIG_SENSORS_PMBUS
- CONFIG_SENSORS_UCD9200

15.5.3 UCD90120

TI's UCD90120 power supply sequencer and monitors are commonly used on Xilinx platforms like the zc706. The Linux driver for these controllers allows voltage and current monitoring through a sysfs interface exposed in /sys/devices/amba.0/e0004000.ps7-i2c/i2c-0/i2c-8/8-0065. The driver is documented in Documentation/hwmon/ucd9000
To compile a kernel with this driver enable the following kernel config option:
- CONFIG_PMBUS
- CONFIG_SENSORS_PMBUS
- CONFIG_SENSORS_UCD9000

15.6 Changelogs

2017.2
No change

2017.3
No change

15.7 Related Links

Controlling FCLKs in Linux
Zynq-7000 AP SoC Measuring ZC702 Power using Linux Application Tech Tip
Zynq-7000 AP SoC Measuring ZC702 Power using TI Fusion Power Designer Tech Tip
Zynq-7000 AP SoC Measuring ZC702 Power using Standalone Application Tech Tip

16 Zynq-7000 BIST Guide

This page provides a walkthrough of the Built-In Self Test (BIST) for Zynq-7000 evaluation boards. The BIST may be used to verify board functionality.

16.1 Table of Contents

16.2 Introduction

This guide applies to the following boards. User guides for each board are also linked below.

- ZC702
- ZC706

16.3 Boot Modes

The following table can be used to determine mode switch configuration. The switches used to set configuration mode depends on board in use. Refer individual board callouts and boot mode tables below to identify appropriate switch.

16.3.1 ZC702

Boot Mode	Setting	SW16 (SW16.1-SW16.5)
JTAG	00000	OFF, OFF, OFF, OFF, OFF

Boot Mode	Setting	SW16 (SW16.1-SW16.5)
Independent JTAG	10000	ON, OFF, OFF, OFF, OFF
QSPI	00010	OFF, OFF, OFF, ON, OFF
SD	00110	OFF, OFF, ON, ON, OFF

PL JTAG Programming Option	Setting	SW10[1:2]
None	00	OFF, OFF
USB-to-JTAG	01	OFF, ON,
Cable connector J2	10	ON, OFF
JTAG Header J58	11	ON, ON, OFF, ON

16.3.2 ZC706

Boot Mode	Setting	SW11 (SW11.1-SW11.5)
JTAG	00000	OFF, OFF, OFF, OFF, OFF
Independent JTAG	10000	ON, OFF, OFF, OFF, OFF
QSPI	00010	OFF, OFF, OFF, ON, OFF
SD	00110	OFF, OFF, ON, ON, OFF

PL JTAG Programming Option	Setting	SW[1:2]
None	00	OFF, OFF
USB-to-JTAG (US30)	01	OFF, ON,

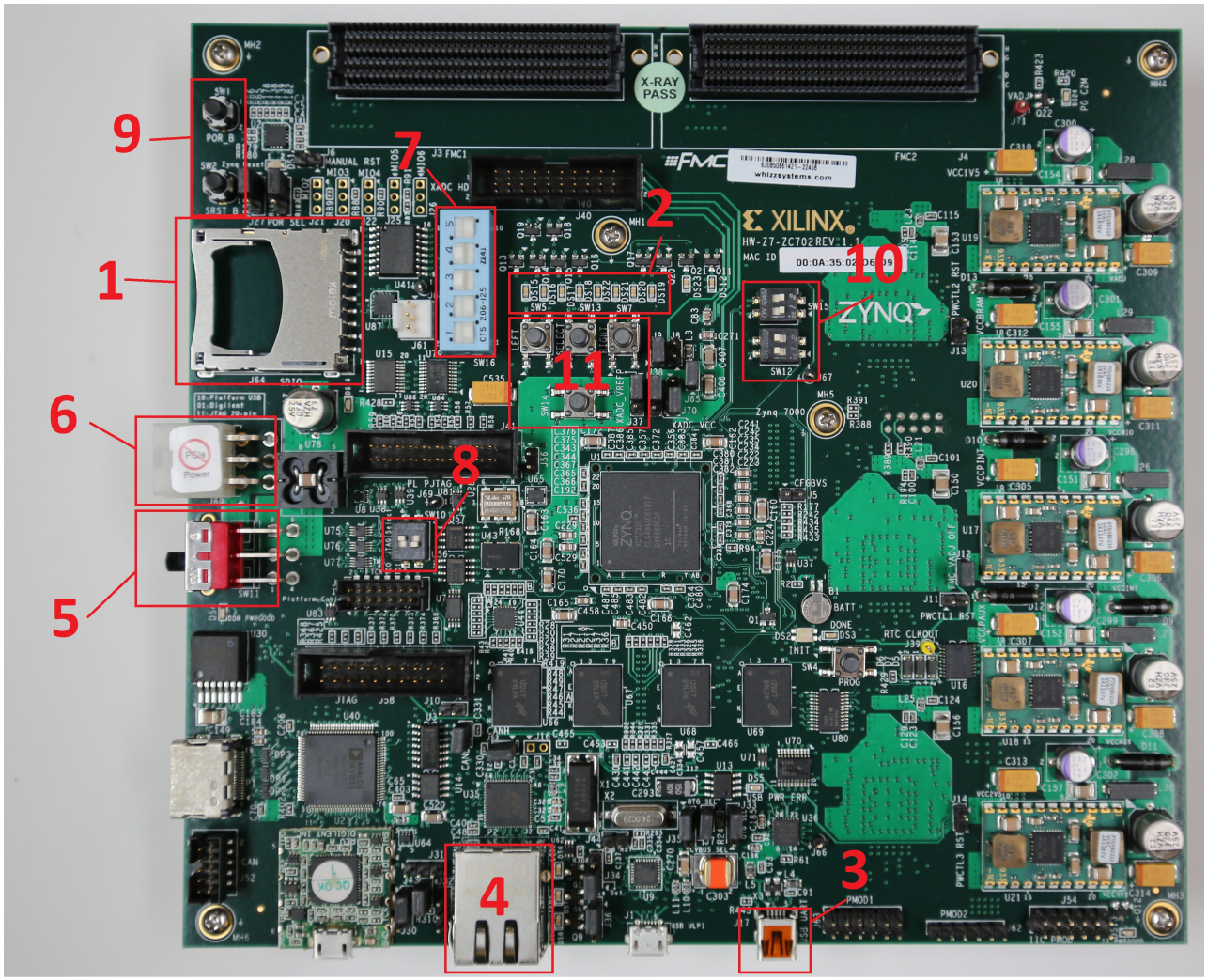
PL JTAG Programming Option	Setting	SW[1:2]
Cable Connector J3	10	ON, OFF
JTAG Header J62	11	ON, ON, OFF, ON

16.4 Board Callouts

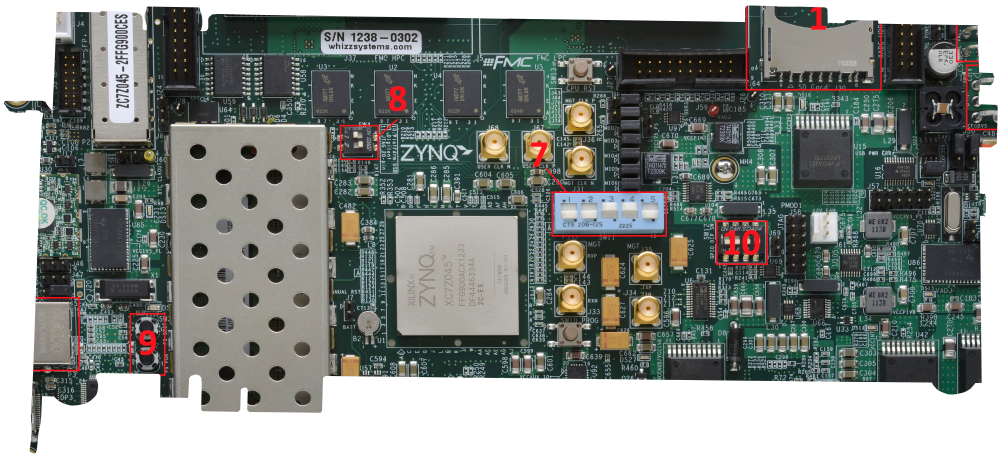
More comprehensive component lists available in board user guides.

1. SD Card Interface
2. User LEDs
3. USB to UART Interface
4. Ethernet Interface
5. Power On/Off Switch
6. Power Connector
7. Mode Configuration Switches
8. PL JTAG Programming Switches
9. POR_B and SRST_B Switches
10. PL DIP Switches
11. Pushbuttons

16.4.1 ZC702



16.4.2 ZC706



16.5 Built In Self-Test (BIST)

- Note: Board layout will vary. Reference callouts above to help with setup.
- **The ZC706 BIST requires supplemental files.** They can be downloaded from the following link: ZC706 BIST ZIP

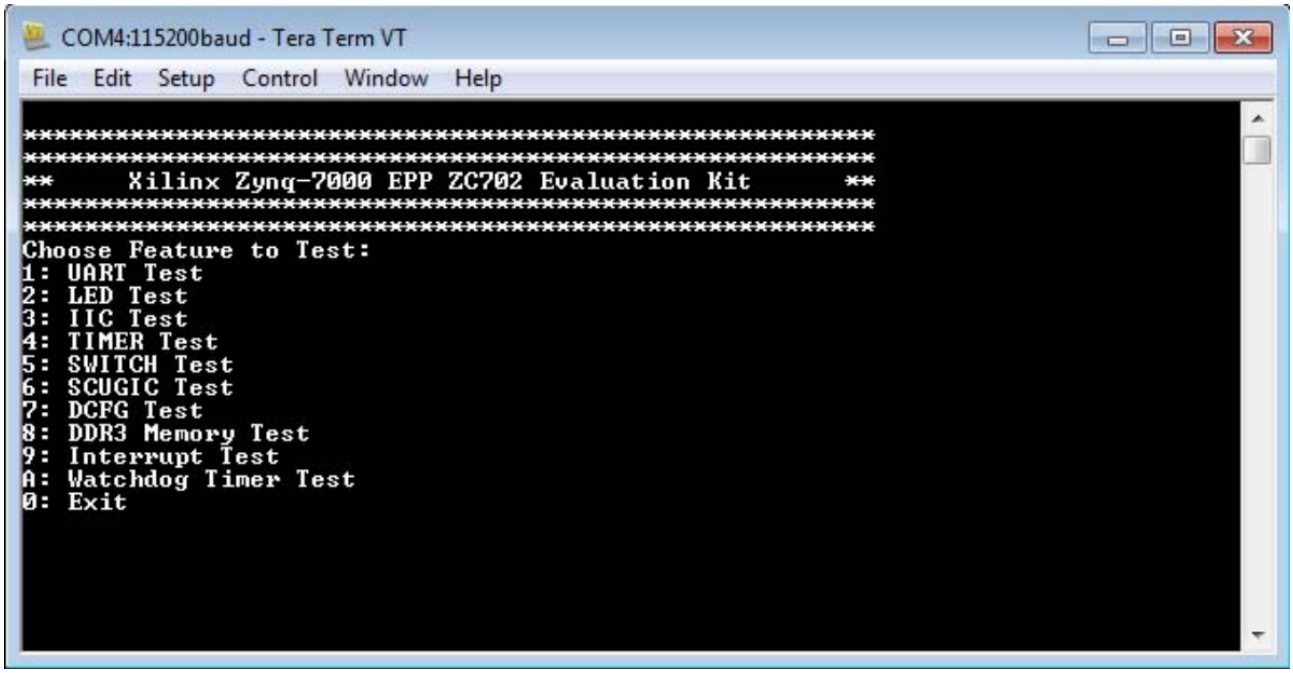
16.5.1 Starting the Board

- Verify hardware setup—see “General Board HW Setup/Debug” page in related links. Board should be powered off at the start of tutorial.
- Set mode switch to QSPI according to the tables above.
- Set up your terminal emulator (see instructions for Tera Term setup in “General Board HW Setup/ Debug” page linked below).
    - For serial setup, set *baud rate* to 115200, *data* to 8 bit, *parity* to none, *stop* to 1 bit, and *flow control* to none.
    - Individual built-in tests will be run and verified through the terminal display.
- Connect to PC via board’s USB UART interfaces, and power board by connecting to 6-pin power supply. Power on board with switch.
- To begin BIST program:
    - ZC702: With terminal display open, press SW1 (POR_B) to reinitialize board configuration.
    - ZC706: With terminal display open, cycle board power.

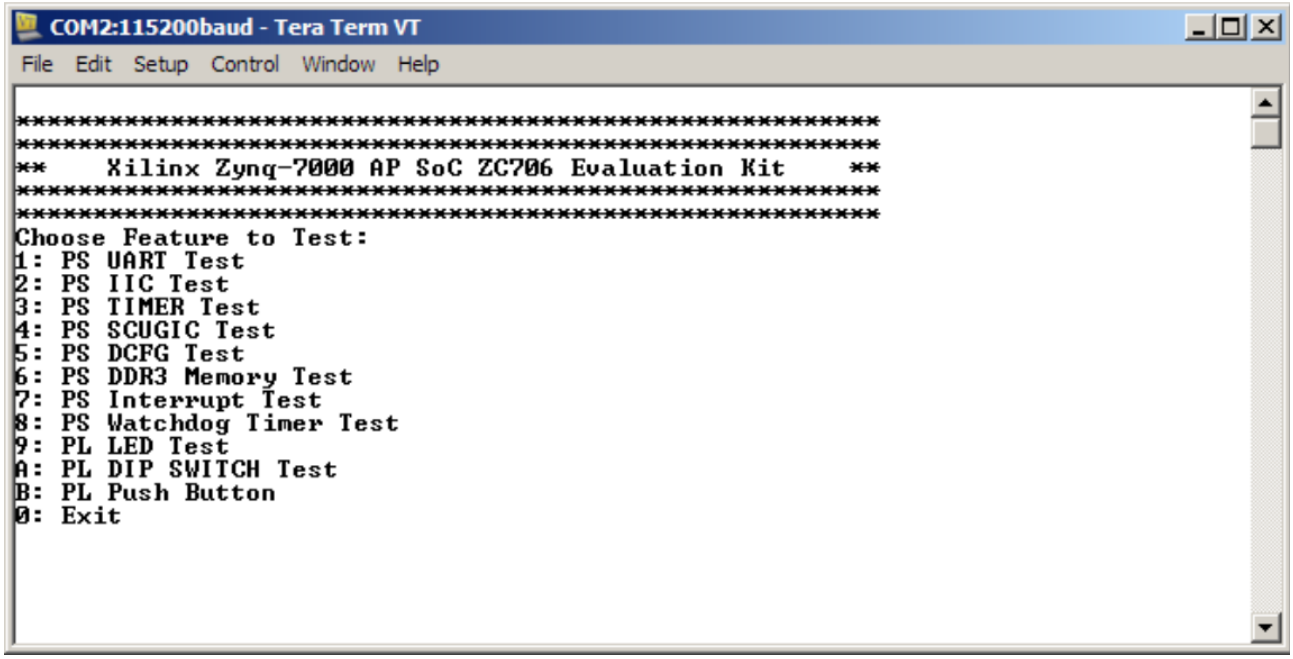
16.5.2 Running BIST

- The BIST display should appear as below.

ZC702:



ZC706:



- Tests can be entered into the window to run the corresponding test (e.g., typing “1” will run the UART test). A success message will print when a test is passed.

- For ZC702 BIST: SWITCH test (5) DIP switches SW15-1 and SW15-2 are tied to SW13 and SW14. SW15-1 and SW15-2 can be set to OFF to use pushbuttons SW13 and SW14 for test, or be kept ON to be used in the test.

16.6 Related Links

- Board Schematics (Links below lead to downloads at the Xilinx website)
  - ZC702
  - ZC706
- Board Product Pages
  - ZC702
  - ZC706
- Master AR list (link)
  - Zynq-7000 ZC706 Debug Checklist (AR 54013)
  - Zynq-7000 ZC702 Debug Checklist (AR 54012)
- Zynq-7000 Targeted Reference Designs (TRD) Page
- Xilinx Evaluation Boards Help Forum

17 Zynq Qt and Qwt Base Libraries-Build Instructions

Table of Contents

17.1
1 Introduction

This wiki page summarizes the build steps for Qt 4.7.3 and Qwt 6.0.1 libraries as used in the Zynq Base TRD 2015.2 or older.
The Zynq Base TRD version 2015.4 uses Qt 5.4.2 and Qwt 6.1.2. Please refer to this wiki page for build instructions.

17.2
2 Prerequisites

This tutorial requires the ARM GNU tools, which are part of the Xilinx Software Development Kit (SDK), to be installed on your host system. Specify the ARM cross-compiler by setting the CROSS_COMPILE environment variable and adding the cross-compiler to your PATH. Refer to the Zynq Tools wiki page for more information on how to set up the tool chain and known issues.

Note: These instructions currently require a Linux host for building.

```
bash> export CROSS_COMPILE=arm-xilinx-linux-gnueabi- bash> export PATH=/path/to/
cross/compiler/bin:$PATH
```

For simplicity we define two environment variables in this tutorial:

- ZYNQ_QT_BUILD refers to the Qt build area
- ZYNQ_QT_INSTALL refers to the Qt install area

```
bash> export ZYNQ_QT_BUILD=/path/to/qt/build bash> export ZYNQ_QT_INSTALL=/path/to/
qt/install bash> export PATH=$ZYNQ_QT_INSTALL/bin:$PATH
```

17.3
3 Cross-Compile Qt4 for Embedded Linux

Qt for Embedded Linux is a C++ framework for GUI and application development for embedded devices. It runs on a variety of processors, usually with Embedded Linux. Qt for Embedded Linux provides the standard Qt API for embedded devices with a lightweight window system. Qt for Embedded Linux applications write directly to the framebuffer, eliminating the need for the X Window System and saving memory.

17.3.1
3.1 Download the Qt Source Archive

Download the Qt sources and extract the archive to your Qt build area.

```
bash> cd $ZYNQ_QT_BUILD bash> tar xzfv qt-everywhere-opensource-src-4.7.3.tar.gz
bash> cd qt-everywhere-opensource-src-4.7.3
```

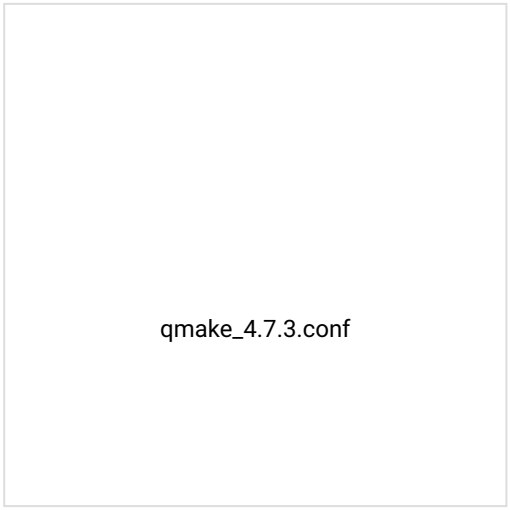
17.3.2
3.2 Prepare a mkspec

Before we can do the configuration for the target system, we will need a set of mkspecs that tells qmake which tool chain it should reference when it creates the Makefiles. In this example we will provide an mkspec to go along with the ARM GNU tools.

The mkspec consists of two files:

- qmake.conf – This is a list of qmake variable assignments that tells qmake what flags to pass through to the compiler, which compiler to use etc
- qplatformdefs.h – This is a header file with various platform-specific #includes and #defines. Often this just refers to an existing qplatformdefs.h file from another generic mkspec

Download the qmake configuration file from the link below to override the existing linux-arm mkspec.




```
bash> cp /path/to/downloaded/qmake_4.7.3.conf mkspecs/qws/linux-arm-gnueabi-g++/
qmake.conf
```

The corresponding qplatformdefs.h file just refers to an existing generic one, so nothing needs to be done here.

17.3.3
3.3 Configure the Target Build

We are now ready to configure Qt to use our mkspec and hence use our cross-compiling toolchain.

```
bash> ./configure \ -embedded arm \ -xplatform qws/linux-arm-gnueabi-g++ \ -little-
endian \ -opensource \ -host-little-endian \ -confirm-license \ -nomake demos \
-nomake examples \ -prefix $ZYNQ_QT_INSTALL
```

Here is some information on some of the options that appear:

- -xplatform <mkspec files to use> – Cross compile for the target platform using the environment specified in the mkspec files
- -embedded <target CPU architecture> – The CPU architecture of the target platform

- -prefix <path> – The path to install the cross-compiled Qt to
- -confirm-license – Lazy option to save agreeing to license agreement during configure process

17.3.4
3.4 Build and Install

Run make to build the cross-compiled target version of Qt.

```
bash> make
```

Once the build has completed it is time to install Qt. You may need to su to root to do this part depending upon what prefix you configured the build with.

```
bash> su - #if you need root access to be able to install bash> make install
```

17.4
4 Cross-Compile Qwt – Qt Widgets for Technical Applications

The Qwt library contains GUI Components and utility classes which are primarily useful for programs with a technical background. Beside a 2D plot widget it provides scales, sliders, dials, compasses, thermometers, wheels and knobs to control or display values, arrays, or ranges of type double.

17.4.1
4.1 Download the Qwt Source Archive

Download the Qwt sources and extract the archive to your Qt build area.

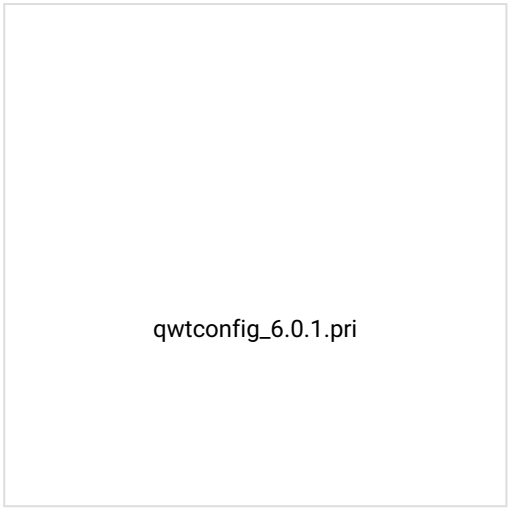
```
bash> cd $ZYNQ_QT_BUILD bash> tar xjfv qwt-6.0.1.tar.bz2 bash> cd qwt-6.0.1
```

Note: Make sure you configure, compile and install the qwt libraries in the same shell as the Qt installation as some Qt environment variables are reused for this build.

17.4.2
4.2 Configure the Build using qmake

Projects are described by the contents of project files (.pro). Files that end with the suffix .pri are included by the project files and contain definitions that are common for several project files. The information within these is used by qmake to generate a Makefile containing all the commands that are needed to build each project.

Download the config file from below and replace the existing one.




```
bash> cp /path/to/downloaded/qwtconfig_6.0.1.pri qwtconfig.pri bash> qmake qwt.pro
```

17.4.3
4.3 Build and Install

Run make to build the cross-compiled target version of Qwt.

```
bash> make
```

Once the build has completed it is time to install Qwt. You may need to su to root to do this part depending upon what prefix you configured the build with.

```
bash> su - #if you need root access to be able to install bash> make install
```

17.5
5 Add the GNU Standard C++ Library

In order to build and run your Qt application, the GNU Standard C++ Library of the ARM GNU Tools needs to be copied to the Qt install area's lib directory.

```
bash> cp -P /path/to/cross/compiler/arm-xilinx-linux-gnueabi/libc/usr/lib/libstdc+
+.so* \ $ZYNQ_QT_INSTALL/lib
```

17.6
6 Create a File System Image with Pre-Compiled Qt/Qwt Libraries

NOTE: This step is optional and is needed for Zynq Base TRD 14.x version for copying QT libraries on a formatted ext2 filesystem.
The final step shows how to create an image file of the Qt install area that can be copied onto an SD card and then mounted on the target system. These libraries are required to run the Qt application on the target system. First we create an empty image file of size 80MB, then we format the image with the ext2 file system.

```
bash> cd $ZYNQ_QT_BUILD bash> dd if=/dev/zero of=qt_lib.img bs=1M count=80 bash>
mkfs.ext2 -F qt_lib.img
```

Now we just need to mount the image on our host machine, copy over the libraries from the Qt install area and we are done. You may need to su to root to do this part.

```
bash> su - #if you need root access to be able to install bash> chmod go+w qt_lib.img
bash> mount qt_lib.img -o loop /mnt bash> cp -rf $ZYNQ_QT_INSTALL/* /mnt bash> chmod
go-w qt_lib.img bash> umount /mnt
```

17.7
7 Web Sources

Qt for Embedded Linux Reference Documentation
Qwt - Qt Widgets for Technical Applications User's Guide

17.8
8 Licensing

All files provided by Xilinx may be used under the terms of the GNU General Public License (GPL) version 3.0. Please refer to Qt License Options and Qwt License Version 1.0 for details.

18 Zynq-7000 AP SoC SATA part 1 – Ready to Run Design Example Setup

18.1 Zynq-7000 AP SoC SATA part 1 – Ready to Run Design Example Setup

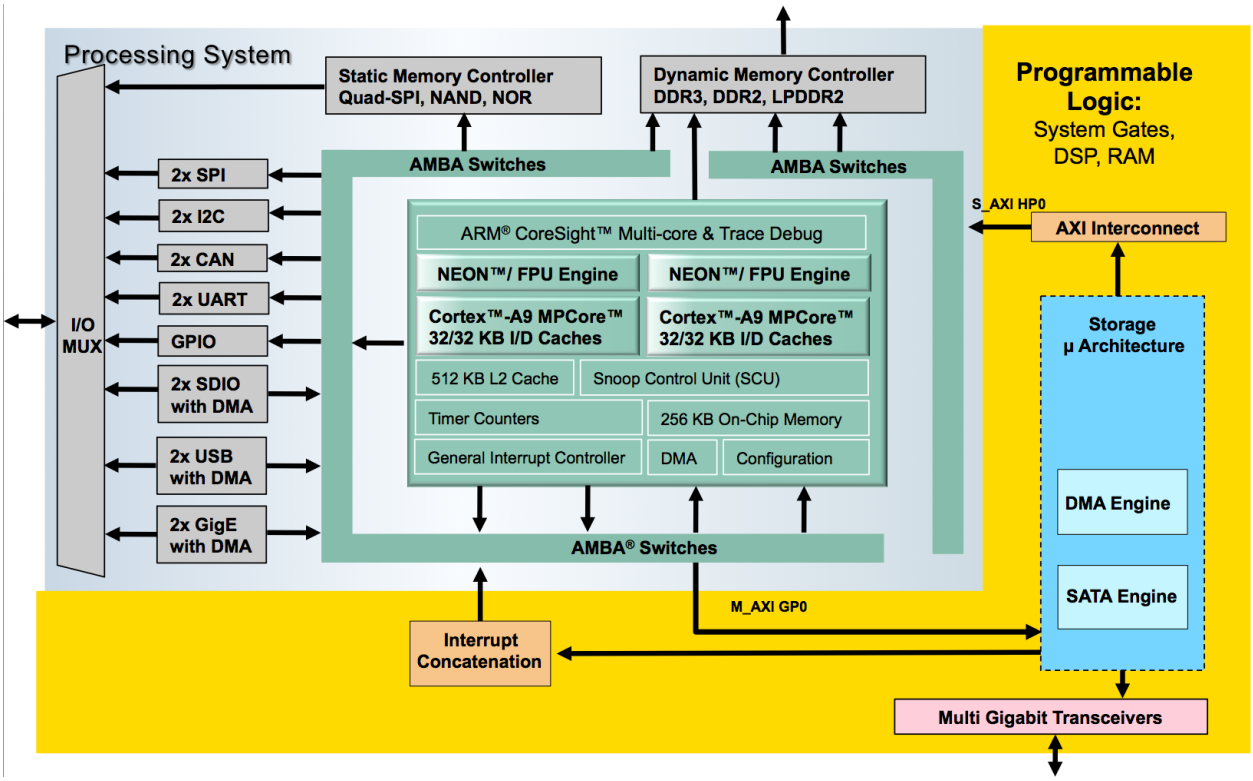
18.2 Zynq SATA Storage Extension

18.3 Table of Contents

The Zynq SATA Storage Extension (Zynq SSE) is a fully integrated and pre-validated system stack comprising 3rd-party SATA Host Controller and DMA IP cores from ASICS World Services, a storage micro-architecture from Missing Link Electronics (MLE), Xilinx PetaLinux, and an Open Source SATA Host Controller Linux kernel driver, also from MLE. Zynq SSE utilizes the Xilinx GTX Multi Gigabit Transceivers to deliver SATA I (1.5Gbps), SATA II (3.0Gbps), or SATA III (6 Gbps) connectivity. The Zynq SSE is delivered as a complete reference design for the Xilinx Zynq-7000 SoC (Zynq), and effectively extends Zynq with one single SATA host port for HDD and SSD storage connectivity. This Technical Tip shows how to setup the Zynq SSE. After going through the steps described herein, you will have a working Linux System running on the Zynq with an attached SATA HDD or SSD.
For sales and technical support please visit us at:
http://MLEcorp.com/ZynqSSE

18.3.1 Block Diagram

=The block diagram shown below gives an overview over the Zynq SSE reference design: Within the Zynq Programmable Logic (PL) the MLE storage micro-architecture instantiates the DMA and the SATA Host Controller IP blocks. The storage micro-architecture itself interfaces with the Zynq Processing System (PS) via the high-performance AXI HP0 slave port. The ARM A9 in the PS runs Xilinx PetaLinux and the SATA Linux kernel driver.



18.3.2 Implementation

Implementation Details	
Design Type	PS+PL
SW Type	Linux (Petalinux)

CPUs	2 CPUs 700 MHz
PS Features	DDR, USB, UART, ETHERNET
PL Cores	ASICS. WSSATAIP

Boards/Tools	Avnet MiniITX Z045
Xilinx Tools Version	Vivado 2014.1, PETALINUX 2013-2

Other Details	SATA SSD (including Cable and Power Supply), SD-Card

Address Map			
	Base Address	Size	Interface
SATA IP	0x41000000	4K	S AXI
DMA IP	0x41010000	4K	S AXI, M AXI

Files Provided	
SD Card Image	All Files needed to run the ZYNQ SSE

For the available Project to built your ZYNQ SSE go to http://mlecorp.com/zynqsse

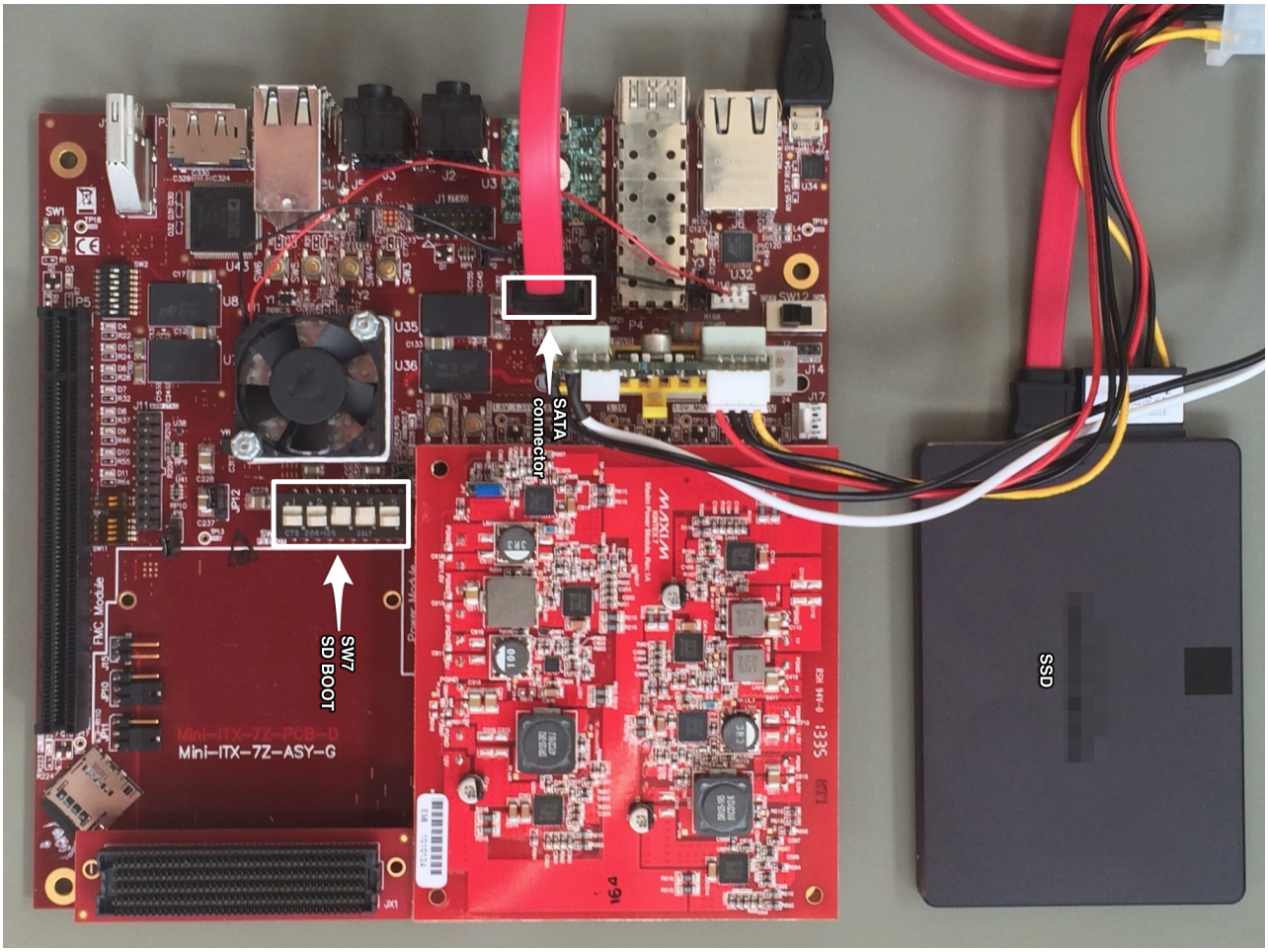
18.3.3 Step by Step Instructions

Hardware needed:

- Avnet Mini ITX Z045 Board (including Power supply) Link
- Micro USB Cable for USB Console
- Supported SSD (for Example Samsung 840)
- Power Supply for SSD
- Micro SD Card (2 GB or bigger)
- PC for UART console
  Software Assembly:
  1. Format the SD Card using FAT32 File system
  2. Put image.ub and BOOT.bin into the root directory of the SD Card
  Hardware Assembly:

1. Insert the SD Card into the Avnet Mini ITX board's SD Slot
2. Connect the micro USB cable to the UART port of the Avnet Mini ITX board and the USB Port of your PC
3. Connect the SSD to the SATA connector on the Avnet Mini ITX board using the SATA Cable.
4. Connect the SSD to power
5. Switch the Avnet Mini ITX board's SW7 to SD Boot mode (as shown in picture)
6. Connect the Avnet Mini ITX board to the Power Supply

The system should now look like in the supplied image



Startup:

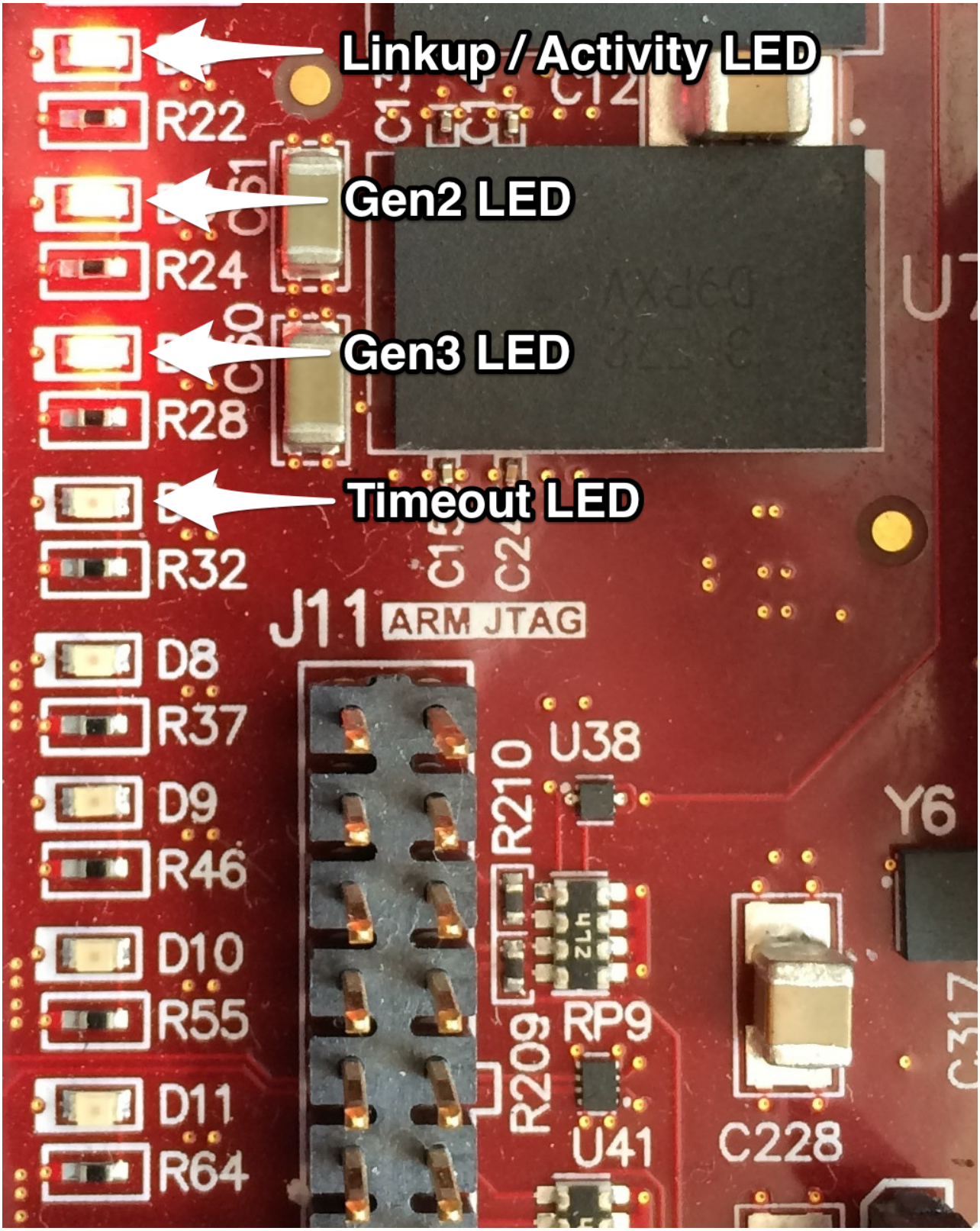
1. Switch on the Avnet Mini ITX board
2. On the PC open a Serial Terminal on the new serial port using the settings 115200 Baud 8N1

3. Observe the Linux System booting

4. After some time you should see a screen similar to the screenshot in the Expected Results.
5. Login using

• Login:root
• Password: root

18.3.4 LED Description

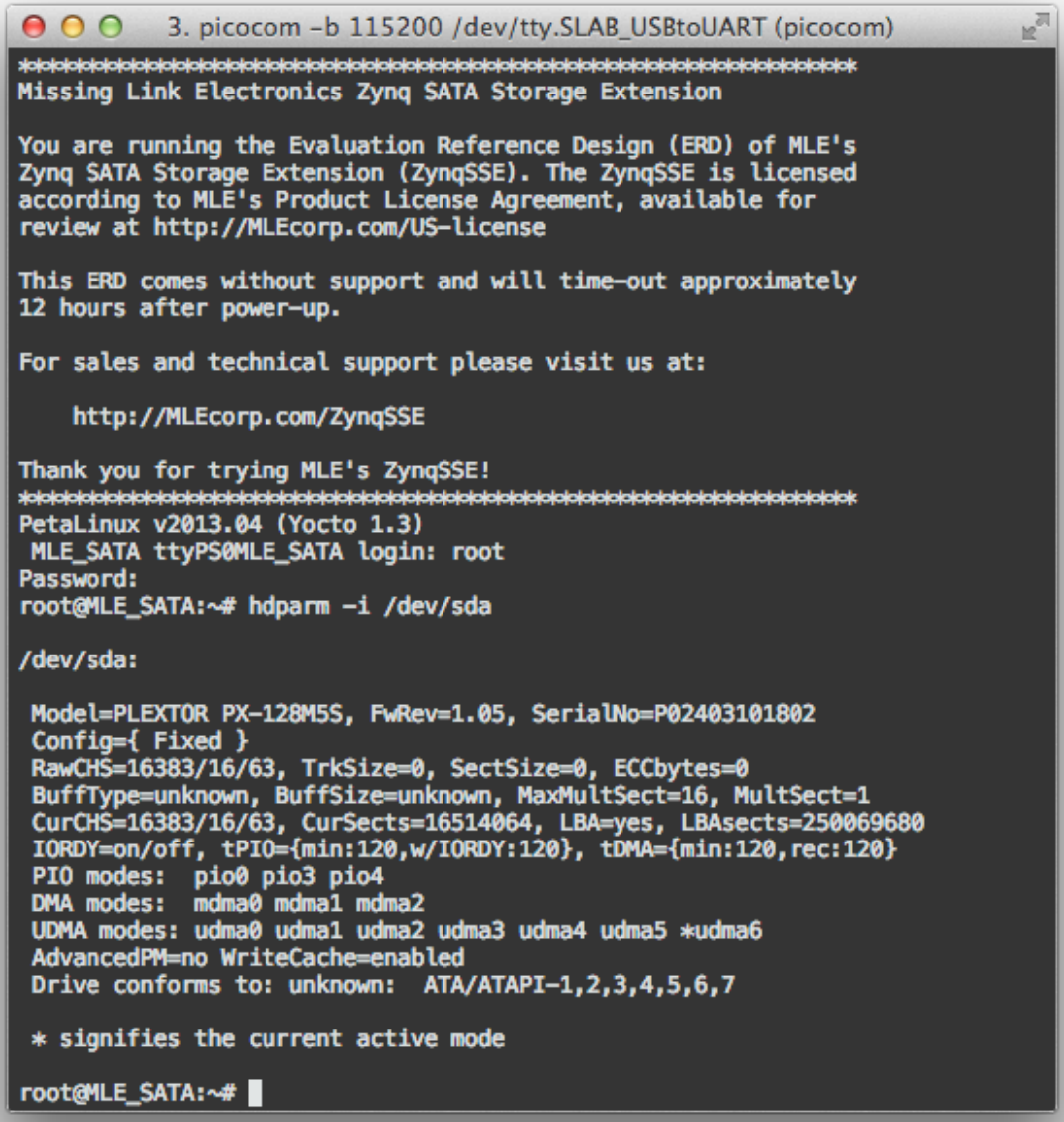


Status LEDs can be found next to the PCIe connector. The associated meanings can be seen in the following table and in the Image below.

18.3.5 Expected Results

As a result you should be having a running Linux system on the Zynq board. The UART Console output should be similar to the screenshot below.
The Evaluation Reference Design (ERD) of the Zynq SSE comprises a hardware license management which allows to run full SATA functionality for up to 12 hours after power-up. After approximately 12 hours the evaluation expires, which is indicated by illuminating the LED 'timeout'. You will also notice that the Linux kernel driver informs you of having lost the SATA link to the SSD/HDD.

D7	Timeout LED, indicates the Timeout of the IP core.
D6	Gen3 Link, indicates 6 GBit/s connection
D5	Gen2 Link, indicates 3 GBit/s connection
D4	Linkup and Activity, Led will light up on Linkup and will go dark during data transfers



18.4 Zynq-7000 AP SoC SATA part 2 – Ready to Run Design Example Benchmarking

18.4.1 Zynq-7000 AP SoC SATA part 2 – Ready to Run Design Example Benchmarking

18.4.2 ZYNQ-SSE- Benchmarks for the Avnet Mini-ITX

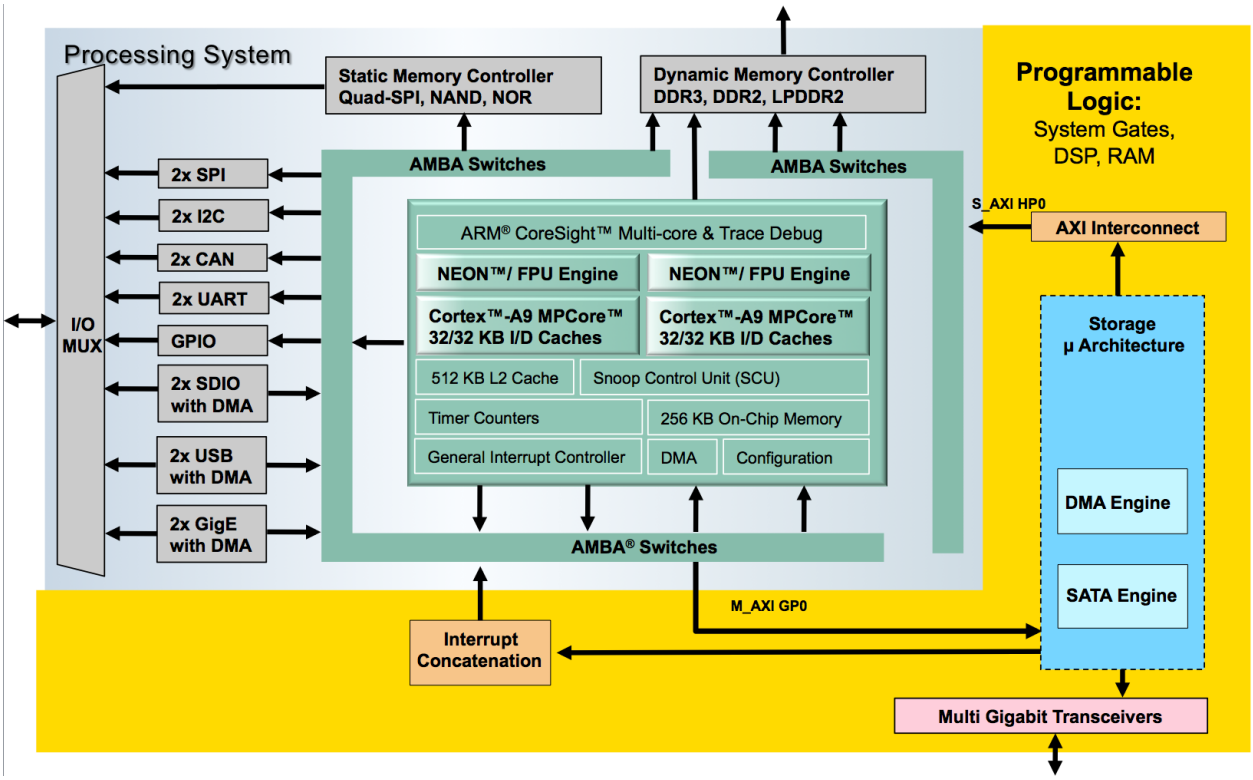
18.4.3 Table of Contents

For the evaluation of Zynq SSE Missing Link Electronics (MLE) supports the Avnet Zynq Mini-ITX board. The Zynq SATA Storage Extension (Zynq SSE) is a fully integrated and pre-validated system stack comprising 3rd-party SATA Host Controller and DMA IP cores from ASICS World Services, a storage micro-architecture from MLE, Xilinx PetaLinux, and an Open Source SATA Host Controller Linux kernel driver, also from MLE. Zynq SSE utilizes the Xilinx GTX Multi Gigabit Transceivers to deliver SATA I (1.5 Gbps), SATA II (3.0 Gbps), or SATA III (6 Gbps) connectivity. The Zynq SSE is delivered as a complete reference design for the Xilinx Zynq-7000 SoC (Zynq), and effectively extends Zynq with one single SATA host port for HDD and SSD storage connectivity. This Technical Tip shows the Benchmark results you will see after correctly setting up the system as shown in the techtip: Zynq SATA Storage Extension. Team MLE has spent significant efforts to try and test all aspects of Zynq SSE. However, if you feel that you encounter something not right, or if you do have any questions, please do not hesitate to contact us.
The best way to contact MLE is to fill out the Contact Request Form at
>http://MLEcorp.com/ZynqSSE

18.4.3.1 Block Diagram

The block diagram shown below gives an overview over the Zynq SSE reference design: Within the Zynq Programmable Logic (PL) the MLE storage micro-architecture instantiates the DMA and the SATA Host Controller IP blocks. The storage micro-architecture itself interfaces with the Zynq Processing System (PS) via the high-performance AXI HP0 slave port. The ARM A9 in the PS runs Xilinx PetaLinux and the SATA Linux kernel driver.

18.4.3.2 Implementation



Implementation Details	
Design Type	PS + PL
SW Type	Linux (Petalinux)
CPUs	2 CPUs 700 MHz
PS Features	DDR, USB, UART, ETHERNET
PL Cores	ASICS.WS SATA IP
Boards/Tools	Avnet Mini ITX Z045
Xilinx Tools Version	Vivado 2014.1, PETALINUX 2013-2
Other Details	SATA SSD(including Cable and Power Supply), SD-Card

Address Map			
	Base Address	Size	Interface
SATA IP	0x41000000	4K	S AXI
DMA IP	0x41010000	4K	S AXI, M AXI

Files Provided	
BOOT.bin	Compilation of Bitstream, FSBL and U-Boot
Image.ub	Linux Ramdisk Image

MLE's Zynq SATA Storage Extension (Zynq SSE) is a fully integrated and pre-validated system stack comprising 3rd-party SATA Host Controller and DMA IP cores from ASICS World Services, a storage micro-architecture from MLE, Xilinx PetaLinux, and an Open Source SATA Host Controller Linux kernel driver, also from MLE. Zynq SSE utilizes the Xilinx GTX Multi Gigabit Transceivers to deliver SATA I (1.5 Gbps), SATA II (3.0 Gbps), or SATA III (6 Gbps) connectivity.
The Zynq SSE is delivered as a complete reference design for the Xilinx Zynq-7000 SoC (Zynq), and effectively extends Zynq with one single SATA host port for HDD and SSD storage connectivity.
This Technical Tip shows the Benchmark results you will see after correctly setting up the system as shown in the techtip: Zynq SATA Storage Extension.

18.4.3.3 Step by Step instructions

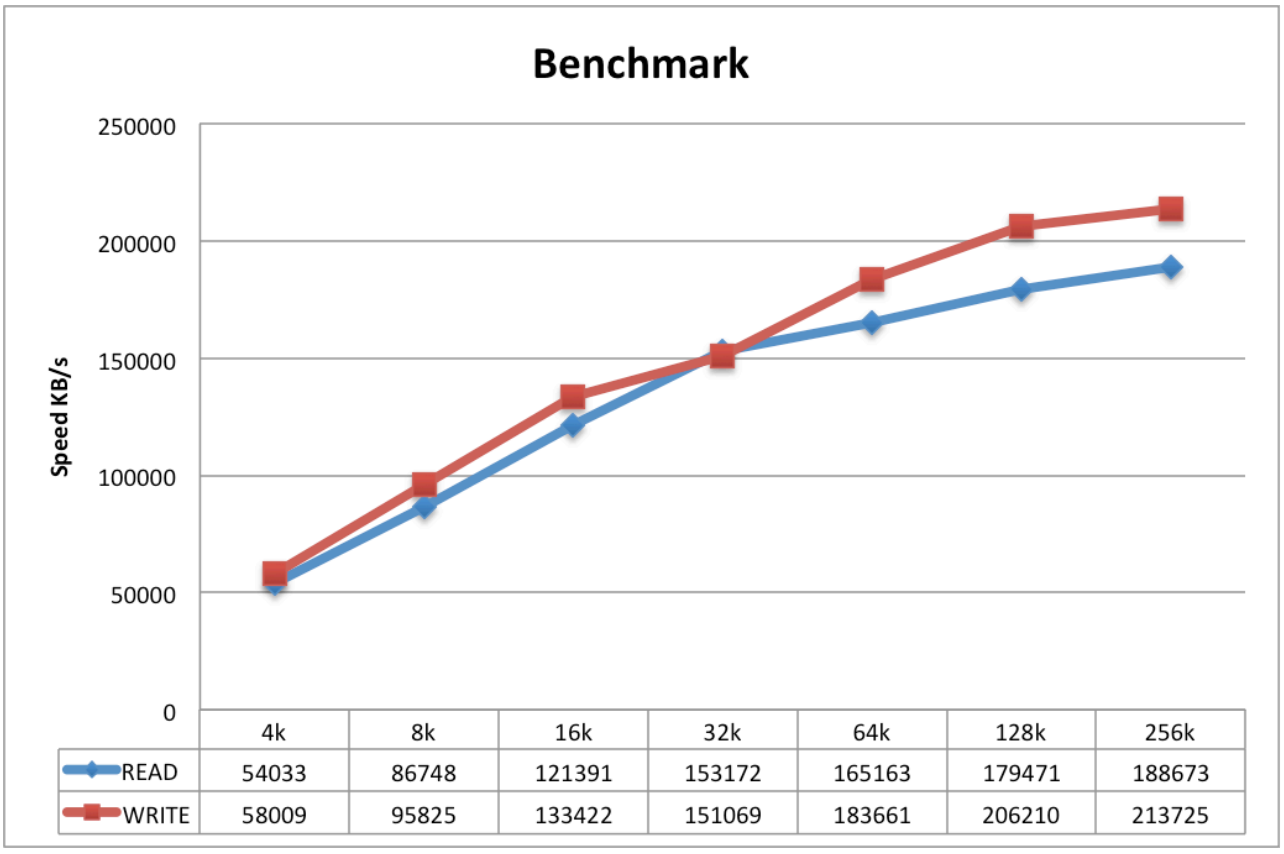
In front of Benchmarking the System, the Tech Tip "Zynq- Sata Storage Tech Tip" should be executed to have a working system. For Benchmarking the System, a tool called FIO is used. FIO is a tool that will spawn a number of threads or processes doing a particular type of I/O action as specified by the user. The typical use of FIO is to write a job file matching the I/O load one wants to simulate. We will now execute FIO with different Block sizes to see the impact on the Read and Write speeds of the system. Apply the following commands to the running SATA system, varying the Block size by using different values from 4k to 16M.
Read:

```
fio -ioengine=sync -direct=1 -rw=read -runtime=10 -filename=/dev/sda -size=1g
-name=job1 -numjobs=4 -bs=4k
```

Write:

```
fio -ioengine=sync -direct=1 -rw=write -runtime=10 -filename=/dev/sda -size=1g
-name=job1 -numjobs=4 -bs=4k
```

18.4.3.4 Expected Results



	4k	8k	16k	32k	64k	128k	256k
READ	54033	86748	121391	153172	165163	179471	188673
WRITE	58009	95825	133422	151069	183661	206210	213725

Figure 1: Benchmark on a -2 FPGA at 700 MHZ CPU speed

18.5 Zynq-7000 AP SoC SATA part 3 – Ready to Run NAS Design Example Setup

18.5.1 Zynq-7000 AP SoC SATA part 3 – Ready to Run NAS Design Example Setup

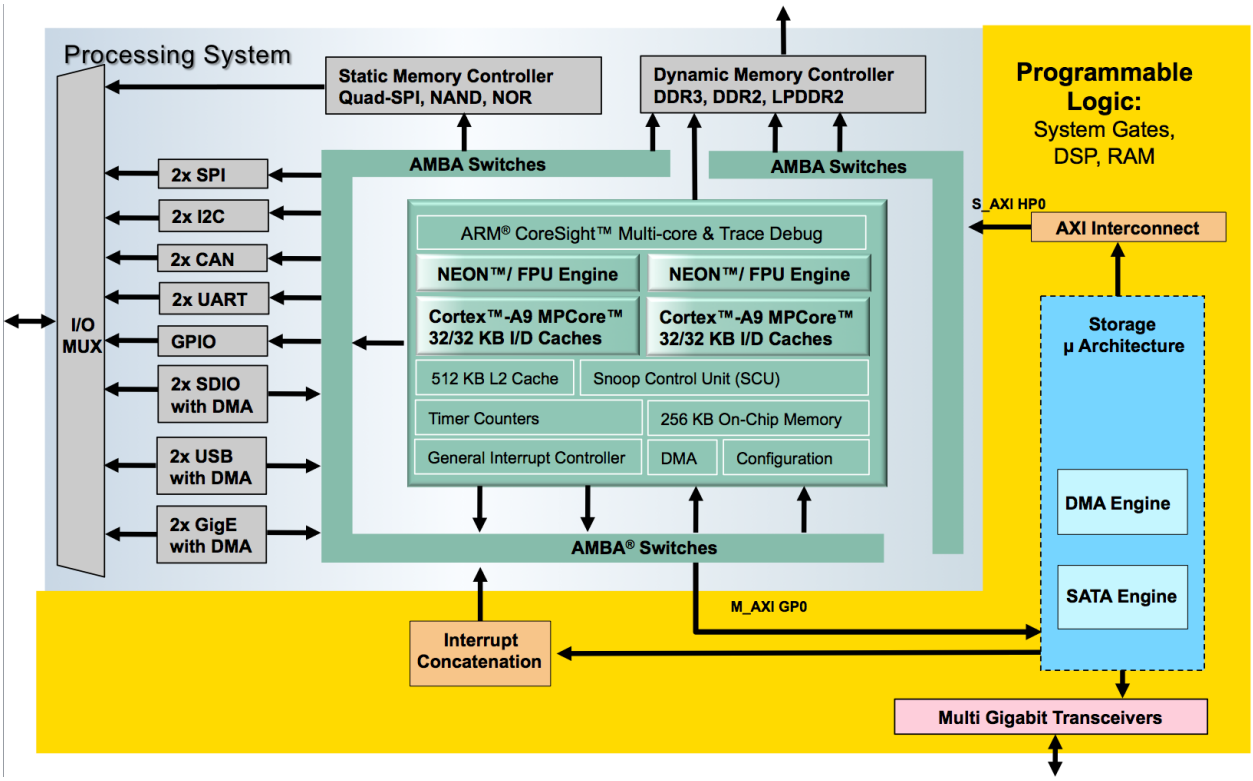
18.5.2 Zynq SSE for Network-Attached Storage for the Avnet Mini-ITX - Setup

18.5.3 Table of Contents

Network-Attached Storage (NAS) is a file-level data storage connected to a computer net- work, typically via Ethernet transporting TCP/IP and/or UDP communication. Missing Link Electronics (MLE) Zynq SATA Storage Extension (Zynq SSE) is a fully integrated and pre-validated system stack comprising 3rd-party SATA Host Controller and DMA IP cores from ASICS World Services, a storage micro-architecture from MLE, Xilinx PetaLinux, and an Open Source SATA Host Controller Linux kernel driver, also from MLE. Zynq SSE utilizes the Xilinx GTX Multi Gigabit Transceivers to deliver SATA I (1.5 Gbps), SATA II (3.0 Gbps), or SATA III (6 Gbps) connectivity. When combined with Zynq's networking capabilities this effectively leads to a demonstration of Networked Storage, or NAS. This Technical Tip shows how to setup the Zynq SSE to demonstrate NAS functionality. After going through the steps described herein, you will have a working Linux System running on the Zynq with an attached SATA HDD or SSD, making files stored on the attached disk available to other networked clients.
Team MLE has spent significant efforts to try and test all aspects of Zynq SSE. However, if you feel that you encounter something not right, or if you do have any questions, please do not hesitate to contact us. The best way to contact MLE is to fill out the Contact Request Form at
http://MLEcorp.com/ZynqSSE

18.5.3.1 Block Diagram

The block diagram shown below gives an overview over the Zynq SSE reference design: Within the Zynq Programmable Logic (PL) the MLE storage micro-architecture instantiates the DMA and the SATA Host Controller IP blocks. The storage micro-architecture itself interfaces with the Zynq Processing System (PS) via the high-performance AXI HP0 slave port. The ARM A9 in the PS runs Xilinx PetaLinux and the SATA Linux kernel driver.



18.5.3.2 Implementation

Implementation Details	
Design Type	PS + PL
SW Type	Linux (Petalinux)
CPUs	2 CPUs 700 MHz
PS Features	DDR, USB, UART, ETHERNET
PL Cores	ASICS.WS SATA IP
Boards/Tools	Avnet Mini ITX Z045
Xilinx Tools Version	Vivado 2014.1, PETALINUX 2013-2
Other Details	SATA SSD(including Cable and Power Supply), SD-Card

Address Map			
	Base Address	Size	Interface
SATA IP	0x41000000	4K	S AXI
DMA IP	0x41010000	4K	S AXI, M AXI

Files Provided	
SD Card Image NAS	All Files needed to run the ZYNQ SSE

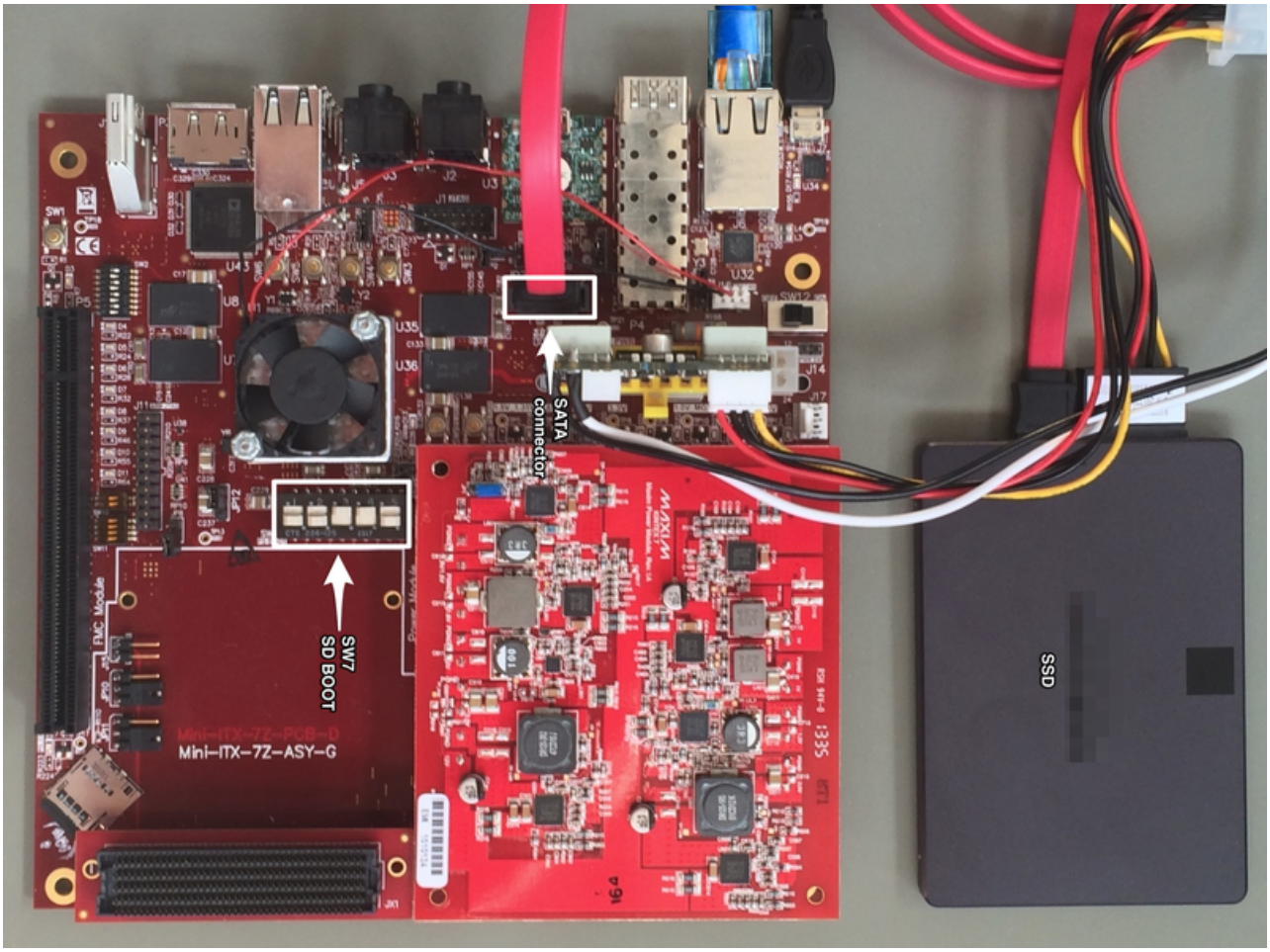
For the available Project to built your ZYNQ SSE got to http://mlecorp.com/zynqsse

18.5.3.3 Step by Step Instructions

18.5.3.3.1 Hardware needed:

• Avnet Mini ITX Z045 Board (including Power supply) Link
- Micro USB Cable for USB Console
- Supported SSD (for Example Samsung 840)
- Power Supply for SSD
- Micro SD Card (2 GB or bigger)
- Ethernet Cable to connect Host PC and Avnet ITX board

• PC for UART console



18.5.3.3.2 Software Assembly:

1. Format the SD Card using FAT32 File system
2. Put image.ub and BOOT.bin into the root directory of the SD Card

Hardware Assembly:

1. Insert the SD Card into the Avnet Mini ITX board's SD Slot
2. Connect the micro USB cable to the UART port o

f the Avnet Mini ITX board and the USB Port of your PC

1. Connect the Ethernet port of the host pc to the ethernet port of the Avnet MINI ITX
2. Connect the SSD to the SATA connector on the Avnet Mini ITX board using the SATA Cable.
3. Connect the SSD to power
4. Switch the Avnet Mini ITX board's SW7 to SD Boot mode (as shown in picture)
5. Connect the Avnet Mini ITX board to the Power Supply

The system should now look like in the supplied image

18.5.3.3.3 Startup:

18.5.3.3.3.1 Connect to the Avnet Board using RS232

1. Switch on the Avnet Mini ITX board
2. On the PC open a Serial Terminal on the new serial port using the settings 115200 Baud 8N1
3. Observe the Linux System booting
4. After some time you should see a screen similar to the screenshot in the Expected Results.
5. Login using

   Login:root
   Password: root

18.5.3.3.3.2 Connect to the Avnet Board using FTP

1. configure the ethernet port on the host PC to have an IP address of 10.89.231.1 and a subnet mask of 255.255.255.0
   It is crucial to configure the Host PC that way so that it can connect to the ZYNQ board
2. The ZYNQ-board has an pre-supplied IP of 10.89.231.200
3. now connect to the ZYNQ-board which has the IP of 10.89.231.200 using an FTP program and start transferring data to it

```
$ ftp 10.89.231.200
Connected to 10.89.231.200.
220 Operation successful
Name (10.89.231.200:christian): root
230 Operation successful
Remote system type is UNIX.
Using binary mode to transfer files.
ftp>
```

18.5.3.4 LED Description

Status LEDs can be found next to the PCIe connector. The associated meanings can be seen in the following table and in the Image below.

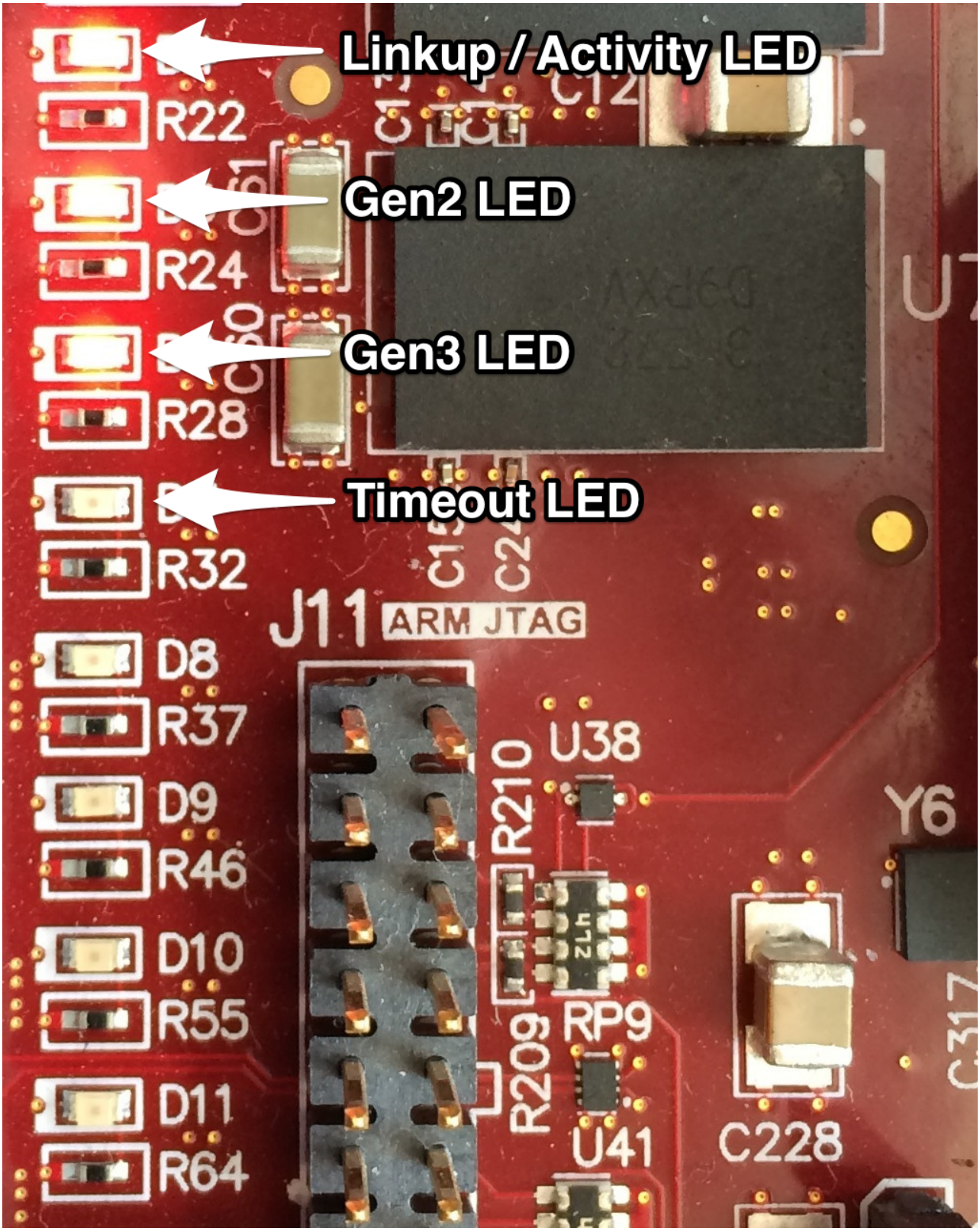


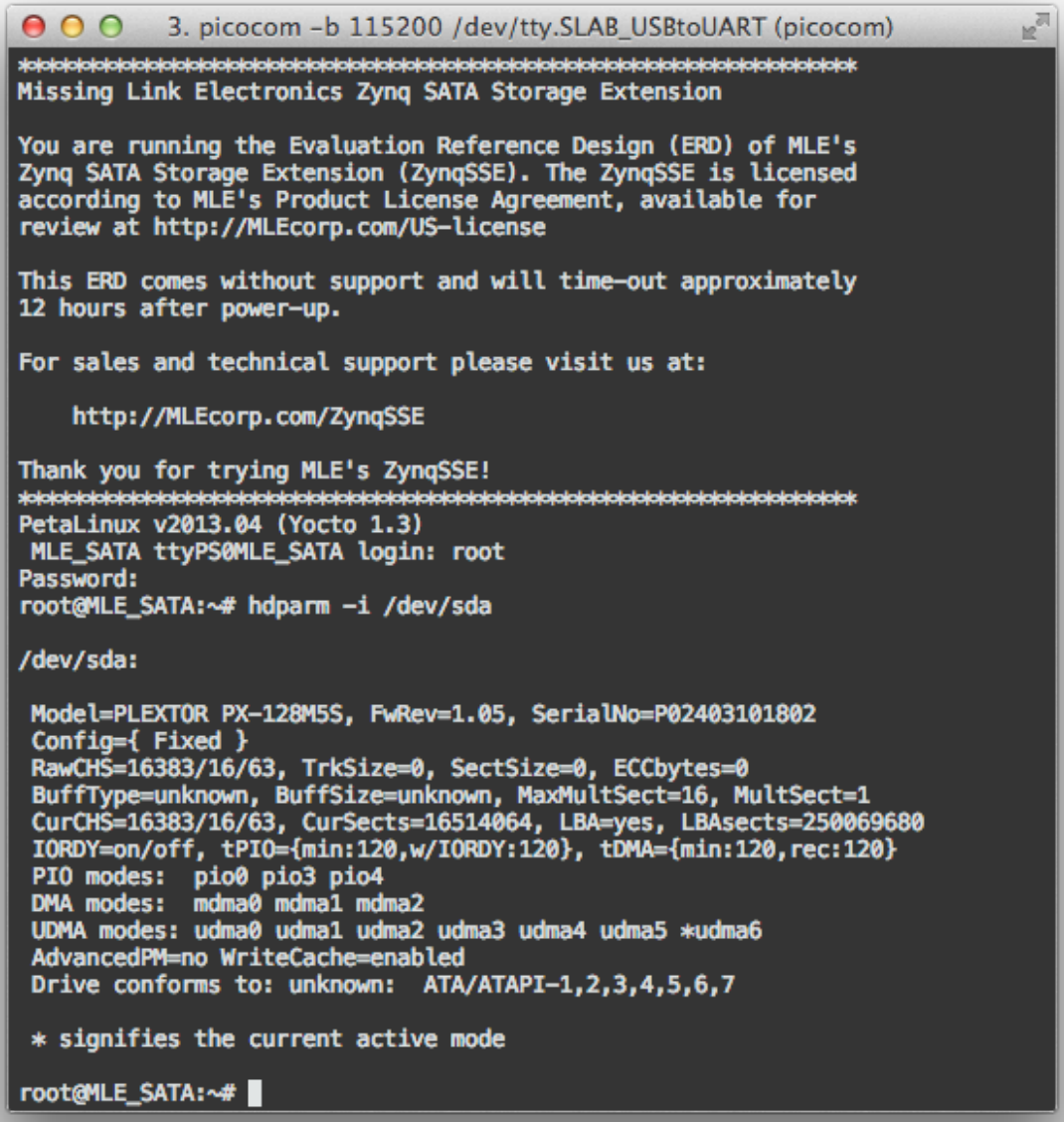
Figure 1: Location of the User LEDs on the Avnet Mini ITX Board

18.5.3.5 Expected Results

As a result you should be having a running Linux system on the Zynq board. The UART Console output should be similar to the screenshot below.

Also you will have file-system access to the attached SSD using FTP, and by this evaluate and test the Zynq Sata Storage extension.
The Evaluation Reference Design (ERD) of the Zynq SSE comprises a hardware license management which allows to run full SATA functionality for up to 12 hours after power-up. After approximately 12 hours the evaluation expires, which is indicated by illuminating the LED 'timeout'. You will also notice that the Linux kernel driver informs you of having lost the SATA link to the SSD/HDD.

D7	Timeout LED, indicates the Timeout of the IP core.
D6	Gen3 Link, indicates 6 GBit/s connection
D5	Gen2 Link, indicates 3 GBit/s connection
D4	Linkup and Activity, Led will light up on Linkup and will go dark during data transfers



18.6 Zynq-7000 AP SoC SATA part 4 – Ready to Run NAS Design Example Benchmarking

18.6.1 Zynq-7000 AP SoC SATA part 4 – Ready to Run NAS Design Example Benchmarking

18.6.2 Zynq SSE - NAS Benchmarks for the Avnet Mini-ITX

18.6.3 Table of Contents

For the evaluation of Zynq SSE Missing Link Electronics (MLE) currently supports the Avnet Zynq Mini-ITX board.
Network-Attached Storage (NAS) is a file-level data storage connected to a computer net- work, typically via Eth
ernet transporting TCP/IP and/or UDP communication.
MLE's Zynq SATA Storage Extension (Zynq SSE) is a fully integrated and pre-validated system stack comprising 3rd-party SATA Host Controller and DMA IP cores from ASICS World Services, a storage micro-architecture from MLE, Xilinx PetaLinux, and an Open Source SATA Host Controller Linux kernel driver, also from MLE. Zynq SSE utilizes the Xilinx GTX Multi Gigabit Transceivers to deliver SATA I (1.5 Gbps), SATA II (3.0 Gbps), or SATA III (6 Gbps) connectivity. When combined with Zynq's networking capabilities this effectively leads to a demonstration of Networked Storage, or NAS. This Technical Tip shows how to setup the Zynq SSE to demonstrate NAS functionality. After going through the steps described herein, you will have a working Linux System run- ning on the Zynq with an attached SATA HDD or SSD, making files stored on the attached disk available to other networked clients.
Team MLE has spent significant efforts to try and test all aspects of Zynq SSE. However, if you feel that you encounter something not right, or if you do have any questions, please do not hesitate to contact us. The best way to contact MLE is to fill out the Contact Request Form at
http://MLEcorp.com/ZynqSSE

18.6.3.1 Implementation

Implementation Details	
Design Type	PS + PL

SW Type	Linux (Petalinux)
CPUs	2 CPUs 700 MHz
PS Features	DDR, USB, UART, ETHERNET
PL Cores	ASICS.WS SATA IP
Boards/Tools	Avnet Mini ITX Z045
Xilinx Tools Version	Vivado 2014.1, PETALINUX 2013-2
Other Details	SATA SSD(including Cable and Power Supply), SD-Card

Address Map			
	Base Address	Size	Interface
SATA IP	0x41000000	4K	S AXI
DMA IP	0x41010000	4K	S AXI, M AXI

Files Provided	
BOOT.bin	Compilation of Bitstream, FSBL and U-Boot
Image.ub	Linux Ramdisk Image

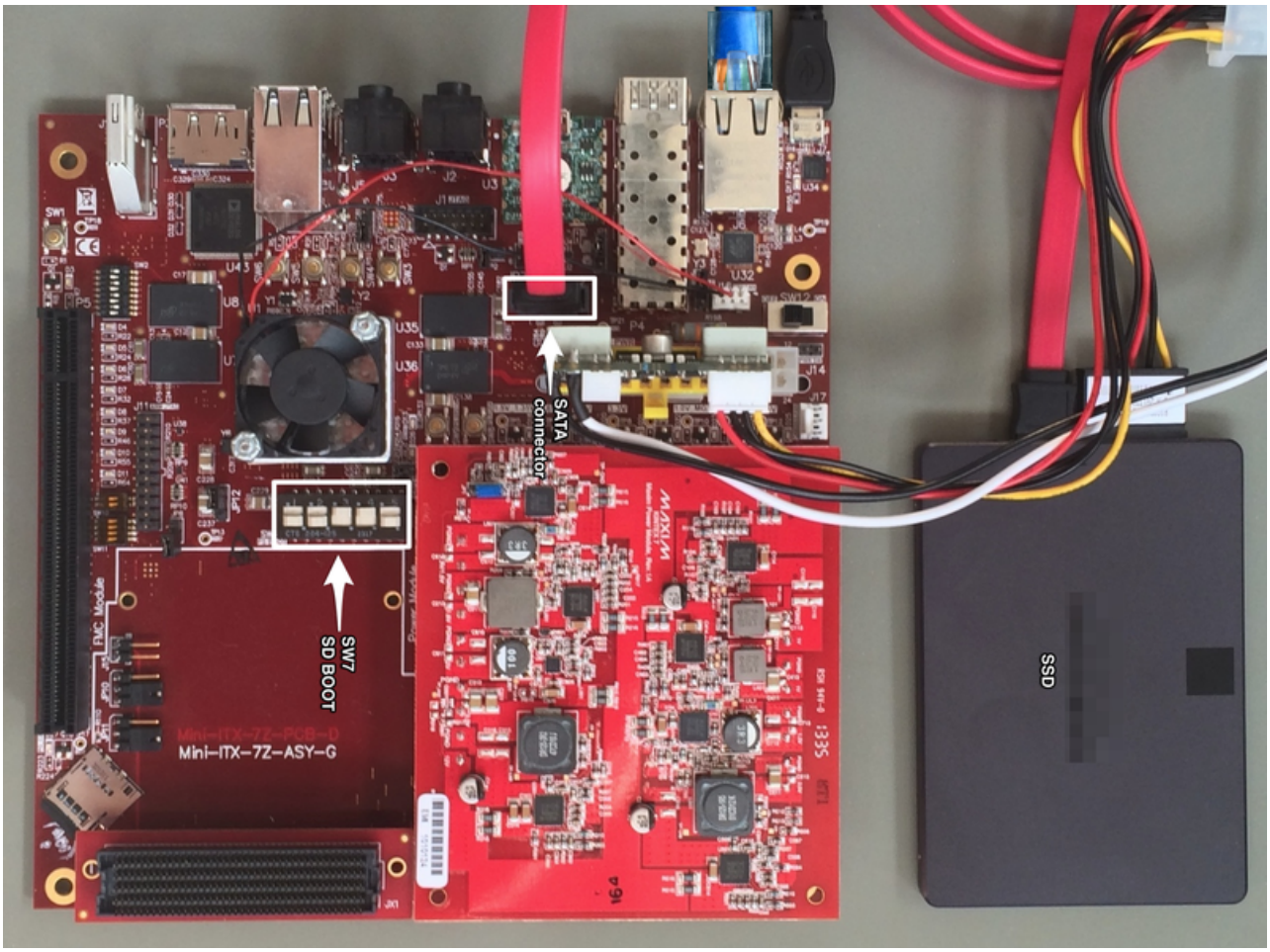
18.6.3.2 Step by Step Instructions

18.6.3.2.1 Hardware needed:

- Avnet Mini ITX Z045 Board (including Power supply) Link

- Micro USB Cable for USB Console
- Supported SSD (for Example Samsung 840)
- Power Supply for SSD
- Micro SD Card (2 GB or bigger)
- Ethernet Cable to connect Host PC and Avnet ITX board
- PC for UART console
  Software Assembly:
  1. Format the SD Card using FAT32 File system
  2. Put image.ub and BOOT.bin into the root directory of the SD Card

18.6.3.2.2 Hardware Assembly:



1. Insert the SD Card into the Avnet Mini ITX board's SD Slot
2. Connect the micro USB cable to the UART port of the Avnet Mini ITX board and the USB Port of your PC
3. Connect the Ethernet port of the host pc to the ethernet port of the Avnet MINI ITX
4. Connect the SSD to the SATA connector on the Avnet Mini ITX board using the SATA Cable.

5. Connect the SSD to power
6. Switch the Avnet Mini ITX board's SW7 to SD Boot mode (as shown in picture)
7. Connect the Avnet Mini ITX board to the Power Supply

The system should now look like in the supplied image
Startup:

1. Switch on the Avnet Mini ITX board
2. On the PC open a Serial Terminal on the new serial port using the settings 115200 Baud 8N1

3. Observe the Linux System booting

1. After some time you should see a screen similar to the screenshot in the Expected Results.
2. Login using

   • Login:root
   • Password: root
3. configure the ethernet port on the host PC to have an IP address of 10.89.231.1 and a subnet mask of 255.255.255.0
4. The ZYNQ-board has an pre-supplied IP of 10.89.231.200
5. now connect to the ZYNQ-board using an FTP program and start transferring data to it

LED Description
Status LEDs can be found next to the PCIe connector. The associated meanings can be seen in the following table and in the Image below.

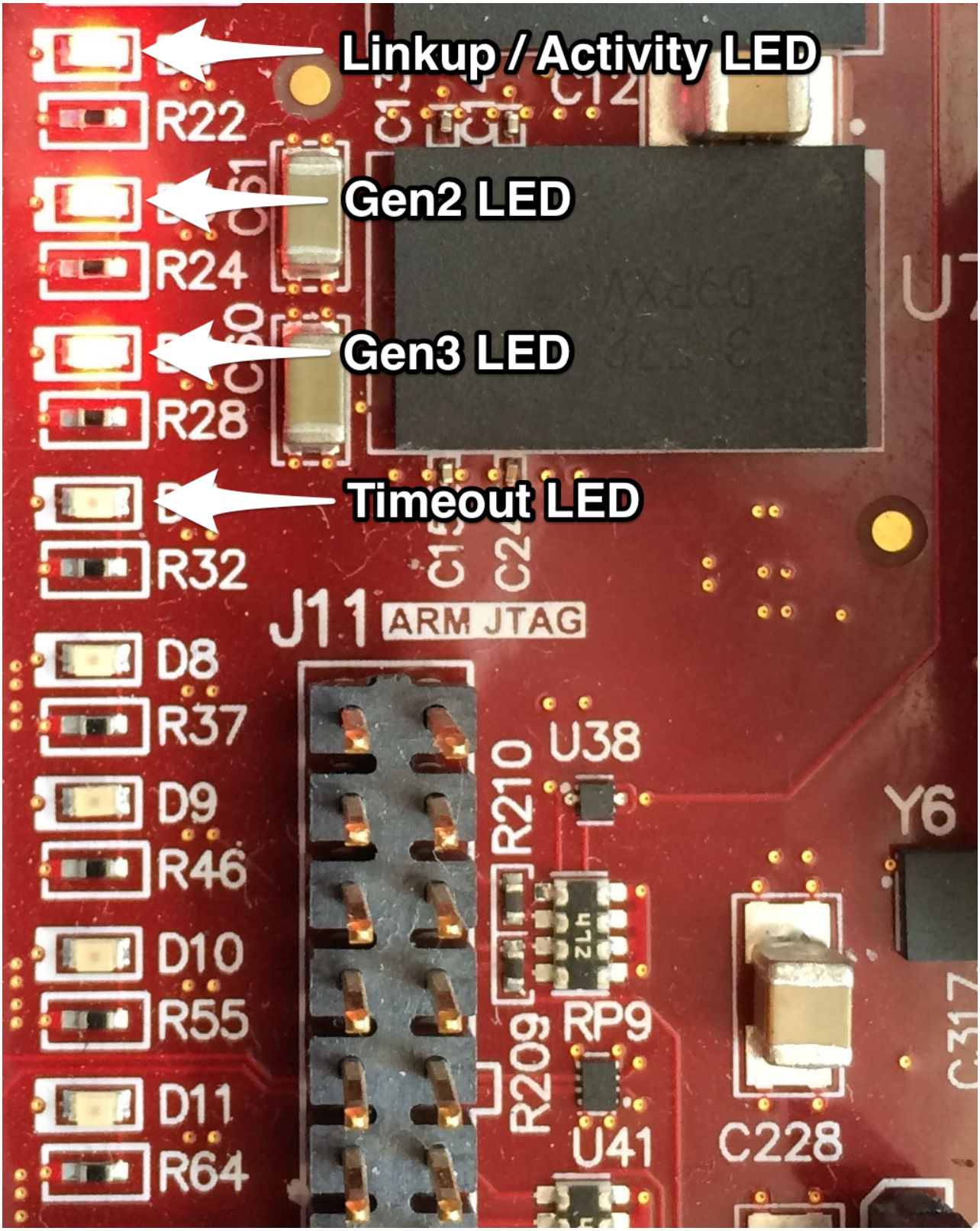


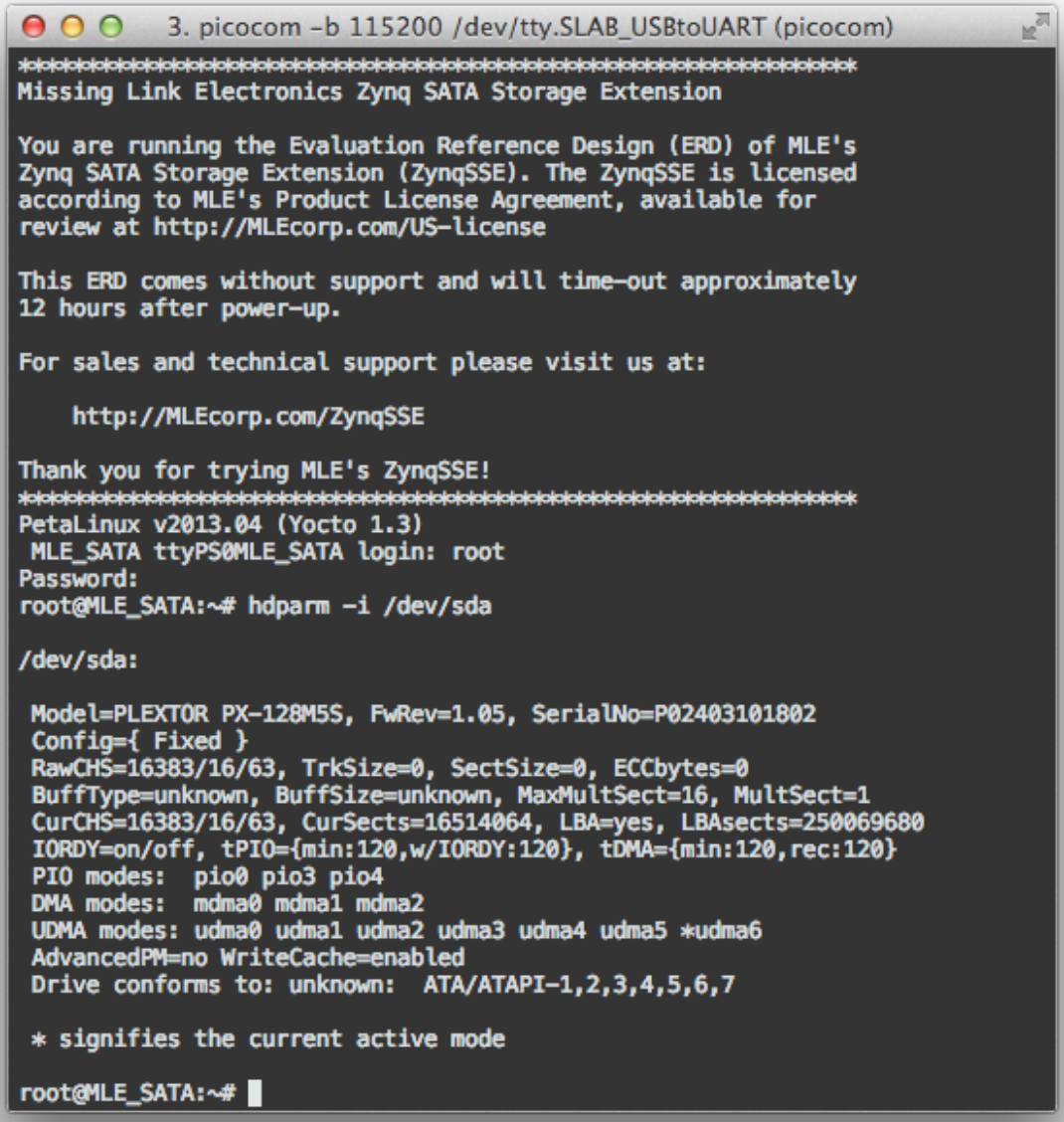
Figure 1: Location of the User LEDs on the Avnet Mini ITXBoard

18.6.3.3 Expected Results

As a result you should be having a running Linux system on the Zynq board. The UART Console output should be similar to the screenshot below.

Also you will have file-system access to the attached SSD using FTP, and by this evaluate and test the Zynq Sata Storage extension.
The Evaluation Reference Design (ERD) of the Zynq SSE comprises a hardware license management which allows to run full SATA functionality for up to 12 hours after power-up. After approximately 12 hours the evaluation expires, which is indicated by illuminating the LED 'timeout'. You will also notice that the Linux kernel driver informs you of having lost the SATA link to the SSD/HDD.

D7	Timeout LED, indicates the Timeout of the IP core.
D6	Gen3 Link, indicates 6 GBit/s connection
D5	Gen2 Link, indicates 3 GBit/s connection
D4	Linkup and Activity, Led will light up on Linkup and will go dark during data transfers



18.6.3.3.1 Performance

The Performance has been measured with the standard FTP program which is coming with every linux distribution and is in the region of 30-40 MB/s in both directions.
To measure it, create a 1GB test file using:

```
dd if=/dev/zero of=./testfile.dd count=1024 bs=1024k
```

Now transfer the file to the ZYNQ using:

```
ftp 10.89.231.200
```

Push the testfile to the linux side:

```
put testfile.dd
```

Get the testfile:

```
get testfile.dd
```

18.6.3.3.2 Example

This is an Example of the above.

```
$ ftp 10.89.231.200                                                    [17:57:4
3]
Connected to 10.89.231.200.
220 Operation successful
Name (10.89.231.200:christian): root
230 Operation successful
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> put ./testfile.dd
local: ./testfile.dd remote: ./testfile.dd
229 EPSV ok (|||50663|)
150 Ok to send data
100% |*****************************************|  1024 MiB   36.42 MiB/s    00:00
 ETA
226 Operation successful
1073741824 bytes sent in 00:28 (36.37 MiB/s)
ftp> get testfile.dd
local: testfile.dd remote: testfile.dd
229 EPSV ok (|||40913|)
150 Opening BINARY connection for testfile.dd (1073741824 bytes)
100% |*****************************************|  1024 MiB   30.35 MiB/s    00:00
 ETA
226 Operation successful
1073741824 bytes received in 00:33 (30.35 MiB/s)
Can't parse time `19700001000244'.
ftp>
```

18.7 Zynq-7000 AP SoC SATA part 5 – Building the Design Example

18.7.1 Zynq SATA Storage Extension (Zynq SSE) Developers Guide

18.7.2 Table of Contents

This Zynq SSE Developers Guide has been written for system-level architects, FPGA developers, and embedded software engineers who design with and integrate Missing Link Electronics (MLE) Zynq SATA Storage Extension (Zynq SSE). This Developers Guide:

- Describes the items delivered as part of the Zynq SSE.
- Gives an architectural overview over the Zynq SSE.
- Shows the Xilinx Vivado FPGA design flow for Zynq SSE.
- Explains how to activate the MLE license keys for Zynq SSE.

MLE's Zynq SATA Storage Extension (Zynq SSE) is a fully integrated and pre-validated system stack comprising 3rd-party SATA Host Controller and DMA IP cores from ASICS World Services, a storage micro-architecture from MLE, Xilinx PetaLinux, and an Open Source SATA Host Controller Linux kernel driver, also from MLE. Zynq SSE utilizes the Xilinx GTX Multi Gigabit Transceivers to deliver SATA I (1.5 Gbps), SATA II (3.0 Gbps), or SATA III (6 Gbps) connectivity.
The Zynq SSE is delivered as a complete reference design for the Xilinx Zynq-7000 SoC (Zynq), and effectively extends Zynq with one single SATA host port for HDD and/or SSD storage connectivity.

To ease integration work, Zynq SSE provides a complete FPGA design project for Xilinx Vivado targeting the Avnet Zynq Mini-ITX-7045 Board (Avnet Mini-ITX).
This enables you to quickly bring up Zynq SSE on a known-good FPGA and hardware platform, and to migrate Zynq SSE to your particular target system. Please refer to the following MLE Technical Briefs for further information:

- TB20140428 Zynq SSE basic setup for Avnet Mini-ITX.
- TB20140429 Zynq SSE benchmarking for the Avnet Mini-ITX.

Team MLE has spent significant efforts to try and test all aspects of Zynq SSE. However, if you feel that you encounter something not right, or if you do have any questions, please do not hesitate to contact us. The best way to contact MLE is to fill out the Contact Request Form at:
http://MLEcorp.com/ZynqSSE

18.7.2.1 1 Delivered Items

Zynq SSE is delivered inside an installable archive for Windows, 32 bit Linux, and 64 bit Linux. Inside this archive you will find the following files.
To obtain this file, go to http://mlecorp.com/zynqsse and fill out the Contact Request Form.

Files Provided	
zynqsse.zip	Complete Vivado 2014.3 design project targeting the Avnet Mini-ITX.
BOOT.bin	Compilation of Bitstream, FSBL and U-Boot, all targeted for Avnet Mini-ITX.
Image.ub	Linux Ramdisk Image with MLE SATA extensions including the Zynq SSE Linux kernel modules.
sata_driver.zip	Archive with Zynq SSE Linux kernel modules, as running in PetaLinux system, and as they need to be installed in your target system.
U-Boot.elf	U-Boot Bootloader to get Petalinux running.
fsbl.elf	First Stage Boot Loader, with SATA ICAP patch (refer to Section 3.4).

Please note that all of these files are licensed under the terms and conditions of MLE's Product License Agreement, a copy of which can be found at:
http://MLEcorp.com/US-license
For additional information about the Avnet Mini-ITX board, please refer to the MLE Technical Brief TB20140428, or visit Avnet's product webpage at:
Avnet Mini ITX

18.7.2.2 2 Setup of Hardware Platforms for Zynq SSE

Ready-to-run Evaluation Reference Designs (ERD) for Zynq SSE are available the Avnet Zynq Mini-ITX-7Z045 Board (Avnet Mini-ITX), shown in Figure 3. For easy diagnostics Zynq SSE uses on-board LEDs.

18.7.2.2.1 2.1 LED Description

Status LEDs can be found next to the PCIe connector. The associated meanings can be seen in the following table:

D7	Timeout LED, indicates that the run-time license expired.

D6	Gen3 Link, indicates 6 GBit/s connection.
D5	Gen2 Link, indicates 3 GBit/s connection.
D4	Linkup and Activity, LED will light up on Linkup and will go dark during data transfers.

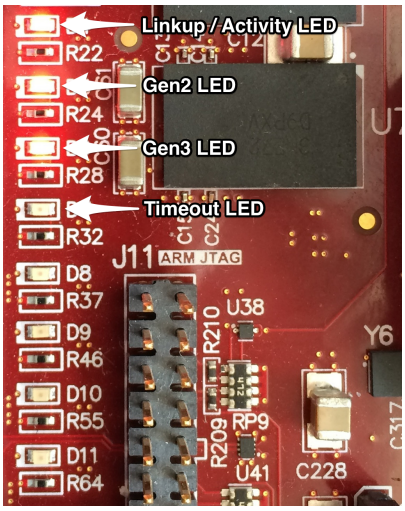


Figure 1: Location of the User LEDs on the Avnet Mini ITX Board

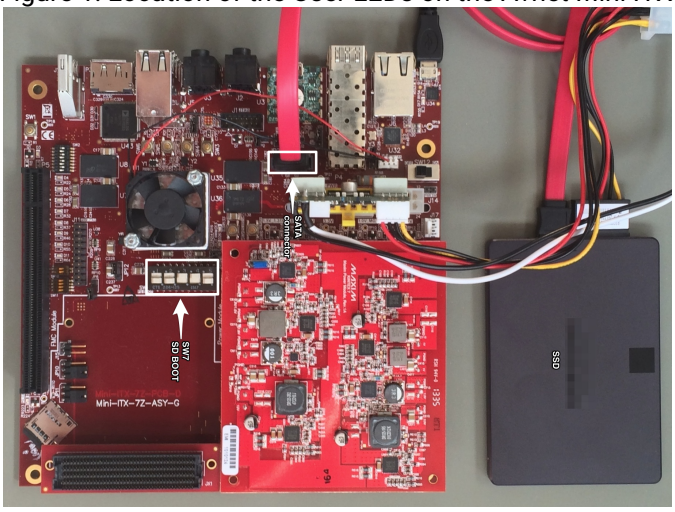


Figure 3: Zynq SSE Hardware Setup for Avnet Mini-ITX

18.7.2.3 3 Architecture Details

The system architecture diagram in Figure 5 gives an overview of the Zynq SSE reference design: Within the Zynq Programmable Logic (PL) the MLE Storage Micro-Architecture instantiates the DMA and the SATA Host Controller IP blocks. The storage micro-architecture itself interfaces with the Zynq Processing System (PS) via the AXI Interconnect. The ARM A9 in the PS runs Xilinx PetaLinux and the SATA Linux kernel drivers. Please note that this figure is not to scale and that there is plenty of programmable logic resources left for

you in the Zynq FPGA device to implement your portions of your target design! The block diagram in Figure (for the Avnet Mini-ITX) provides more details of the Storage Micro-Architecture. To ease integration of Zynq SSE into your target system, the Storage Micro-Architecture assembles all 3rd-party IP cores, instantiates and parameterizes them in accordance to the original IP core vendor's design manual, wires them up properly, and implements all connectivity to internal signals as well as to external FPGA IO. All of this is encapsulated into one top-level module mle_storage_uarch. Further down, we provide details on how system software parameters and memory addresses have been set.

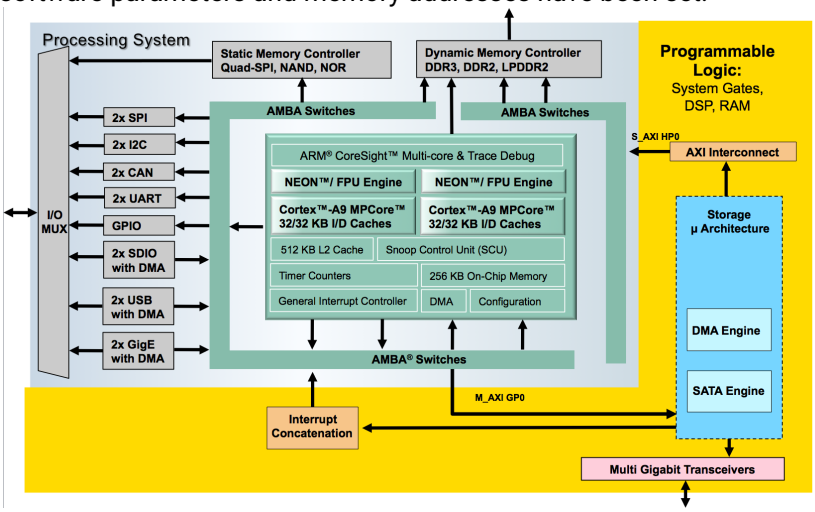


Figure 5: Zynq SSE System Architecture

18.7.2.3.1 3.1 Pin Assignments for the Avnet Mini-ITX

As external IO, the Storage Micro-Architecture connects to the following FPGA pins to establish SATA connectivity and to provide user LEDs for diagnostic purposes. For your convenience Figure 1 identifies LEDs on the Avnet Mini-ITX-7045 board (for Mini-ITX-7045 hardware revision 1.1, and newer).

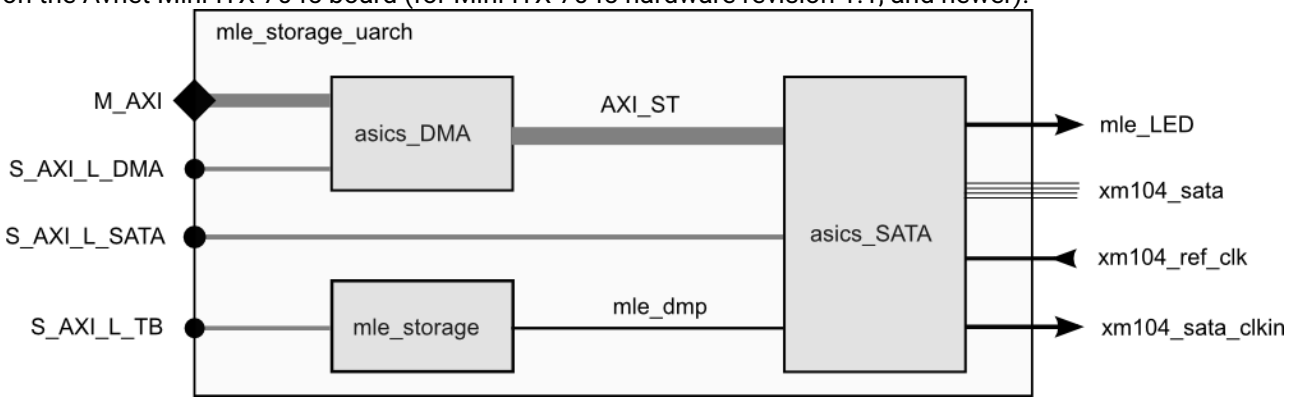


Figure 6: MLE Storage Micro-Architecture Details for Avnet Mini-ITX

Name	Direction	Description
sata_ref_clk_p_i sata_ref_clk_n_i	in in	Incoming Reference Clock

sata_rxn_i sata_rxp_i	in in	SATA Rx Pair differential
sata_txn_o sata_txp_o	out out	SATA Tx Pair differential
sata_clkin_p_o sata_clkin_n_o	out out	SATA Clock differential
sata_led_linkup_o	out	Illuminates when SATA link is established; goes dark when data is read or written from the SATA disk
sata_led_gen2_o	out	Illuminates when SATA 3 Gbps speed is established.
sata_led_gen3_o	out	Illuminates when SATA 6 Gbps speed is established.
sata_led_timeout_o	out	Illuminates when Zynq SSE license has expired.

ATTENTION!

Please keep in mind that this design project targets the Avnet Mini-ITX-7Z045 and that you need to change those pin assignments to match your target hardware setup!

18.7.2.3.2 3.2 AXI Memory Interfaces

According to Figure 6, the Storage Micro-Architecture connects to the Zynq PS via four AXI interconnects. The interfaces for the Storage Micro-Architecture have been pre-configured with the following memory address space:

Name	Offset Address	High Address	Description
M_AXI	0x00000000	0x3FFFFFFF	AXI master for direct memory access into the DDR3 RAM attached to the Zynq PS
S_AXI_L_SATA	0x41000000	0x4100FFFF	AXI-lite interface to control the SATA host controller IP core
S_AXI_L_DMA	0x41010000	0x4101FFFF	AXI-lite interface to control the DMA controller IP core
S_AXI_L_TB	0x41020000	0x4102FFFF	AXI-lite interface to control the micro-architecture and for production license key activation

ATTENTION!

The entire memory address space between 0x41000000 and 0x4102FFFF is fixed and reserved for Zynq SSE. You must ensure that you do not conflict with this memory address space, and, in particular, that you do not write into that memory address space. You must not change those address values or Zynq SSE stops functioning as specified!

18.7.2.3.3 3.3 IEEE 1275 Device Tree

The memory address setting are also reflected in the device tree. This device tree is stored inside the image.ub file. The following snippet shows the Zynq SSE specific section:

```
satah_0_hc: satahc@41000000
    {compatible = "aws,satah4-1.00.a";
     reg = <0x41000000 0x800>;
     interrupt-parent = <&&ps7_scugic_0>;
     interrupts = <0 58 1>;
     dma-engine = <&&satah_0_dma 0>; };
satah_0_dma: dmaeng@41010000
     {compatible = "aws,axi-dma-1.00.a";
      reg = <0x41010000 0x800>;
      interrupt-parent = <&&ps7_scugic_0>;
      interrupts = <0 59 4>; };
mle_ipwatch: ipwatch@41020000
     {compatible = "mle,axi-ip-watch-1.00";
      reg = <0x41020000 0x1000>; };
```

18.7.2.3.4 3.4 Bootloader Settings

To ensure that Zynq SSE runs on a supported device, Zynq SSE will check for your FPGA's device ID. To do this the ICAP interface is used. However, the ICAP interface will only work if the PCAP interface has been deactivated. This has been done by MLE by running a modified First Stage Bootloader (FSBL), which is a transparent change because the PCAP interface will be activated again by the U-Boot, later.
We recommend that you to use the FSBL from MLE. If you need to use your own customized FSBL (maybe, because you need to initialize your own hardware blocks) please make sure to adjust your FSBL by editing the function FsblHookBeforeHandoff in the file SDK_Export/fsbl/src/fsbl_hooks.c as shown below.

```
u32 FsblHookBeforeHandoff(void)
{
  u32 Status;

  u32 xdcfg_ctrl;
  int status;
  XDcfg xdcfg_instance;
  XDcfg_Config *ConfigPtr;

  Status = XST_SUCCESS;
  ConfigPtr = XDcfg_LookupConfig(XPAR_PS7_DEV_CFG_0_DEVICE_ID);
  status = XDcfg_CfgInitialize(&&xdcfg_instance, ConfigPtr, ConfigPtr->BaseAddr);
  if (status != XST_SUCCESS) {
    return XST_FAILURE;
  }
  xdcfg_ctrl = XDcfg_ReadReg(xdcfg_instance.Config.BaseAddr, XDCFG_CTRL_OFFSET);
  xdcfg_ctrl &&= ~XDCFG_CTRL_PCAP_PR_MASK;
  XDcfg_WriteReg(xdcfg_instance.Config.BaseAddr, XDCFG_CTRL_OFFSET, xdcfg_ctrl);

  /*
   * User logic to be added here.
   * Errors to be stored in the status variable and returned
```

```
    */
   fsbl_printf(DEBUG_INFO,"In FsblHookBeforeHandoff function \r\n");

   return (Status);
}
```

ATTENTION!

Please make sure that this code is executing on startup of the system. If not, Zynq SSE will not work!

18.7.2.4 4 Zynq SSE Design Flow

Typically, integrating a 3rd-party IP core requires reading a lot of documentation to properly parameterize and instantiate the IP core into your target design. To reduce your design effort when integrating Zynq SSE, MLE provides you with a complete reference design project. This reference design runs on the Avnet Mini-ITX-7Z045 which makes system-level verification and software development easy. To port this reference design to your target system, please follow the three simple rules of “Things You Must Do, Things You Can Do, and Things You Must Not Do”.

18.7.2.4.1 4.1 Things You Must Do

Xilinx Vivado offers an innovative new design flow via the so-called “Run”. To migrate the Zynq SSE Reference Design to your target system, please create a new “Run”, where you will have to specify your particular part from the Zynq-7000 family, including picking the proper speed-grade, device and constraints.
You must also adjust the pin assignments in the Design Constraints.
You must further supply a SATA-capable differential reference clock with low jitter at 150 MHz to the following pins:
sata_ref_clk_n_i sata_ref_clk_p_i
On the software side you must install the MLE Linux kernel modules into your target system. You can copy them from your Avnet Mini-ITX-7Z045 system, or extract them from the separate archive mlelkm.tar which we provided for your convenience.
To compile Zynq SSE under Xilinx Vivado you must have a compile-time license key. This is described in Section 5.1.

18.7.2.4.2 4.2 Things You Can Do

You can add pretty much any additional functionality to the PS and/or PL. Zynq SSE does not require exclusive access to the AXI ports, so you can share those with the AXI Interconnect for your user logic. Obviously, you should follow good design practice when sharing resources to not create bottlenecks which may impact your SATA performance.
You can also leave out the four user LEDs (sata_led) as they are provided solely for convenient diagnostic purposes. It is okay to leave those LED outputs open in your target design.

18.7.2.4.3 4.3 Things You Must Not Do

Most importantly, you must not change the setup of the memory address space of the MLE Storage Micro-Architecture. This is so important, that we repeat this here, again!
And, you must not pick any target FPGA device other than those devices with multi-gigabit transceivers from the Xilinx Zynq-7000 SoC family as Zynq SSE will not run without it. This has legal and technical reasons because MLE's Storage Micro-Architecture for Zynq SSE is designed and licensed for Zynq only.

18.7.2.4.4 4.4 Running Zynq SSE in Xilinx Vivado

Zynq SSE has been tested (under Windows and Linux) for:
Xilinx 2014.3 Vivado
Xilinx 2014.3 SDK
First of all extract the archive zynq_sse.zip. This folder does not yet contain any Vivado project sources. They have to be created first using the build script build.sh inside the top folder zynq_sse_avnet_7045. This build script has to be executed inside a Xilinx shell. Please make sure you have the proper Xilinx Vivado version installed. And make sure that you set the executable flag for the build script by executing chmod +x. If this was successful you will receive an output similar to the following:

```
Wrote  : </home/fass/simon/work/test_sata/zc706_satah4_axi_ad/project/zc706_sata_2014.1.srcs/sources_1/bd/s
ata_system_avnet_z100/sata_system_avnet_z100.bd>
## close_bd_design $bd_name_avnet_z100
## import_files -norecurse  $hdl_dir/${bd_name_avnet_z100}_wrapper.v
## update_compile_order -fileset sources_1
## create_fileset -constrset constrs_avnet_z100
## add_files -fileset constrs_avnet_z100 -norecurse $constrs_dir/avnet_z100_sata.xdc
## add_files -fileset constrs_avnet_z100 -norecurse $constrs_dir/avnet_z100_sata_impl.xdc
## set_property used_in_synthesis false [get_files $constrs_dir/avnet_z100_sata_impl.xdc]
## create_run synth_avnet_z100 -part $part_avnet_z100 -flow {Vivado Synthesis 2013} -constrset constrs_avne
t_z100
Run is defaulting to srcset: sources_1
## create_run impl_avnet_z100 -part $part_avnet_z100 -parent_run synth_avnet_z100 -flow {Vivado Implementat
ion 2013} -constrset constrs_avnet_z100
Run is defaulting to parent run srcset: sources_1
## set_property strategy Performance_Explore [get_runs impl_avnet_z100]
## current_run [get_runs synth_avnet_z100]
## delete_fileset constrs_1
## exit
INFO: [Common 17-206] Exiting Vivado at Tue Jun 17 16:38:55 2014...
17.91user 0.83system 0:24.45elapsed 76%CPU (0avgtext+0avgdata 1462912maxresident)k
0inputs+58888outputs (0major+102192minor)pagefaults 0swaps

real    0m24.455s
user    0m17.913s
sys     0m0.836s
simon@topf:~/work/test_sata/zc706_satah4_axi_ad$
```

Figure 8: Build of the Vivado Project

After executing the build script, the Vivado project and the block design will have been generated. You will find a new folder project in your zynq_sse_avnet folder. Open this folder and verify that it contains the folders and Vivado project files displayed below.

```
project/zynq_sse_sata_2014.3.cache/
project/zynq_sse_sata_2014.3.srcs/
project/zynq_sse_sata_2014.3.xpr
```

Now open your Vivado tool with the project file zynq_sse_sata_2014.3.xpr. You will see the following Vivado Design, shown in Figure 9:
Click on Open Block Design and you will see the Block Design, shown in Figure 10:

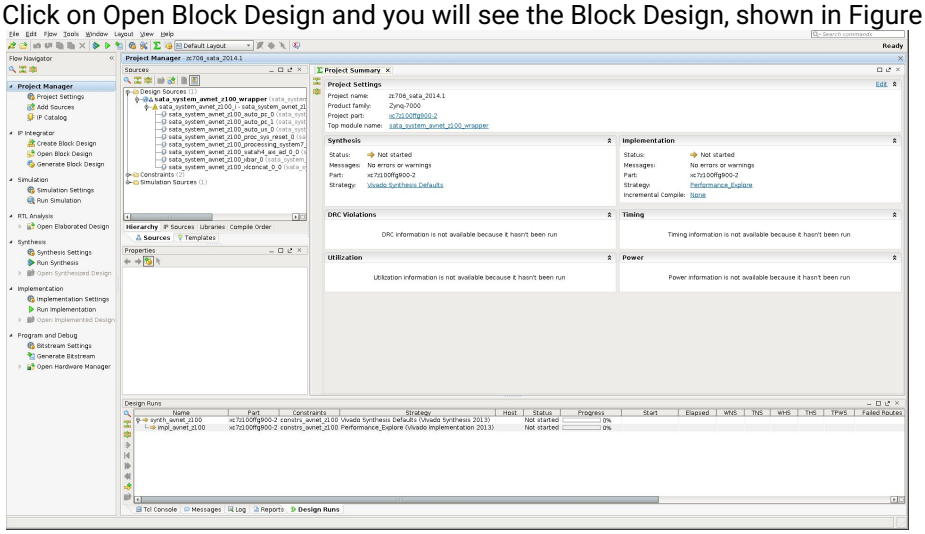


Figure 9: Vivado Startup Screen

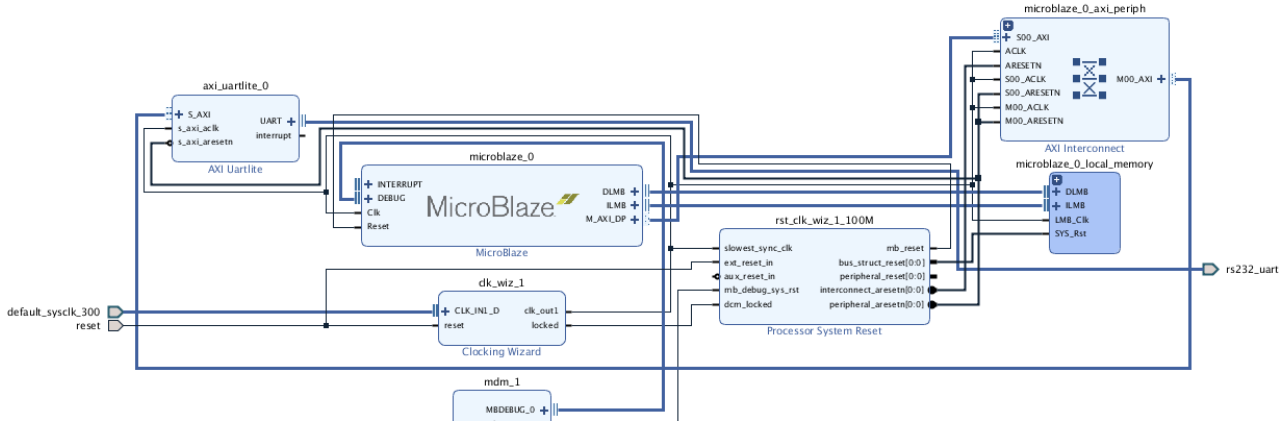


Figure 10: Block Design

This is where you can add your own design blocks to Zynq SSE. You can add additional AXI-ports and AXI-interconnects for your additional hardware blocks. By clicking on the Address Editor you will see the offsets of the different AXI-Interfaces. Please make sure that they match the values from Section 3.3.
At this point you will be able to generate your own bitfile. This is required to build your BOOT.BIN. To do this click on the Generate Bitstream Button or via the menu: Flow -> Generate Bitstream
Once this successfully completes, you will have a bitfile in the folder: zynq_sse_sata_2014.3.runs/impl_avnet. Using this bitfile you can generate the file BOOT.BIN.
To do this open a Petalinux version 2013.04 shell. You will need the U-Boot.elf and the fsbl.elf file. We

recommend that you to use the pre-built files and delivered by MLE (in the pre-compiled folder), however, it is also okay to use your own customized files. Just make sure you have made the necessary adjustments described in Section 3.4.
Once you have all the files needed you can create your BOOT.BIN file with the following command:

```
petalinux-gen-zynq-boot -b <path to fsbl.elf> -f <path to bitfile> -u <path to
u-boot.elf> -o ./BOOT --force
```

Now copy your BOOT.BIN and your image.ub file onto a FAT32 formatted SD-Card. After you have followed the instructions for setting up your target hardware, you will have a fully functional Zynq SSE!

Document History

Date	Author	Revision
2014-03-25	CG	Initial draft document
2014-05-19	SL	Added licensing concepts
2014-06-17	SL	Updated Vivado design flow
2014-06-30	ES	Update for ZynqSSE Release 1.2
2014-12-5	AS	Update for Vivado 2014.3

19 Zynq-7000 Analog Data Acquistion using AXI_XADC

Table of Contents

19.1 1 Introduction

This page provides introduction and steps to build the various artifacts used in the Analog Data Acquisition application note (XAPP1183). It is demonstrated on ZC702 Evaluation kit which is based on a XC7Z020 CLG484-1 Zynq-7000 SoC device.

19.1.1 1.1 Detailed Description

The Zynq-7000 family is based on the Xilinx SoC architecture. These products integrate a feature-rich dual-core ARM® Cortex™-A9 based processing system and 28 nm Xilinx programmable logic (PL) in a single device. The ARM Cortex-A9 CPUs are the heart of the PS
and also include on-chip memory, external memory interfaces, and a rich set of peripheral connectivity interfaces. XADC is an integrated 12-bit, 17 channel, 1 Ms/s ADC. The Zynq-7000 SoC PS communicates with the XADC using an AXI interface when the XADC is instantiated in the PL. XADC is an embedded block available in all Zynq-7000 SoCs.The LogiCORE™ XADC wizard IP provides an AXI4-Lite compatible interface and an optional AXI4-Stream interface. The AXI4-Lite interface is used to configure the XADC, and the AXI4-Stream interface is used for data communication. The AXI4-Stream interface includes options for interfacing the XADC data interface to other signal processing IP. This application note demonstrates using the AXI4-Lite interface to control the XADC configuration parameters, and using the AXI4-Stream interface to capture

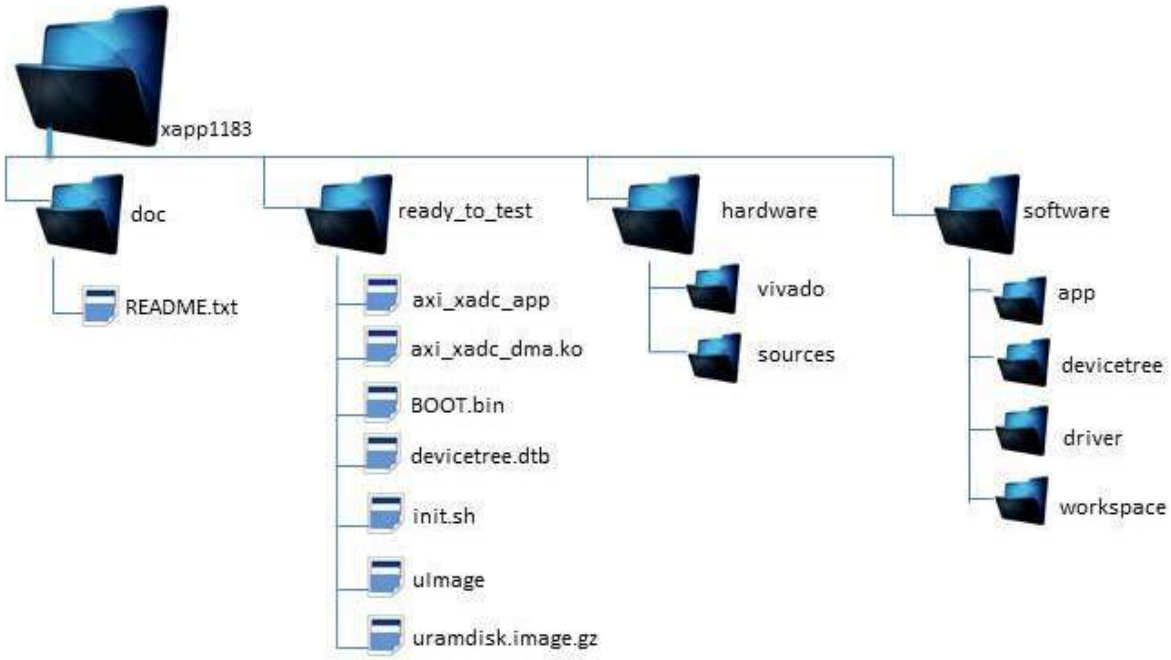
samples of the input analog data.This application note provides a hardware design in the PL that establishes the datapath between the XADC and the PS using the general-purpose (GP) port interface. An AXI DMA isused to interface to the XADC AXI4-Stream interface, and the DMA stores XADC samples in the PS DDR3. The Cortex-A9 processor is used to configure the XADC for user-specific configuration parameters. A LabView based GUI interface is provided to configure the XADC and display the collected samples.The samples are analyzed in the GUI to plot the linearity of the samples and various performance characteristics of the XADC,including signal-to-noise ratio (SNR), total harmonic distortion (THD), signal-to-noise and distortion (SINAD), and effective number of bits (ENOB). The LabView GUI interfaces to the XADC with the UART interface.
This highlights of this application note are:

1. How to quickly establish an interface between Zynq-7000 PS and XADC using AXI interface
2. Showcase optimized analog data acquisition using Zynq XADC.

19.2 2 Package Content

Application note package contains hardware/software sources along with ready to test binaries.
An archive with application note files can be downloaded here . (Requires signup in www.xilinx.com)

The following Figure shows the directory structure tree.



19.3 3 Prerequisites

- The ZC702 Evaluation Kit ships with the version 14.x Device-locked to the Zynq-7000 XC7Z020 CLG484-1 device and all required licenses to build the application note. For additional information, refer to UG798 ISE Design Suite 14: Installation and Licensing Guide.
- A Linux development PC with the ARM GNU tools installed. The ARM GNU tools are included with the Xilinx ISE Design Suite Embedded Edition or can be downloaded separately.
- A Linux development PC with the distributed version control system Git installed. For more information, refer to Using Git and to UG821: Xilinx Zynq-7000 EPP Software Developers Guide.

Tools for Software build
For building Linux kernel ,drivers and application:

- Xilinx XSDK 14.6/2013.2
- ARM cross compile tool
- mkimage
- corkscrew
- git

From here onward lets consider the unzipped package is available at $ZYNQ_AXI_XADC_HOME.
$ZYNQ_AXI_XADC_HOME/software/workspace can be used for development area.

19.4 4 Build Hardware components

19.5 Follow the below steps to build the Hardware design

1. Browse to $ZYNQ_AXI_XADC_HOME/hardware/vivado/scripts
2. Type the below command on Linux/Windows command line:

```
bash> vivado -source axi_xadc_gui.tcl
```

The above step will open Vivado GUI and the project.
3. Click on Generate Bitstream option which generates the bitstream
4. Go to File-> Export -> Export Hardware for SDK option and click OK

19.6 5 Build software components

19.6.1
5.1 Standard Zynq Software Components

- Includes creation of FSBL (First Stage boot loader) . FSBL used in the application note is based on zc702_hw_platform(predefined).
- Building of FPGA Hardware bitsream using Xilinx Vivado™ Design Suite
- Building Xilinx configured uboot (Second stage boot loader) for zc702 board. Check out tag "xilinx-v14.6" and follow standard uboot build process.
- Create BOOT.bin (zynq_fsbl.elf ,system.bit and u-boot.elf)

NOTE: Building standard software/hardware components are not covered in this application note and avoided for simplicity.
User may refer to detail steps mentioned in Zynq Base TRD 14.5 wiki which are identical to this application note requirements.

19.6.2 5.2 Build Xilinx kernel

Steps for building the Linux kernel.Set the CROSS_COMPILE environment variable and add it to your PATH.

```
bash> export CROSS_COMPILE=arm-xilinx-linux-gnueabi-
bash> export PATH=/path/to/cross/compiler/bin:$PATH
```

Linux kernel compilation internally uses mkimage command for creating uImage (Linux Kernel Image).
Hence path for mkimage command should also be added in PATH environment variable as shown below.
One can use the mkimage command that is built during u-boot building process (Section 4.1).
Clone the latest Zynq Linux kernel git repository from the Xilinx git server.

```
bash> cd $ZYNQ_AXI_XADC_HOME
bash> git clone git://github.com/Xilinx/linux-xlnx.git
```

Create a new branch named zynq_axi_xadc_dma based on the xilinx-v14.6

```
bash> cd $ZYNQ_AXI_XADC_HOME/linux-xlnx
bash> git checkout -b zynq_axi_xadc_dma  xilinx-v14.6
```

Configure the Linux kernel for zc702 default configuration

```
bash> make ARCH=arm xilinx_zynq_defconfig
```

Build the Linux kernel. The generated kernel image can be found at $ZYNQ_AXI_XADC_HOME/linux-xlnx/arch/arm/boot/uImage.

```
bash> make ARCH=arm uImage modules UIMAGE_LOADADDR=0x8000
```

19.6.3 5.3 Building the Linux Device Tree Blob

Device tree configures kernel loglevel to "1". It also adds Xilinx DMA engine, AXI XADC DMA nodes to default ZC702 device-tree.
Compile the device tree source provided as part of application note package.

The output of this step is a device tree blob and can be found at $ZYNQ_AXI_XADC_HOME/linux-xlnx/devicetree.dtb.

```
bash>  cd $ZYNQ_AXI_XADC_HOME/linux-xlnx
bash> ./scripts/dtc/dtc -I dts -O dtb -o devicetree.dtb $ZYNQ_AXI_XADC_HOME/software/devicetree/zynq-zc702-xadc-dma.dts
```

19.6.4 5.4 Build LInux AXI XADC DMA Driver

This driver configures AXI XADC and DMA IP. It provides a character driver interface to the user application for reading ADC samples .It is designed based on standard Linux DMA API framework.
Linux AXI DMA driver is build a loadable module, which can be easily modified as per user requirements.
For compiling the kernel Makefile **KDIR** environment variable needs to point to Linux kernel source repository.User may need to modify it for linking with existing kernel sources.

```
bash> cd $ZYNQ_AXI_XADC_HOME/software/driver
bash> make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi-
```

19.6.5 5.5 Build Linux User Application

User application communicates with kernel driver and Lab View GUI. It configures Linux driver using standard IOCTL interface,read samples and store it configured circular buffers.Read commands from Lab View GUI through UART interface and reply with appropriate response.
For building it run make command on bash terminal.

```
bash> cd $ZYNQ_AXI_XADC_HOME/software/app
bash> make ARCH=arm CROSS_COMPILE=arm-xilinx-linux-gnueabi-
```

19.7
6 Setup Requirement

1. The ZC702 evaluation board with the XC7Z020 CLG484-1 part2. XAPP1183 zip file containing the design files and ready_to_test executables
2. Signal generator to test externally applied analog signal
4. Mini USB cable
5. Class 4 equivalent SD card
6. A control PC
7. AMS101 evaluator GUI.
Please follow the instructions in AMS101 targeted reference design to install the AMS101 GUI installer.
8. Terminal emulator software. i.e Teraterm

19.8
7 Execution Steps

This section summarizes the execution steps of the XAPP on ZC702 platform.

19.8.1 7.1 ZC702 Initial Setup

1. All jumpers and switches should be in default setting except SW16.
Mode switch SW16 should be set to boot from SD card.
Use the following switch settings:
SW16.1: OFF
SW16.2: OFF
SW16.3: ON
SW16.4: ON
SW16.5: OFF
2. Connect the AC power adapter
3. If USB-to-UART bridge is used, connect USB Mini-B side of USB-to-Mini-B cable to the on-board mini USB connector (J17). Connect USB side to the control PC.

19.8.2 7.2 Execution steps

1. Copy the XAPP binaries to SD card partition. XAPP1183 binaries can be found at $ZYNQ_AXI_XADC_HOME/ready_to_test directory.
2. Connect Xilinx AMS101 Evaluation Card output to ZC702 evaluation board XADC HDR slot ,and provide analog signal using external signal generator.
3. Configure SD boot mode and power on ZC702 evaluation board.
4. Initialization script provided in the package auto configures required setup. On start-up it loads the kernel

module and then run the Linux user application.
5. Run AMS evaluator GUI on host PC.
6. Select appropriate COM port in LabView and click on connect TAB.
7 Setup is now configured and ready to use. Select continuous/single data acquisition mode on LabView GUI.
8 Depending on external signal input , real-time samples are captured and displayed in time/frequency domain.

NOTE: User application running on zc702 evaluation can terminated by typing "E" on teraterm console.

19.9 8 References

- User Guide for XAPP1183
- Documentation for Xilinx AMS Evaluator card.
- Documentation for Zynq-7000 SoC
- Documentation for ZC702 Evaluation Kit
- Main Xilinx wiki

20 XAPP1231 - Partial Reconfiguration of a Hardware Accelerator with Vivado Design Suite

20.1 Table of Contents

20.2
1 Introduction

This tutorial shows how to develop a Partial Reconfiguration (PR) design for the Zynq-7000 SoC using the Xilinx Vivado Design Suite, Vivado HLS, Software Development Kit (XSDK), and PetaLinux design tools. It complements application note XAPP1231 which focuses on conceptual aspects of the PR flow and design considerations specific to the Zynq architecture.

The previous versions of this reference design is available at the links below:

ISE 14.4 XAPP1159 - Partial Reconfiguration of a Hardware Accelerator on Zynq-7000 SoC Devices

20.2.1 1.1 Design Overview

The PR reference design is built on top of the ZC702 Base Targeted Reference Design (TRD) , an embedded video processing application that demonstrates how it is best to separate control and data processing functions. In this example a compute-intensive video filtering algorithm is moved from the Processing System (PS) onto a hardware accelerator in Programmable Logic (PL). The image filter IP core demonstrated in the ZC702 Base TRD is a Sobel filter configured with edge detection coefficients which has been generated using Vivado HLS. For this reference design, three image filter IP cores are generated using Vivado HLS: Posterize, Sobel, FAST. The provided reference design demonstrates how to use software-controlled Partial Reconfiguration through the Processor Configuration Access Port (PCAP) to dynamically reconfigure part of the PL with the desired image filter IP core and observe the modified video output on a monitor.

20.2.2 1.2 Requirements

20.2.2.1 Software Tools

- Vivado Design Suite 2014.4 System Edition, includes Vivado High-Level Synthesis (HLS)
- Xilinx Software Development Kit 2014.4
- PetaLinux 2014.4
- Silicon Labs CP210x USB to UART Bridge VCP Driver
- Git distributed version control system (optional)

Note: Some tools are optional and the corresponding design flow tutorials can be skipped.

20.2.2.2 Hardware

- ZC702 Evaluation Kit
- Monitor with HDMI or DVI port that supports 1080p60 video resolution
- Avnet FMC-IMAGEON module and external video source that provides 1080p60 video input over HDMI (optional)
- USB mouse and keyboard (optional)

20.2.2.3 Licensing

- Xilinx Partial Reconfiguration is a product inside the Vivado Design Suite that requires a license
- Xilinx IP evaluation licenses may be provided with the Vivado Design Suite or can be ordered online
- Xylon logiCVC-ML is provided as evaluation IP core that does not require a license. The evaluation IP core has a 1 hour timeout built-in such that the display output freezes after the timer expires. The pre-built bitstreams and boot images are built from a full logiCVC-ML IP core.

20.2.3 1.3 Design Files

20.2.3.1 Download

Download and unzip the reference design archive file xapp1231-partial-reconfig-hw-accelerator-vivado.zip to a local directory.

20.2.3.2 Directory Structure

The following files are inside the xapp1231-partial-reconfig-hw-accelerator-vivado top-level directory:

- doc -- Readme and License files
- hardware -- Hardware sources
    - vivado_hls -- Vivado HLS projects for image filters
        - fast_corners -- FAST
        - sobel -- Sobel
        - simple_posterize -- Posterize
    - vivado_pr -- Vivado Partial Reconfiguration project-less design scripts
    - vivado_trd -- Vivado IP Integrator project for Zynq Base TRD
- ready_to_test -- Pre-built SD card image

- software -- Software sources
    - boot -- Zynq boot image sources
    - petalinux -- PetaLinux board support package (BSP)
    - pre-built -- Pre-built Linux open-source libraries, binaries and headers
    - xsdk -- XSDK projects
        - filter_lib -- HLS image filter library
        - perfmon_lib -- AXI performance monitor library
        - video_cmd -- Command line application
        - video_lib -- Xilinx Video library
        - video_qt -- GUI application based on Qt framework

The xapp1231-partial-reconfig-hw-accelerator-vivado directory is the starting point for each of the below tutorials.

20.2.4 1.4 Known Issues

See Known Issues section on Zynq Base TRD wiki.

20.3 2 Running the Reference Design

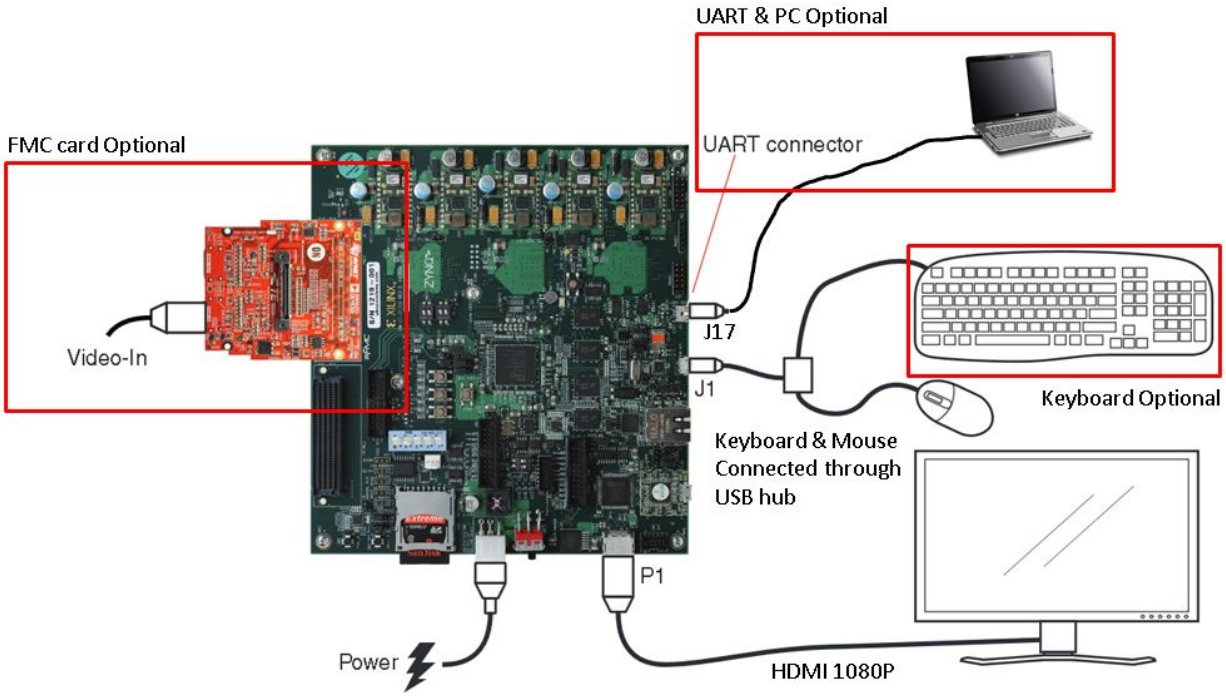
This section describes how to set up and run the reference design applications on the ZC702 evaluation board. The Qt application requires a video resolution of 1080p60 whereas the command line application supports arbitrary resolutions. For either application, the video input and video output resolutions need to match.

20.3.1 2.1 Setting up the ZC702 Evaluation Board

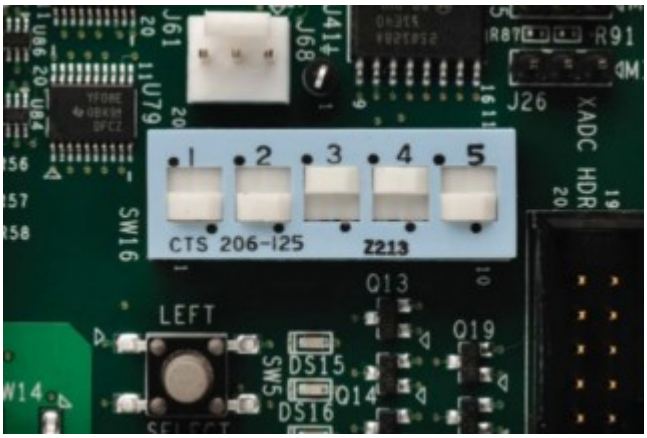
Refer to the ZC702 Evaluation Kit Quick Start Guide for kit contents and default jumper and switch settings. Reference design specific settings are described below.

Connect a power supply to connector J60 on the ZC702 board. Connect a 1080p60-capable monitor to the HDMI port on the ZC702 board. Connect a USB mouse and keyboard (**optional for Qt application**) to the Micro USB port J1 on the ZC702 board. Connect the FMC-IMAGEON daughter card to the FMC2 slot on the ZC702 board and an HDMI cable to the HDMI IN port on the FMC card (**optional - only required to enable**

external video input).



Set the boot mode switch SW16 to SD boot mode.



Connect a Laptop or PC to the Mini USB-UART port J17 on the ZC702 board (**optional for Qt application**). Use the following serial port settings:

Baud Rate	115200
Data	8 bit
Parity	None

Stop	1 bit
Flow Control	None

20.3.2 2.2 SD Card Image

The following files need to be copied to a FAT formatted SD card. A pre-built SD card image can be found in the ready_to_test directory:

- BOOT.BIN -- boot image containing FSBL, full Sobel bitstream, and u-boot
- autostart.sh -- if present, this script is run at the end of the boot process; it starts the Qt application
- bin -- binary executables
    - run_video.sh -- wrapper script for calling command line or Qt application
    - video_cmd -- command line application
    - video_qt -- GUI application based on Qt framework
- devicetree.dtb -- Linux device tree binary with FMC card disabled (default)
- devicetree_fmc.dtb -- Linux device tree binary with FMC card enabled - rename to devicetree.dtb to enable
- partial -- partial bitstreams
    - fast_corners.bin -- FAST filter
    - simple_posterize -- Posterize filter
    - sobel.bin -- Sobel filter
- uImage -- Linux kernel
- uramdisk.image.gz -- Linux root file system

20.3.3 2.3 Software Applications

Insert an SD card with the files listed above into the SD card slot on the ZC702 board and power on the board. The Qt Linux application will start automatically after the system is booted.

20.3.3.1 Qt Application

To start the application, run:

```
run_video.sh -qt -p
```

The -p switch enables Partial Reconfiguration.

Use the mouse to navigate the GUI. The following is a list of control elements and monitors as they appear in the GUI from left to right and top to bottom:

GUI Transparency	Slider to set alpha blending value between 0 and 50
Video Controls	Radio buttons to turn on/off video or select auto mode
Video Source	Combo box to select TPG or HDMI input
Test Pattern	Combo box to select displayed test pattern
Filter Type	Combo box to select between fast_corners, simple_posterize, or sobel filter
Filter Mode	Radio buttons to turn off or select software or hardware accelerated filter
Sobel Controls	Only available when sobel filter and SW or HW filter mode are selected; check box to invert filter colors; slider to set edge sensitivity threshold value between 100 and 255
Performance	Textual visualization of CPU0/CPU1 utilization and HP0/HP2 memory throughput
Exit	Exit application
Min/Max	Minimize/maximize GUI
CPU Utilization	Graphical visualization of CPU0/CPU1 utilization
Memory Throughput HP Ports	Graphical visualization HP0/HP2 memory throughput

To display more options, run:

```
run_video.sh -qt -h
```

20.3.3.2 Command Line Application

Connect a terminal emulator to the serial port assigned to the ZC702 UART. You can use XSDK's built-in terminal or a standalone application like TeraTerm for example.

To start the application, run:

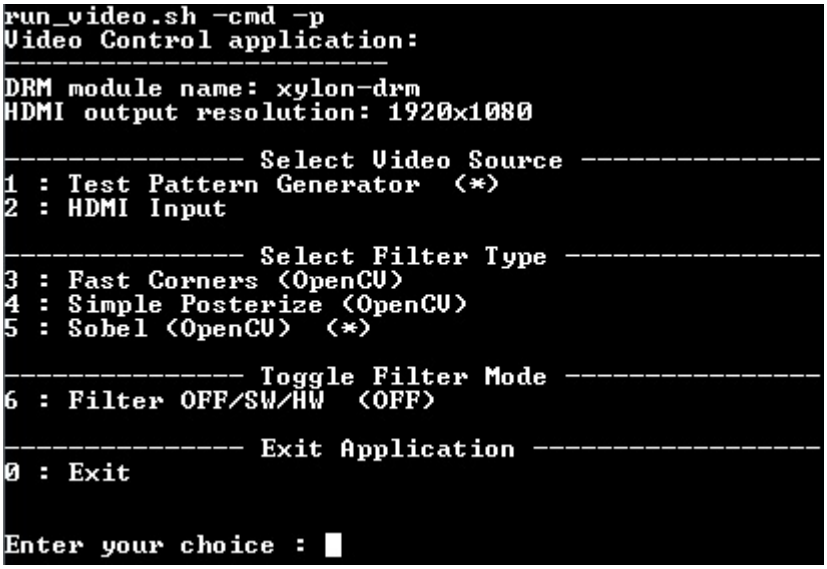
```
run_video.sh -cmd -p -r 'width'x'height'
```

The -p switch enables Partial Reconfiguration. If the -r switch is omitted, the default resolution is set to 1920x1080.

Use the terminal window to control the command line application. The following is a list of control elements:

Video Source	Select TPG or HDMI input - the current selection is marked by asterisk

Filter Type	Select fast_corners, simple_posterize or sobel filter - the current selection is marked by asterisk
Filter Mode	Toggle filter modes off, software, or hardware - the current selection is printed in brackets
Exit	Exit application



To display more options, run:

```
run_video.sh -cmd -h
```

20.4 3. Vivado HLS

Vivado HLS provides a tool and methodology for migrating algorithms coded in C, C++ or System-C from the Zynq PS onto the PL by generating RTL code. Three different image filters are used in this design and the source files are located in the respective directories listed under hardware/vivado_hls:

FAST	fast_corners

Sobel	sobel
Posterize	simple_posterize

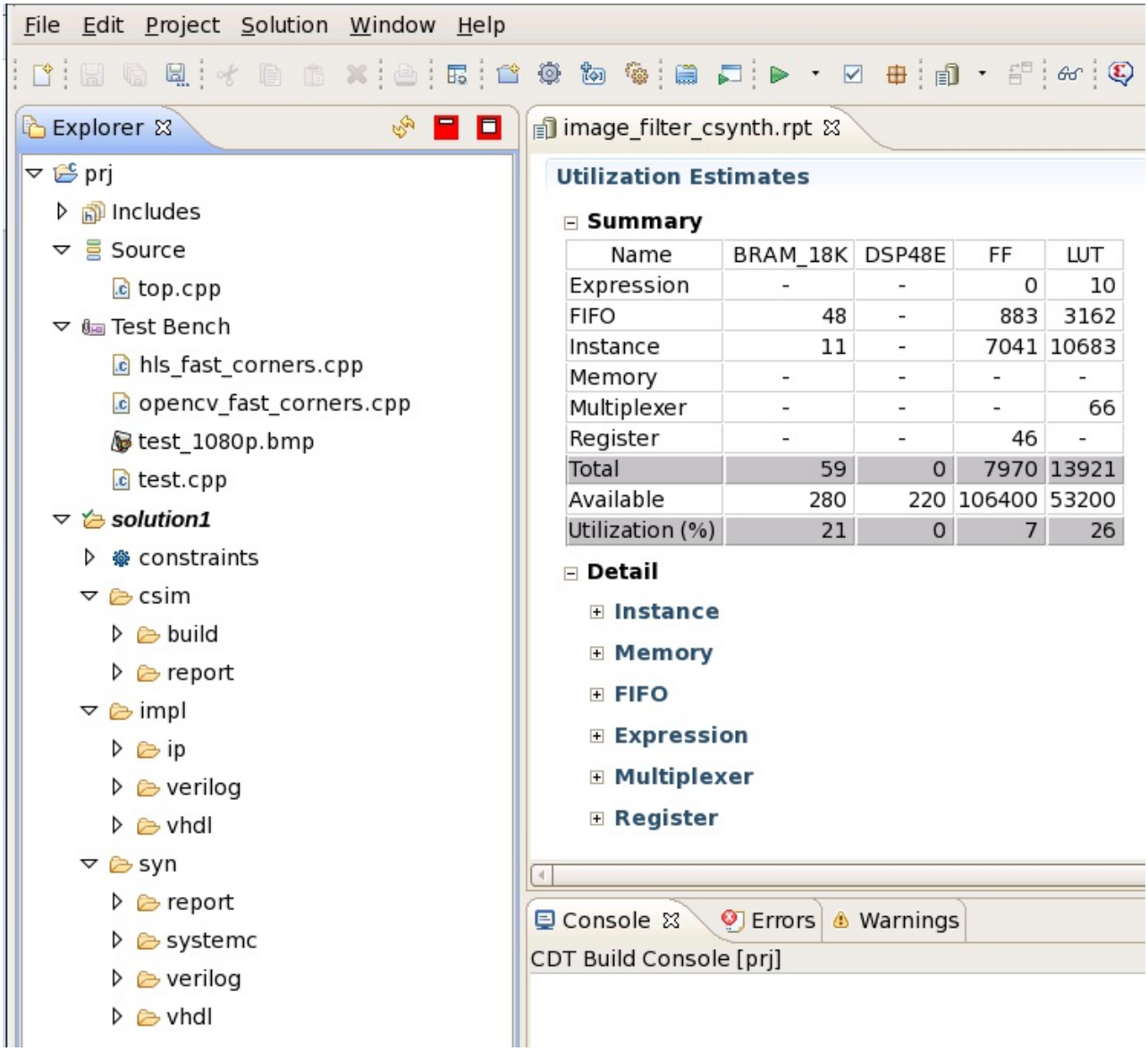
In this design a scripted HLS flow is used. For example, to generate the FAST filter, run:

```
cd hardware/vivado_hls/fast_corners
vivado_hls -f run.tcl
```

To open the generated HLS project, run:

```
vivado_hls -p prj
```

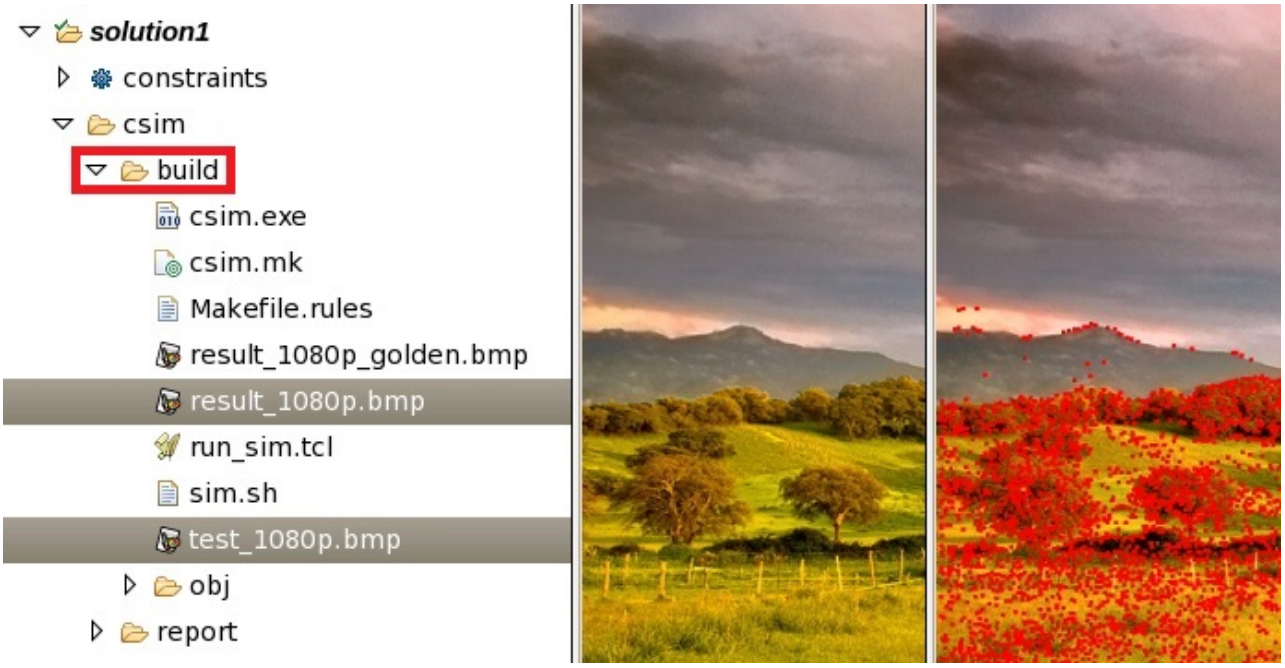
Below screenshot shows the HLS project inside the Vivado HLS IDE and the generated synthesis report file:



The following table shows the steps run by the scripts and where the corresponding output products are located under prj/solution1/:

C-Simulation	csim
High-level synthesis	syn
IP packaging	impl/ip/xilinx_com_hls_image_filter_1_0.zip

Expand the csim/build folder in the Vivado HLS explorer view and open the files test_1080p.bmp and result_1080p.bmp to view the original and processed images after running C Simulation:



Repeat the above steps for all three HLS image filters.

The following files are generated in this tutorial will be used in step 5:

FAST filter IP core	fast_corners/prj/solution1/impl/ip/ xilinx_com_hls_image_filter_1_0.zip
Posterize filter IP core	simple_posterize/prj/solution1/impl/ip/ xilinx_com_hls_image_filter_1_0.zip
Sobel filter IP core	sobel/prj/solution1/impl/ip/ xilinx_com_hls_image_filter_1_0.zip

---

20.5

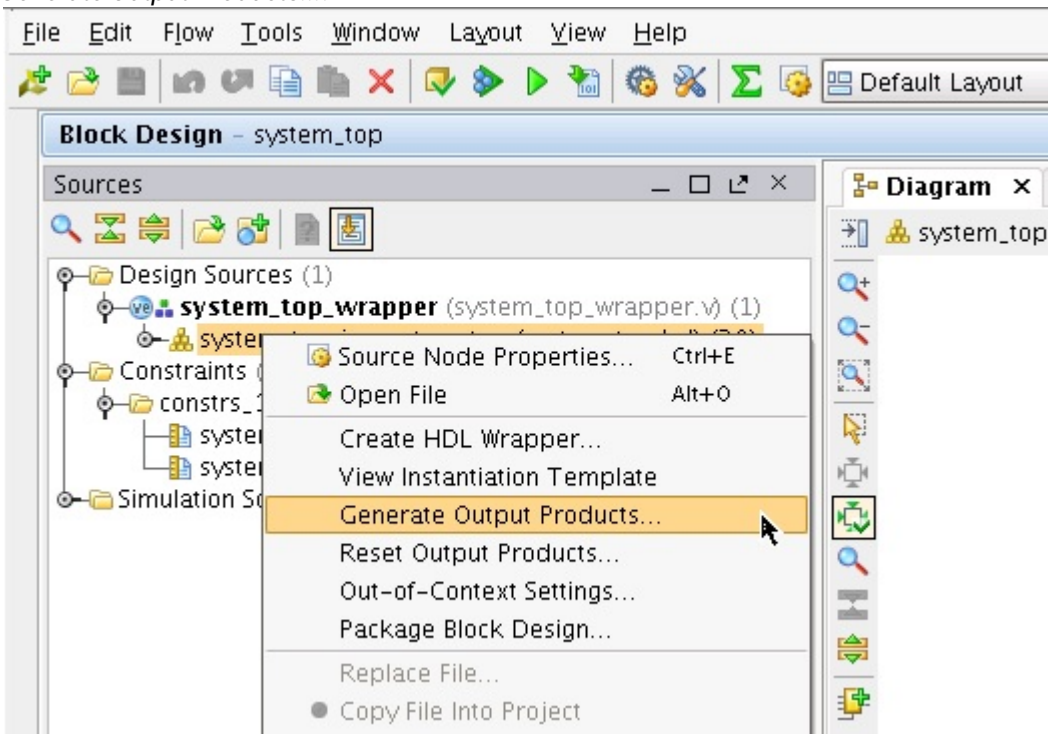
4. Vivado - Base TRD

In this step we will generate a synthesis design checkpoint for the static logic i.e. the portion of the design that can not be reconfigured. We are using the ZC702 Base TRD Vivado IP Integrator project as a starting point. The Base TRD includes the Sobel HLS image filter IP core generated in section 3 above. We will carve out the Sobel filter declare a black box module instead which will be the Reconfigurable Module (RM) in this design. We will also floorplan the design by assigning a Reconfigurable Partition (RP) to the RM. The RP is the region that can be reconfigured using Partial Reconfiguration (PR).

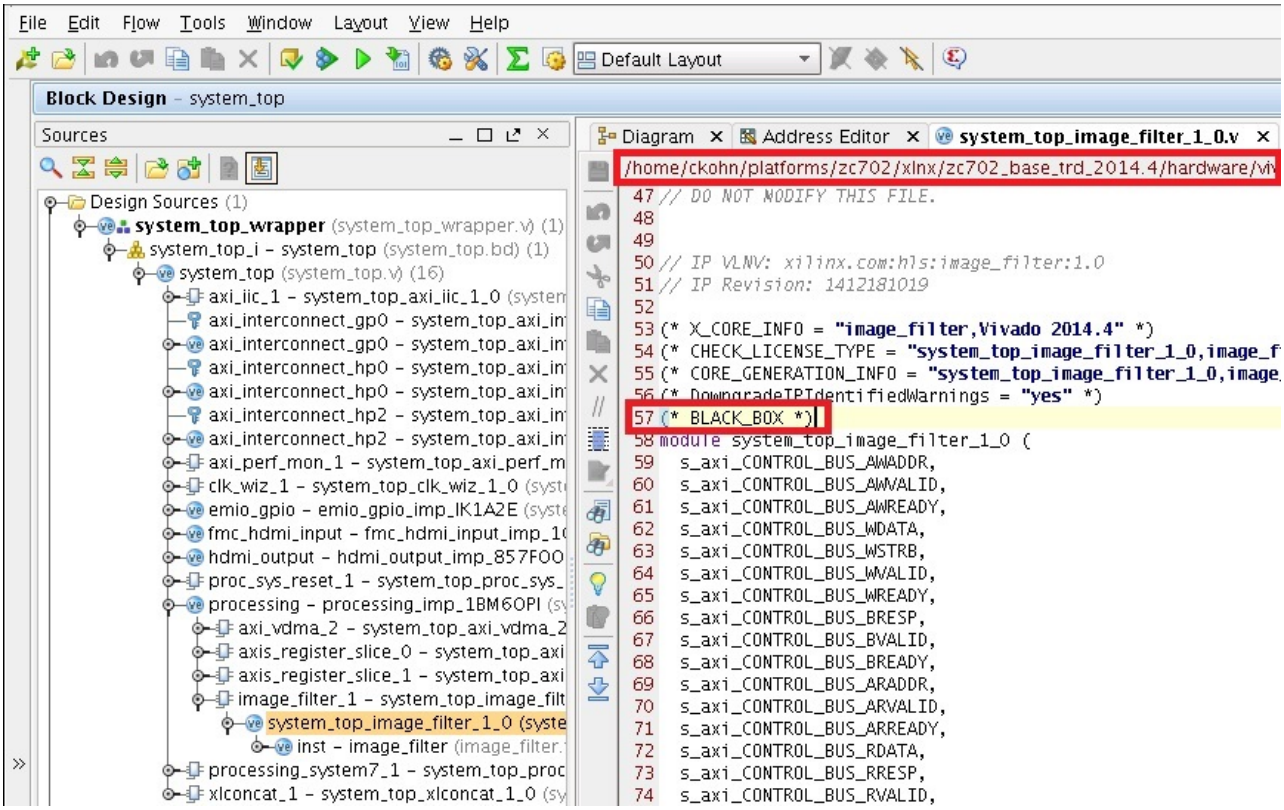
To generate and open the Vivado IP Integrator project, run:

```
cd hardware/vivado_trd
vivado -source ./scripts/project.tcl
```

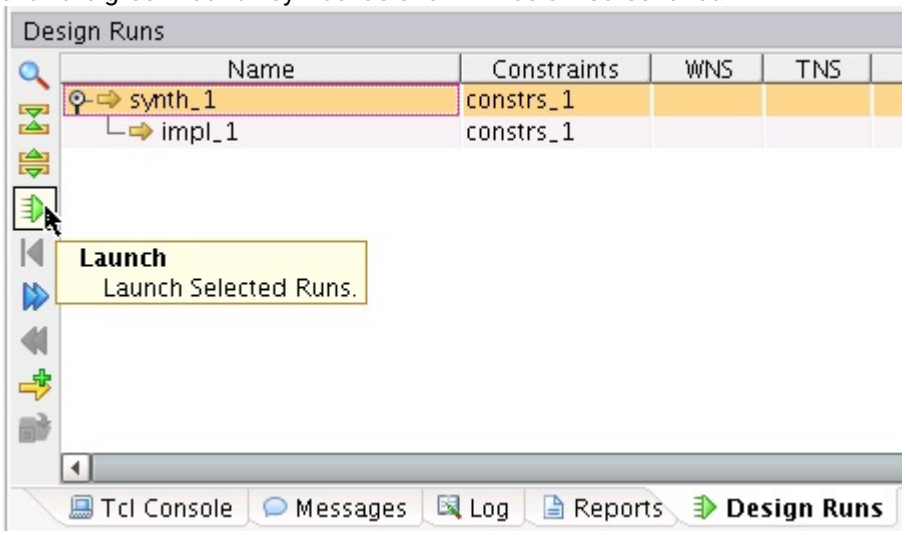
In the Vivado IDE, right-click on the system_top block diagram source file in the *Sources* view and select *Generate Output Products...*:



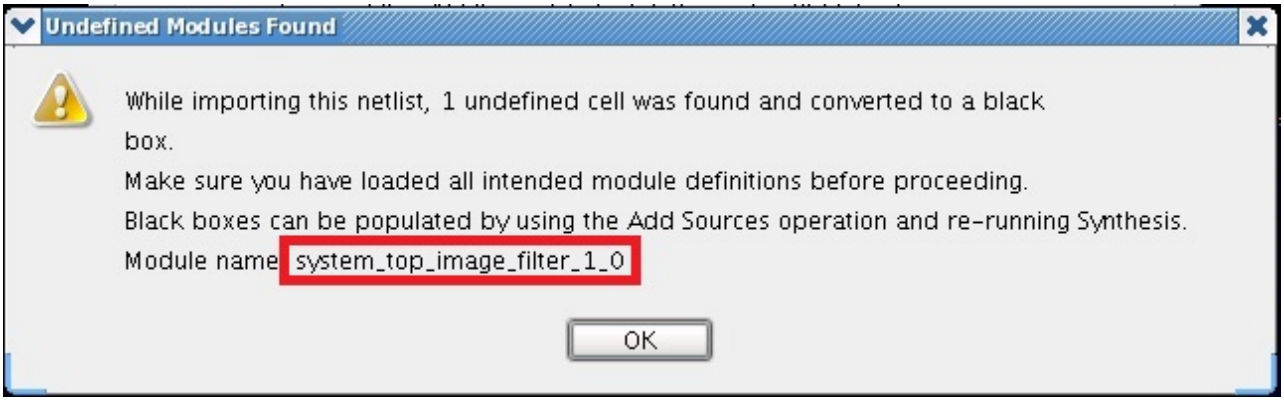
After the HDL files have been generated, expand the hierarchy tree all the way down to system_top/ processing/system_top_image_filter_1_0 as shown in the screenshot below. Double click on the system_top_image_filter_1_0 source file to open it. Since this file is generated by IP Integrator, it is marked as read-only and cannot be modified within the Vivado editor. Copy the full path of the file (see red box at the top) and open the file in an editor outside of Vivado. Insert the BLACK_BOX property on line 57 as shown in the screenshot. Save the file and switch back to the Vivado view and click reload at the top of the editor view to update the changes.



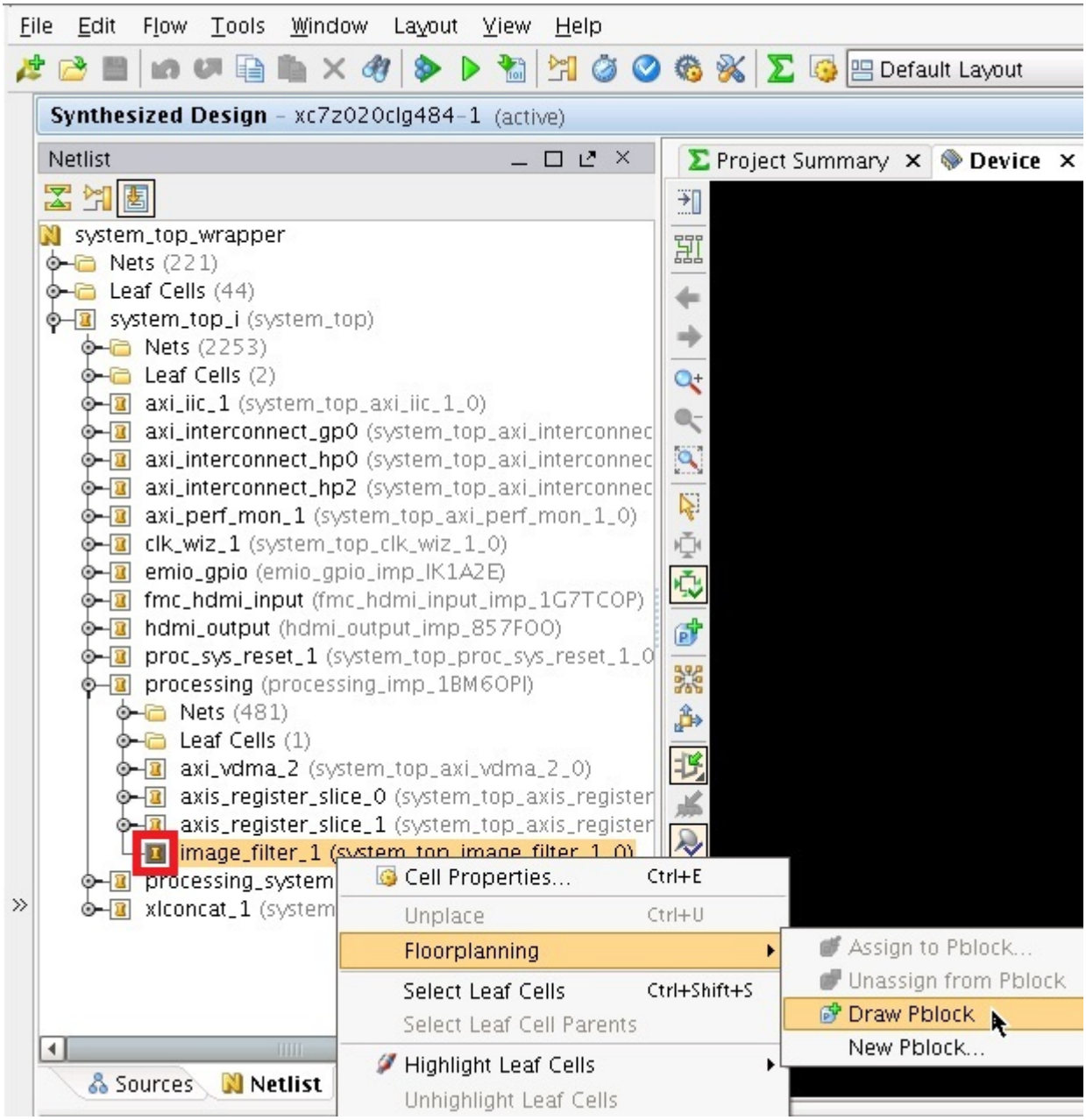
The black box attribute assigned to the Sobel filter wrapper module ensures that the module is not synthesized in this step. To run synthesis, mark the synth_1 run in the *Design Runs* view at the bottom and click the green *Launch* symbol as shown in below screenshot:



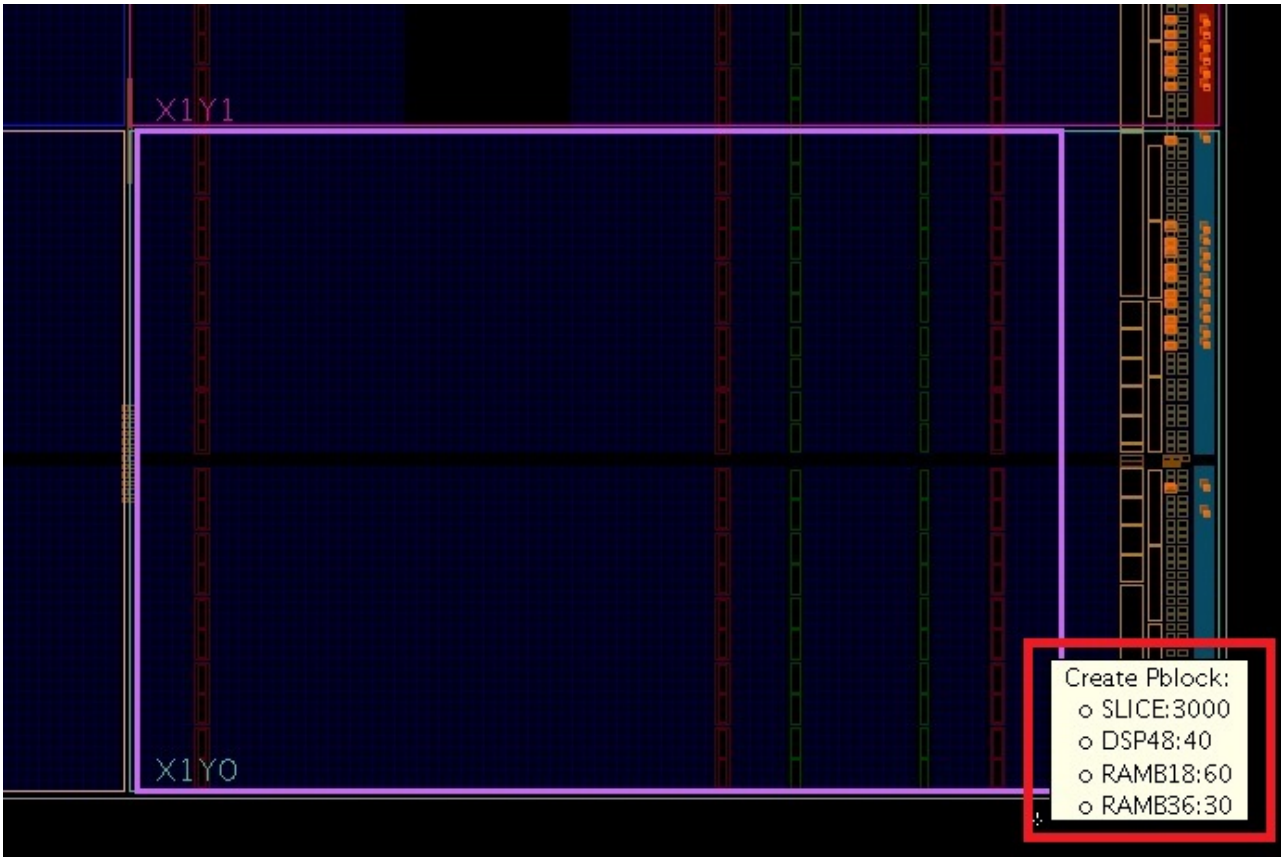
Once synthesis is finished, Vivado will prompt you for the next step. Select *Open Synthesized Design*. The next prompt is a critical warning, stating that no synthesized netlist could be found for the black box module system_top_image_filter_1_0 defined previously. This is expected and intentional. Confirm with *OK*.



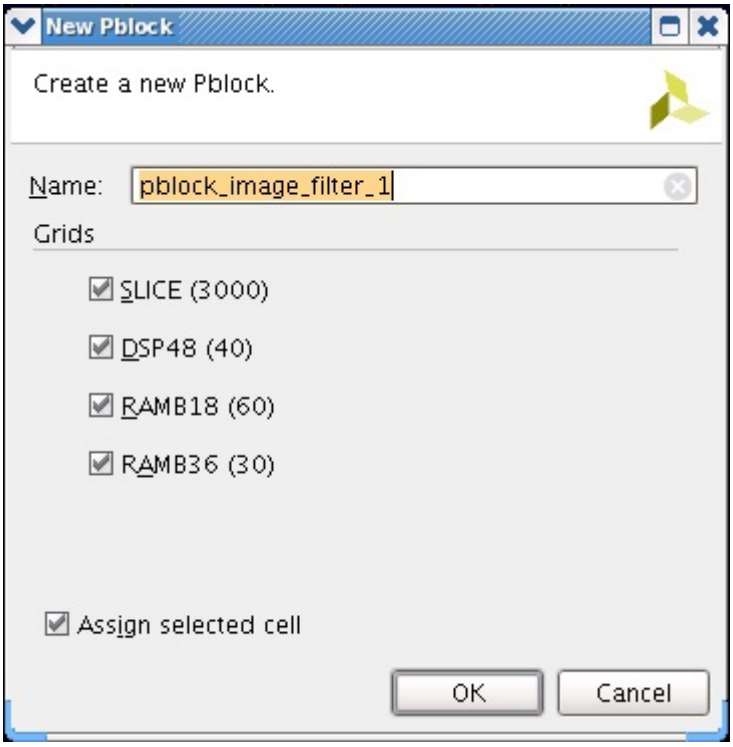
Right-click on the system_top_image_filter_1_0 black box module in the *Netlist* view. Note the black icon compared to the other netlists. Select *Floorplanning* > *Draw Pblock*



Assign a Reconfigurable Partition to the black box module by drawing a Pblock in the desired area in the *Device View*. Zoom into the X1Y0 clock region first, then left-click and drag the mouse to draw a rectangle as shown in the screenshot below. Make sure the resources preview box shows the same numbers.

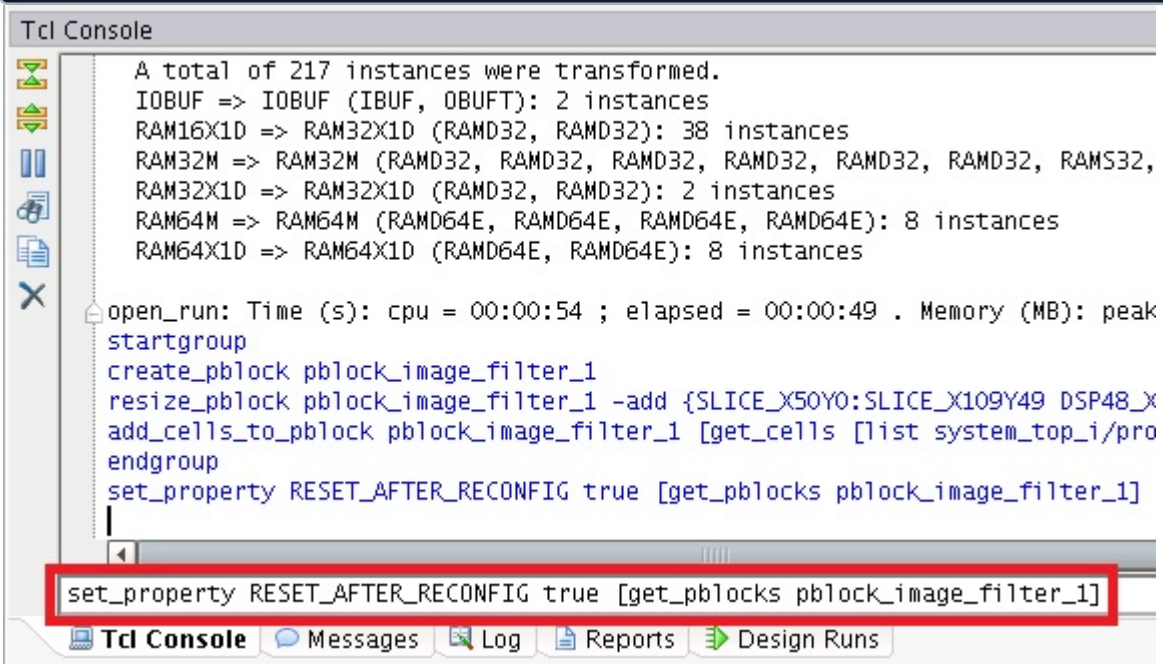


In the next prompt, make sure that all available resources and the *Assign selected cell* box are checked. Confirm with OK.



Next we set the RESET_AFTER_RECONFIG on the Pblock. This will ensure that all the resources in the Reconfigurable Partition are automatically reset every time a new module is loaded into this region. Run the following command in the *Tcl Console* at the bottom:

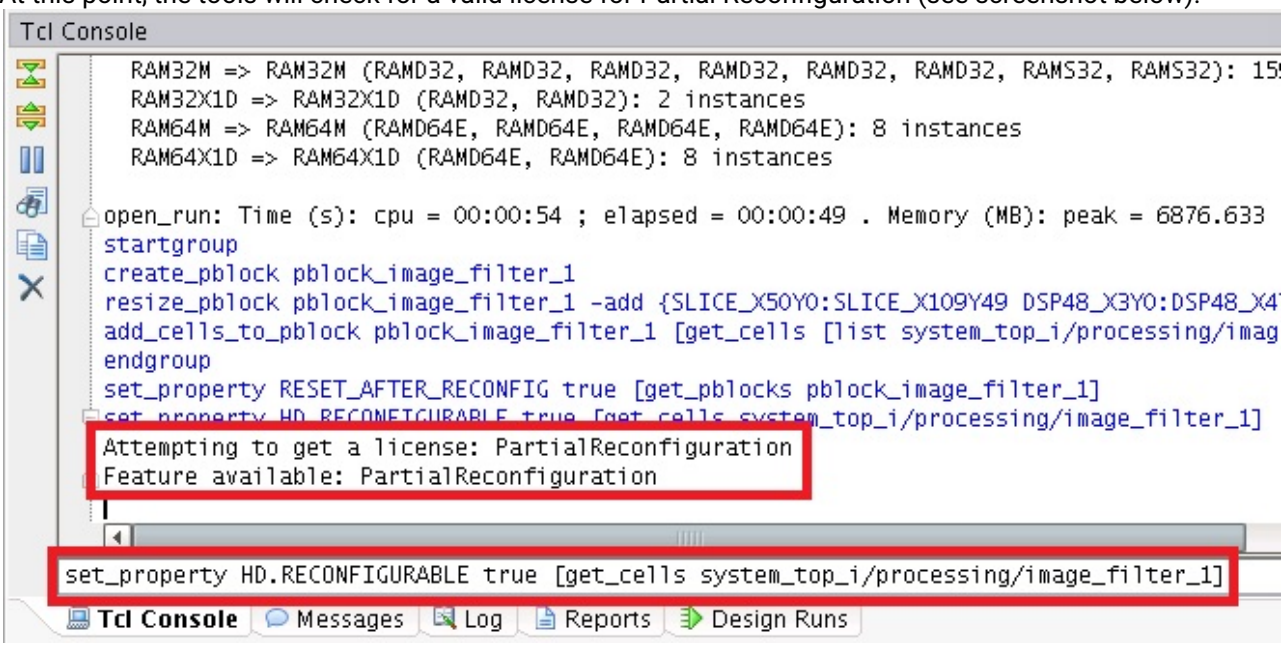
```
set_property RESET_AFTER_RECONFIG true [get_pblocks pblock_image_filter_1]
```



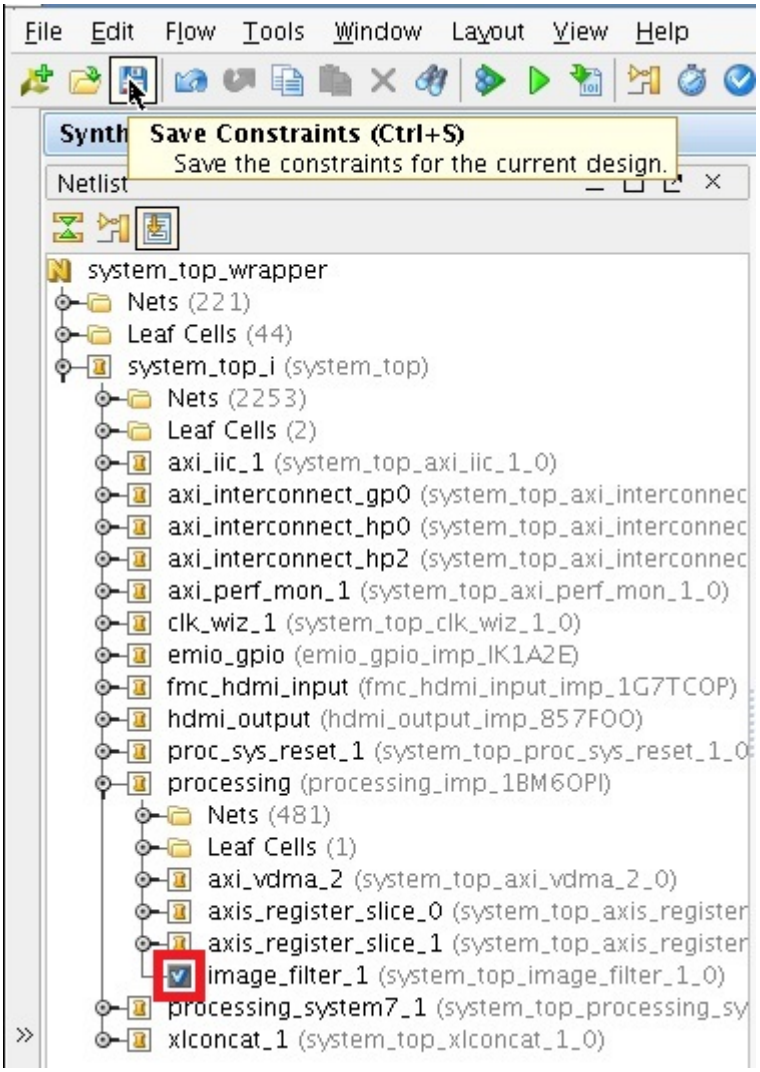
Finally, we set the HD.RECONFIGURABLE property on the image filter black box module.Run the following command in the *Tcl Console*:

```
set_property HD.RECONFIGURABLE true [get_cells system_top_i/processing/
image_filter_1]
```

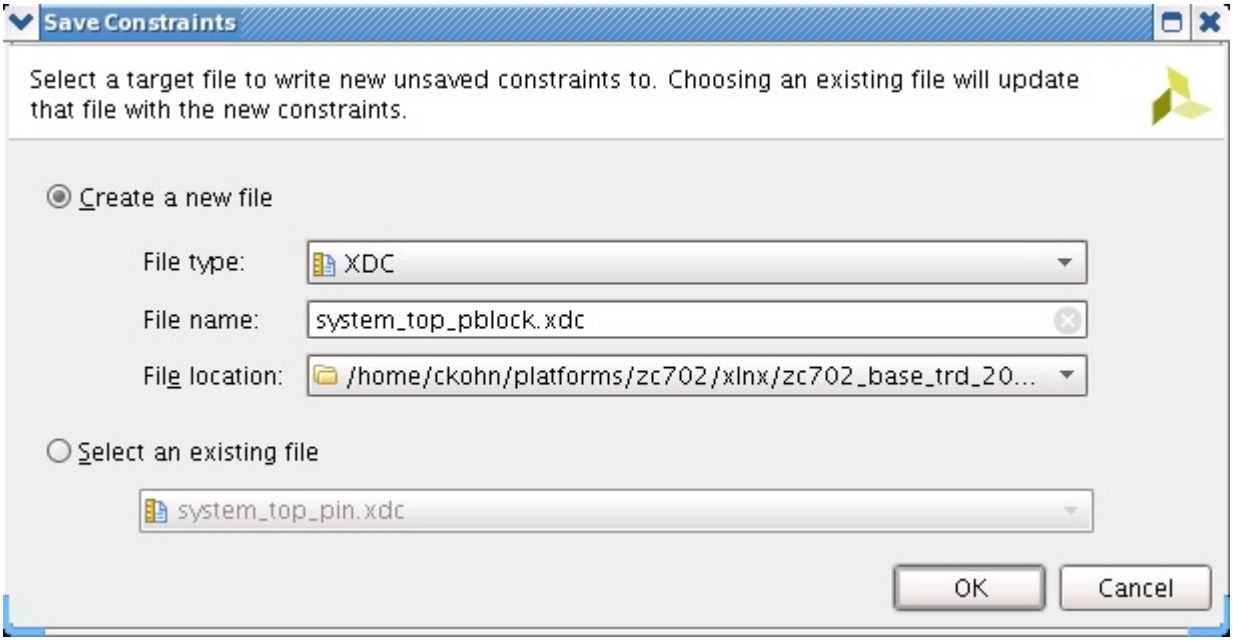
At this point, the tools will check for a valid license for Partial Reconfiguration (see screenshot below).



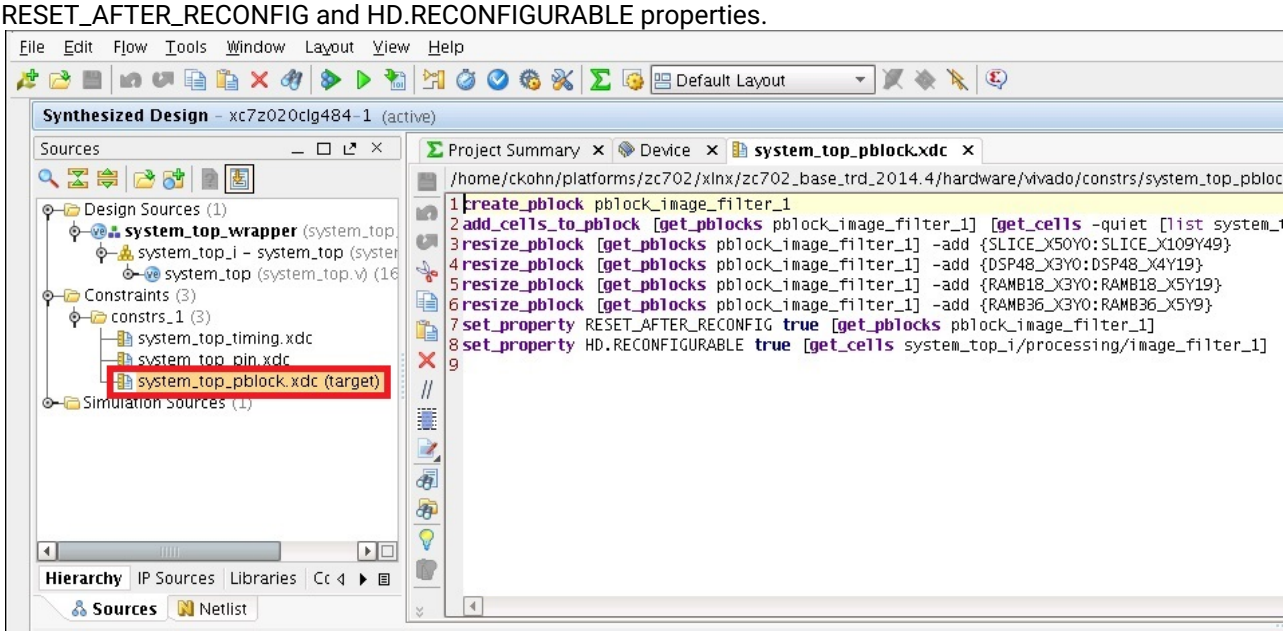
Now save the constraints by left clicking the disk symbol at the top of the screen. Note that the icon for the image filter module has changed again after assigning a Pblock to the module.



Save the constraints to a new file to the location constrs/system_top_pblock.xdc. Confirm with *OK*.



In the *Sources* view, expand the *Constraints* hierarchy and double-click on system_top_pblock.xdc to open and view the newly generated constraints file. Constraints are created for the Pblock and the RESET_AFTER_RECONFIG and HD.RECONFIGURABLE properties.



Exit Vivado.

The following files are generated in this tutorial and will be used in step 5:

Synthesis design checkpoint for static logic	project/zynq_base_trd_2014.4.runs/synth_1/system_top_wrapper.dcp

IP Integrator generated HDL wrapper file for HLS image filter IP core	project/zynq_base_trd_2014.4.srcs/sources_1/bd/system_top/ip/system_top_image_filter_1_0/synth/system_top_image_filter_1_0.v
XDC constraints file for Partial Reconfiguration	constrs/system_top_pblock.xdc

---

20.6
5. Vivado - Partial Reconfiguration

In this step we will run a project-less, script based Vivado flow for Partial Reconfiguration. We define Reconfigurable Modules and Configurations in a tcl file. We out-of-context (OOC) synthesize each Reconfigurable Modules. We link the RMs to the already synthesized static logic and implement each Configuration. Finally, we generate full bitstreams for each Configuration and partial bitstreams for each Reconfigurable Module.

20.6.1 Input Source Files

We are using the following source files and output products from steps 4 and 5 as inputs in this step. The following table lists the origin project and the relative location to the vivado_pr directory:

Vivado HLS	unzipped FAST IP core	srcs/ip/fast_corners
Vivado HLS	unzipped Posterize IP core	srcs/ip/simple_posterize
Vivado HLS	unzipped Sobel IP core	srcs/ip/sobel
Vivado Base TRD	PR constraints	constrs/system_top_pblock.xdc
Vivado Base TRD	Pin constraints	constrs/system_top_pin.xdc
Vivado Base TRD	Timing constraints	constrs/system_top_timing.xdc
Vivado Base TRD	Synthesized design checkpoint	srcs/dcp/system_top_wrapper.dcp

Vivado Base TRD	HLS image filter wrapper	srcs/hdl/ system_top_image_filter_1_0.v

20.6.2 RM Synthesis File

For each of the HLS image filter IP cores, we need to create a .prj file listing the HDL sources and the image filter wrapper source file required for OOC synthesis. The files are located at srcs/prj. For example, this is what the simple_posterize.prj file looks like:

```
verilog xil_defaultLib ./srcs/ip/simple_posterize/hdl/verilog/
FIFO_image_filter_cols36_channel.v
verilog xil_defaultLib ./srcs/ip/simple_posterize/hdl/verilog/
FIFO_image_filter_img_0_cols_V_channel.v
verilog xil_defaultLib ./srcs/ip/simple_posterize/hdl/verilog/
FIFO_image_filter_img_0_data_stream_0_V.v
verilog xil_defaultLib ./srcs/ip/simple_posterize/hdl/verilog/
FIFO_image_filter_img_0_data_stream_1_V.v
verilog xil_defaultLib ./srcs/ip/simple_posterize/hdl/verilog/
FIFO_image_filter_img_0_rows_V_channel.v
verilog xil_defaultLib ./srcs/ip/simple_posterize/hdl/verilog/
FIFO_image_filter_img_1_cols_V.v
verilog xil_defaultLib ./srcs/ip/simple_posterize/hdl/verilog/
FIFO_image_filter_img_1_data_stream_0_V.v
verilog xil_defaultLib ./srcs/ip/simple_posterize/hdl/verilog/
FIFO_image_filter_img_1_data_stream_1_V.v
verilog xil_defaultLib ./srcs/ip/simple_posterize/hdl/verilog/
FIFO_image_filter_img_1_rows_V.v
verilog xil_defaultLib ./srcs/ip/simple_posterize/hdl/verilog/
FIFO_image_filter_rows37_channel.v
verilog xil_defaultLib ./srcs/ip/simple_posterize/hdl/verilog/
image_filter_AXIvideo2Mat.v
verilog xil_defaultLib ./srcs/ip/simple_posterize/hdl/verilog/
image_filter_Block_proc.v
verilog xil_defaultLib ./srcs/ip/simple_posterize/hdl/verilog/
image_filter_CONTROL_BUS_s_axi.v
verilog xil_defaultLib ./srcs/ip/simple_posterize/hdl/verilog/
image_filter_Loop_1_proc.v
verilog xil_defaultLib ./srcs/ip/simple_posterize/hdl/verilog/
image_filter_Mat2AXIvideo.v
verilog xil_defaultLib ./srcs/ip/simple_posterize/hdl/verilog/image_filter.v
verilog xil_defaultLib ./srcs/hdl/system_top_image_filter_1_0.v
```

20.6.3 PR Design File

The Vivado PR scripts are located at scripts/tcl and are read only. The file scripts/design.tcl is the only file that needs to be modified to set up the sources and parameters for the PR run. We are looking at excerpts of the design.tcl file in the following.

This code snippet sources the Vivado PR tcl scripts and makes the corresponding routines available.

```
##################################################################
###   Tcl Variables
##################################################################
####Define location for "Tcl" directory. Defaults to "./Tcl"
set tclHome "./scripts/tcl"
if &#123;&#91;file exists $tclHome&#93;&#125; &#123;
   set tclDir $tclHome
&#125; elseif &#123;&#91;file exists "./tcl"&#93;&#125; &#123;
   set tclDir  "./tcl"
&#125; else &#123;
   error "ERROR: No valid location found for required Tcl scripts. Set \$tclDir in
design.tcl to a valid location."
&#125;
puts "Setting TCL dir to $tclDir"

####Source required Tcl Procs
source $tclDir/design_utils.tcl
source $tclDir/log_utils.tcl
source $tclDir/synth_utils.tcl
source $tclDir/impl_utils.tcl
source $tclDir/hd_floorplan_utils.tcl
```

This code snippet sets up the target part, package and speed grade for implementation.

```
##################################################################
### Define Part, Package, Speedgrade
##################################################################
set device       "xc7z020"
set package      "clg484"
set speed        "-1"
set part         $device$package$speed
check_part $part
```

This code snippet sets up some basic variables to control the PR flow and where the input and output directories are located.

```
##################################################################
```

```
###  Setup Variables
#################################################################
#set tclParams [list <param1> <value> <param2> <value> ... <paramN> <value>]
set tclParams &#91;list hd.visual 1 \
              &#93;

####flow control
set run.rmSynth        1
set run.prImpl         1
set run.prVerify       1
set run.writeBitstream 1
set run.flatImpl       0

####Report and DCP controls - values: 0-required min; 1-few extra; 2-all
set verbose      1
set dcpLevel     1

####Output Directories
set synthDir  "./Synth"
set implDir   "./Implement"
set dcpDir    "./Checkpoint"
set bitDir    "./Bitstreams"

####Input Directories
set srcDir     "./srcs"
set xdcDir     "./constrs"
set rtlDir     "$srcDir/hdl"
set ipDir      "$srcDir/ip"
set prjDir     "$srcDir/prj"
```

This code snippet defines the static logic of the design. Synthesis will not be run on this module since we already have a synthesized design checkpoint available from step 4.

```
#################################################################
### Top Definition
#################################################################
set top "system_top_wrapper"
set static "static"
add_module $static
set_attribute module $static moduleName      $top
set_attribute module $static top_level       1
set_attribute module $static synthCheckpoint $srcDir/dcp/system_top_wrapper.dcp
```

This code snippet defines the three HLS image filters as Reconfigurable Modules of this design. The variant name, an XDC constraints file, and a list of HDL files (.prj file) is provided per RM. The RM instance name in the design hierarchy (black box module) is defined at the bottom. OOC synthesis is run on each of the RMs.

```
#####################################################################
```

```
### RP Module Definitions
########################################################################
set module1 "system_top_image_filter_1_0"

set module1_variant1 "fast_corners"
set variant $module1_variant1
add_module $variant
set_attribute module $variant moduleName   $module1
set_attribute module $variant prj          $prjDir/$variant.prj
set_attribute module $variant xdc          $ipDir/$variant/constraints/
image_filter_ooc.xdc
set_attribute module $variant synth        $&#123;run.rmSynth&#125;

set module1_variant2 "sobel"
set variant $module1_variant2
add_module $variant
set_attribute module $variant moduleName   $module1
set_attribute module $variant prj          $prjDir/$variant.prj
set_attribute module $variant xdc          $ipDir/$variant/constraints/
image_filter_ooc.xdc
set_attribute module $variant synth        $&#123;run.rmSynth&#125;

set module1_variant3 "simple_posterize"
set variant $module1_variant3
add_module $variant
set_attribute module $variant moduleName   $module1
set_attribute module $variant prj          $prjDir/$variant.prj
set_attribute module $variant xdc          $ipDir/$variant/constraints/
image_filter_ooc.xdc
set_attribute module $variant synth        $&#123;run.rmSynth&#125;

set module1_inst1 "system_top_i/processing/image_filter_1"
```

This code snippet defines three Configurations, one for each RM. For each Configuration, the XDC constraints files, the static module, and the RM are defined. Note that the first configuration has the static module set to implement whereas the second and third Configurations have it set to import under the partitions attribute. The static logic only needs to be implemented i.e. placed and routed once. After that it is locked down, saved, and re-used for the remaining partitions which is indicated by the import flag. The PR verify design rule check (DRC) is run after implementation to check whether the implementation results are in fact compatible. Full and partial bitstreams are generated for each Configuration and RM respectively. The partial bitstreams are converted to the binary PCAP format since we are using the Processor Configuration Access Port (PCAP) to load the bitstreams.

```
########################################################################
### Configuration (Implementation) Definition - Replicate for each Config
########################################################################
set config "Config_${module1_variant1}"

add_implementation $config
set_attribute impl $config top             $top
set_attribute impl $config pr.impl         1
```

```
set_attribute impl $config implXDC          &#91;list $xdcDir/system_top_timing.xdc \
                                                  $xdcDir/system_top_pin.xdc \
                                                  $xdcDir/system_top_pblock.xdc \
                                            &#93;
set_attribute impl $config impl             $&#123;run.prImpl&#125;
set_attribute impl $config partitions       &#91;list &#91;list $static           $top
implement&#93; \
                                                  &#91;list $module1_variant1
$module1_inst1 implement&#93; \
                                            &#93;
set_attribute impl $config verify           $&#123;run.prVerify&#125;
set_attribute impl $config bitstream        $&#123;run.writeBitstream&#125;
set_attribute impl $config cfgmem.pcap      1

########################################################################
### Configuration (Implementation) Definition - Replicate for each Config
########################################################################
set config "Config_${module1_variant2}"

add_implementation $config
set_attribute impl $config top              $top
set_attribute impl $config pr.impl          1
set_attribute impl $config implXDC          &#91;list $xdcDir/system_top_timing.xdc \
                                                  $xdcDir/system_top_pin.xdc \
                                                  $xdcDir/system_top_pblock.xdc \
                                            &#93;
set_attribute impl $config impl             $&#123;run.prImpl&#125;
set_attribute impl $config partitions       &#91;list &#91;list $static           $top
import&#93;    \
                                                  &#91;list $module1_variant2
$module1_inst1 implement&#93; \
                                            &#93;
set_attribute impl $config verify           $&#123;run.prVerify&#125;
set_attribute impl $config bitstream        $&#123;run.writeBitstream&#125;
set_attribute impl $config cfgmem.pcap      1

########################################################################
### Configuration (Implementation) Definition - Replicate for each Config
########################################################################
set config "Config_${module1_variant3}"

add_implementation $config
set_attribute impl $config top              $top
set_attribute impl $config pr.impl          1
set_attribute impl $config implXDC          &#91;list $xdcDir/system_top_timing.xdc \
                                                  $xdcDir/system_top_pin.xdc \
                                                  $xdcDir/system_top_pblock.xdc \
                                            &#93;
set_attribute impl $config impl             $&#123;run.prImpl&#125;
set_attribute impl $config partitions       &#91;list &#91;list $static           $top
import&#93;    \
                                                  &#91;list $module1_variant3
$module1_inst1 implement&#93; \
```

```
                                          &#93;
set_attribute impl $config verify         $&#123;run.prVerify&#125;
set_attribute impl $config bitstream      $&#123;run.writeBitstream&#125;
set_attribute impl $config cfgmem.pcap    1
```

Now that all the parameters are set up, the main PR tcl script is sourced which in turn calls all the required sub-routines.

```
##########################################################################
### Task / flow portion
##########################################################################
source $tclDir/run.tcl
```

20.6.4 PR Log File

To start the Vivado PR flow, run:

```
cd vivado_pr
vivado -mode batch -source ./scripts/design.tcl -notrace
```

The script will run autonomously and no further user intervention is required. The following are excerpts of the run log generated by the PR scripts.

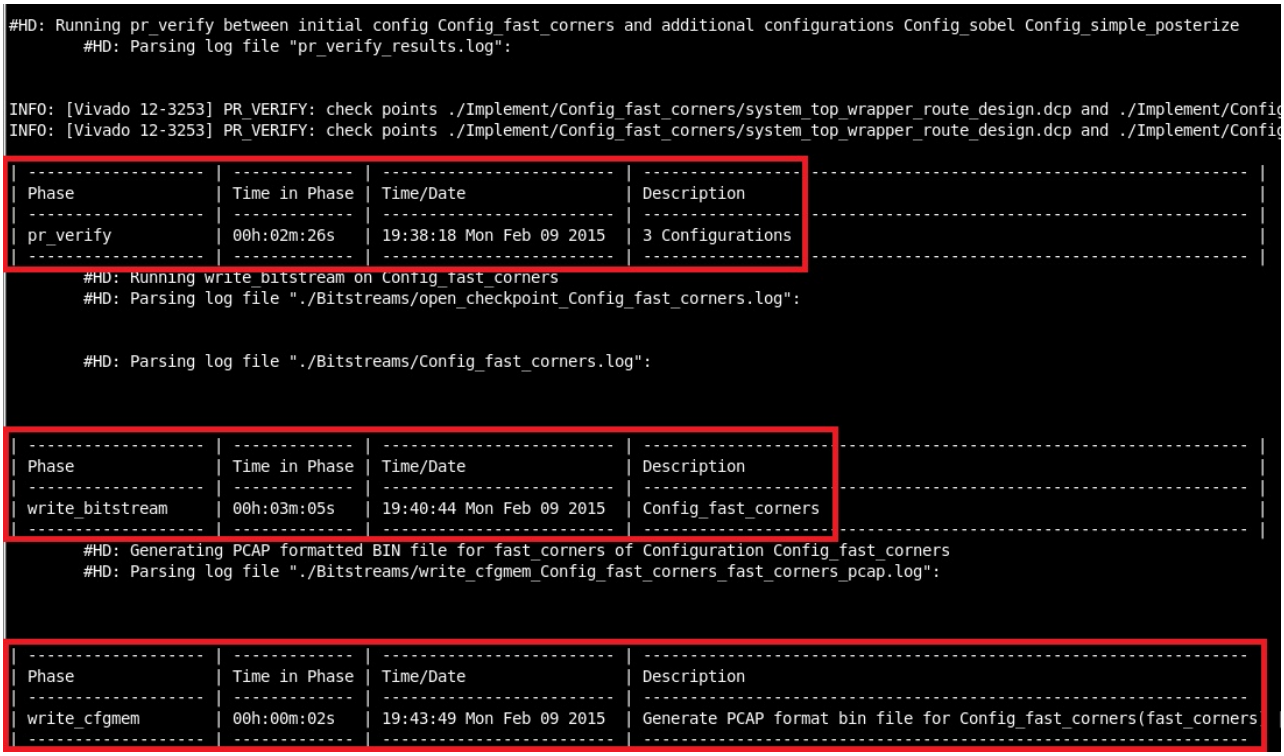
This snippet shows a list of the modules to be synthesized and their selected synthesis options. Note that this list only includes the HLS image filters i.e. the Reconfigurable Modules and hence Top Level is set to 0. The top level module (system_top_wrapper) i.e. the static logic is listed as module not being synthesized since we're just reading in an already synthesized design checkpoint.

```
****** Vivado v2014.4 (64-bit)
  **** SW Build 1071353 on Tue Nov 18 16:47:07 MST 2014
  **** IP Build 1070531 on Tue Nov 18 01:10:18 MST 2014
    ** Copyright 1986-2014 Xilinx, Inc. All Rights Reserved.

INFO: [Common 17-206] Exiting Vivado at Mon Feb  9 19:11:06 2015...
INFO: [Common 17-1239] XILINX_LOCAL_USER_DATA is set to 'NO'.
source ./scripts/design.tcl -notrace
Setting TCL dir to ./scripts/tcl
INFO: Found part matching xc7z020clg484-1
INFO: Found Vivado version 2014.4

#HD: List of modules to be synthesized:
| ----------------- | --------------------------- | ---------- | -------------------------- |
| Module            | Module Name                 | Top Level  | Options                    |
| ----------------- | --------------------------- | ---------- | -------------------------- |
| fast_corners      | system_top_image_filter_1_0 | 0          | -flatten_hierarchy rebuilt |
| ----------------- | --------------------------- | ---------- | -------------------------- |
| sobel             | system_top_image_filter_1_0 | 0          | -flatten_hierarchy rebuilt |
| ----------------- | --------------------------- | ---------- | -------------------------- |
| simple_posterize  | system_top_image_filter_1_0 | 0          | -flatten_hierarchy rebuilt |
| ----------------- | --------------------------- | ---------- | -------------------------- |

#HD: Defined modules not being synthesized:
       1. static (system_top_wrapper)
```

This snippet shows a list of configurations to be implemented. The module associated with each Configuration is listed, as well as additional options in the flow like PR verification or bitstream generation. It is also listed if the static logic is implemented as part of this Configuration or if it is imported from a previous Configuration run.

```
#HD: List of Configurations to be implemented:
| ----------------------- | --------------------------- | ------------- | ---------- | --------------- |
| Configuration           | Reconfig Modules            | Static State  | pr_verify  | write_bistream  |
| ----------------------- | --------------------------- | ------------- | ---------- | --------------- |
| Config_fast_corners     | fast_corners(implement)     | implement     | 1          | 1               |
| ----------------------- | --------------------------- | ------------- | ---------- | --------------- |
| Config_sobel            | sobel(implement)            | import        | 1          | 1               |
| ----------------------- | --------------------------- | ------------- | ---------- | --------------- |
| Config_simple_posterize | simple_posterize(implement) | import        | 1          | 1               |
| ----------------------- | --------------------------- | ------------- | ---------- | --------------- |
```

This snippet shows a summary of additional post-implementation steps like PR verification, bitstream generation, and bitstream formatting for a specific interface like PCAP.



The final output products of this step are full bitstreams for each Configuration and partial bitstreams for each Reconfigurable Module. We use the full Sobel bitstreams as default for the boot image i.e. the PL gets programmed with this bitstream during the boot process. The boot image is generated in step 7. The partial bitstreams are formatted correctly and can be deployed to the SD card image from where they are loaded by the application as discussed in step 6.

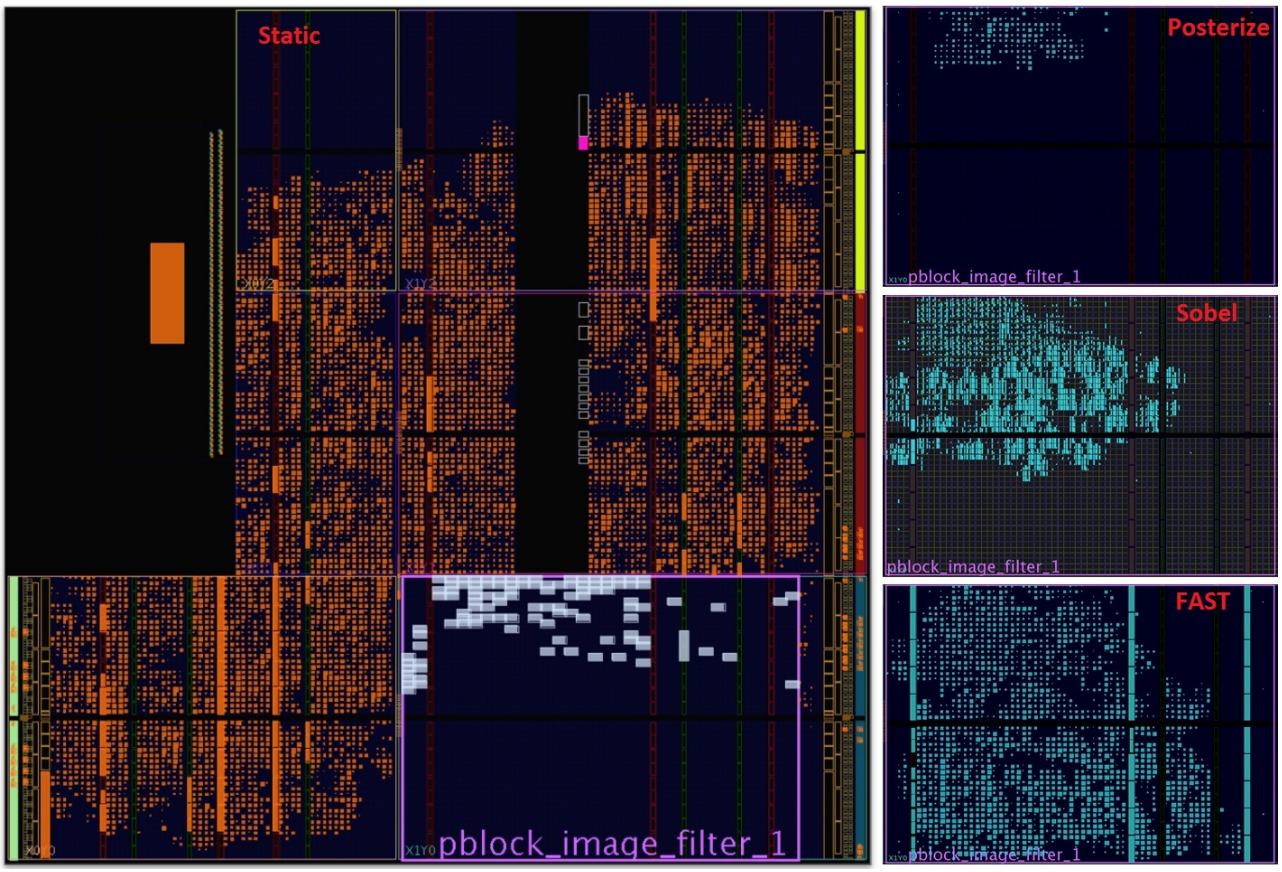
20.6.5 PR Device View

After the Vivado PR run has finished, we can inspect report files or view saved design checkpoints for the various implementation stages.

To open a saved implemented design checkpoint, for example for the static logic, run:

```
vivado ./Checkpoint/system_top_wrapper_static.dcp
```

The following screenshot is a composition of the device view for the static logic (left) and the three RMs (right): simple_posterize (top), sobel (center), fast_corners (bottom). The static resources are highlighted in orange, the Pblock is a purple box in the bottom-right corner. No resources are placed in this partition since it is reserved for the RMs. The gray boxes inside the Pblock are the ports of the Reconfigurable Modules interfacing with the static logic. The RM resources are highlighted in blue in their respective views. Note the difference in resource utilization between the filter variants.



20.7
6. Xilinx SDK

Two Linux applications are available in this reference design:

- a command line based application controlled from a UART terminal
- a GUI based application controlled by mouse from an HDMI monitor

Both applications are extensions of the corresponding Base TRD applications. For a tutorial on how to build the two Linux applications, please refer to the Zynq Base TRD wiki - XSDK. The page also contains information on how to debug and deploy an application using XSDK as well as how to measure CPU and HP/ACP port performance metrics using the integrated System Performance Monitor (SPM).

To enable the Partial Reconfiguration feature in either application, simply add the -p or --partial-reconfig switch when executing the application, for example:

```
run_video.sh -qt -p
```

The following code snippets show the two added functions for Partial Reconfiguration.

The first function identifies the partial bitstream by its name and loads the partial bitstreams from a pre-defined location on the SD card into a buffer variable in memory. This is to speed up the bitstream transfer when the PL is reconfigured in the subsequent step.

```
int filter_prefetch_bin()
&#123;
    char file_name&#91;128&#93;;
    int fd;
    int ret;
    unsigned int i;

    for (i = 0; i < ARRAY_SIZE(filter_types); ++i) &#123;
        if (filter_types&#91;i&#93;.pr_file_name&#91;0&#93; != '\0') &#123;
            // compose file name
            sprintf(file_name, "/media/card/partial/%s", filter_types&#91;i&#93;.pr_f
ile_name);

            // open partial bitfile
            fd = open(file_name, O_RDONLY);
            if (fd < 0) &#123;
                printf("failed to open partial bitfile %s\n", file_name);
                return -1;
            &#125;

            // read partial bitfile into buffer
            ret = read(fd, filter_types&#91;i&#93;.pr_buf, FILTER_PR_BIN_SIZE);
            if (ret < 0) &#123;
                printf("failed to read partial bitfile %s\n", file_name);
                close(fd);
                return -1;
            &#125;

            // close file handle
            close(fd);
        &#125;
    &#125;

    return 0;
&#125;
```

This function uses the xdevcfg driver to load the partial bitstream into the PL through the PCAP interface. The driver initiates a DMA transaction from memory to the configuration logic. Note that the size of the bitstream, defined by the FILTER_PR_BIN_SIZE macro, is required for this operation. Make sure the is_partial_bitstream flag is set to indicate that the bitstream is a partial bitstream.

```
int filter_config_bin(filter_type type)
&#123;
    int fd;

    // Set is_partial_bitfile device attribute
    fd = open("/sys/devices/soc0/amba/f8007000.devcfg/is_partial_bitstream", O_RDWR);
    if (fd < 0) &#123;
        printf("failed to set xdevcfg attribute 'is_partial_bitstream'\n");
        return -1;
    &#125;
    write(fd, "1", 2);
    close(fd);

    // Write partial bitfile to xdevcfg device
    fd = open("/dev/xdevcfg", O_RDWR);
    if (fd < 0) &#123;
        printf("failed to open xdevcfg device\n");
        return -1;
    &#125;
    write(fd, filter_type_pr_buf(type), FILTER_PR_BIN_SIZE);
    close(fd);

    return 0;
&#125;
```

20.8
7. PetaLinux

PetaLinux is an easy-to-use tool to create a complete boot-able Linux image including:

- boot image
    - first stage boot loader (FSBL)
    - bitstream (full Sobel)
    - u-boot (second stage boot loader)
- kernel
- device tree
- root file system

For a tutorial on how to build the PetaLinux image, please refer to the Zynq Base TRD wiki - PetaLinux. The page also contains information on PetaLinux installation.

References:

1. XAPP1231, Partial Reconfiguration of a Hardware Accelerator with Vivado Design Suite
2. UG925, Zynq-7000 SoC ZC702 Base Targeted Reference Design (Vivado Design Suite 2014.4)
3. ZC702 Evaluation Kit Quick Start Guide

- Feb 12, 2015

20.9 zynq-pr-rd

The latest version of this wiki page can be found at the link below:
XAPP1231 - Partial Reconfiguration of a Hardware Accelerator with Vivado Design Suite

21 Zynq 7000 Partial Reconfiguration Reference Design

This wiki page complements XAPP1159.

21.1 Table of Contents

21.2 1 Introduction

This tutorial shows how to develop a Partial Reconfiguration (PR) design for the Zynq-7000 SoC using the Xilinx Platform Studio (XPS), Software Development Kit (SDK), and PlanAhead design tools. It complements application note XAPP1159 which focuses on conceptual aspects of the PR flow and Zynq architecture specific design considerations.

21.2.1 1.1 Design Overview

The PR reference design is built on top of the ZC702 Base Targeted Reference Design (TRD) , an embedded video processing application that demonstrates how it is best to separate control and data processing functions. In this example a compute-intensive video filtering algorithm is moved from the Processing System (PS) onto a hardware accelerator in Programmable Logic (PL). The video filter IP core demonstrated in the ZC702 Base TRD is a Sobel filter configured with edge detection coefficients which has been generated using the High-Level Synthesis tool Vivado HLS. For this reference design, a second video filter IP core (Sepia filter) was generated again using Vivado HLS. The provided reference design demonstrates how to use software-controlled Partial Reconfiguration (PR) to dynamically reconfigure part of the PL with the desired video filter IP core and observe the video output on a monitor.

21.2.2 1.2 Requirements

21.2.2.1 Software

- ISE Design Suite v14.4 Embedded or System Edition
- Vivado HLS v2012.4 to generate video filter IP cores (optional).
- Silicon Labs CP210x USB to UART Bridge VCP Driver (only required when using standalone boot image).
- Terminal emulator software, for example TeraTerm (only required when using standalone boot image).
- Linux development PC with the ARM GNU cross compile tool chain and the Git tool installed (only required for completing Sections 5 and 6.5).

21.2.2.2 Hardware

- ZC702 Evaluation Kit to run the PR reference design in hardware.
- Monitor with HDMI or DVI port that supports 1080p60 video resolution.
- Avnet FMC-IMAGEON module and external video source that provides 1080p60 video input over HDMI (optional).
- USB hub, USB mouse, and USB keyboard (only required when using Linux boot image).

21.2.2.3 Licensing

- Xilinx Vivado HLS: A 30-day evaluation license can be generated after registering a Xilinx account.
- Xilinx ISE Design Suite System Edition: A 30-day evaluation license can be generated after registering a Xilinx account.
- Xilinx Partial Reconfiguration is a product inside the ISE Design Suite that requires a license. 30-day evaluation licenses are available through the Xilinx University Program (XUP).
- Xilinx IP evaluation licenses for the Video Timing Controller and Chroma Resampler IP cores can be ordered online.
- Xylon logiCVC-ML is shipped as evaluation IP core that does not require a license. License options are listed on the Xylon logiCVC-ML product site.
  Note: The provided logiCVC evaluation IP core has a 1 hour timeout built-in such that the display freezes after the timer expires. The pre-generated netlists are built from this evaluation IP core so the user can implement the design without having to purchase a license. The pre-built bitfiles and boot images are built from a full logiCVC IP core and don't expire.

21.2.3 1.3 Directory Structure

Download and unzip the reference design archive file xapp1159.zip to a local directory. The directory structure is:

- zc702_pr_rd -- Top-level directory

- doc -- Readme file
- hw -- Hardware sources
    - base_trd_prj -- Pre-configured Base TRD PlanAhead project
    - hls_sepia_prj -- Sepia filter Vivado HLS project
    - hls_sobel_prj -- Sobel filter Vivado HLS project
    - pr_prj -- Pre-configured PR PlanAhead project
- sd -- SD card images
    - linux -- Linux boot image, kernel, devicetree, ramdisk, executables, partial binaries
    - standalone -- Standalone boot image, partial binaries
- sw -- Software sources
    - boot -- Zynq boot image sources
        - linux -- Linux boot image sources
        - standalone -- Standalone boot image sources
    - patch -- Linux kernel patch
    - repo -- SDK standalone user repository
        - drivers -- SDK standalone user drivers
        - sw_services -- SDK standalone user software services
    - workspace -- SDK workspace/projects
        - filter_cmd -- Linux command line software application
        - filter_qt -- Linux Qt-GUI software application
        - filter_std -- Standalone software application
        - hw_platform -- Hardware platform information
        - standalone_bsp -- Bare-metal board support package
        - zynq_fsbl -- First stage boot loader

21.2.4 1.4 Known Issues

21.2.4.1 License error when using pre-synthesized logiCVC netlist

The provided pre-synthesized netlist for the logiCVC IP core (zc702_pr_rd/hw/pr_prj/zynq_pr_rd.srcs/sources_1/system_logicvc_0_wrapper.ngc) was generated from the full license core. Unless you posses a full license for this core, your implementation run will error out with the following message:

[Netlist 29-57] IP License required for netlist cell 'system_logicvc_0_wrapper', instantiated as 'system_i/LOGICVC_0'.
WARNING:Security:141 - No 'Source' license available for 'ip_xap_349logicvcml_TDP' version '1.0' (-5).
WARNING:Security:141 - No 'Bought' license available for 'ip_xap_349logicvcml_TDP' version '1.0' (-5).
WARNING:Security:141 - No 'Hardware_Evaluation' license available for 'ip_xap_349logicvcml_TDP' version '1.0' (-5).
WARNING:Security:141 - No 'Design_Linking' license available for 'ip_xap_349logicvcml_TDP' version '1.0' (-5).
ERROR:Security:142 - No IP Core license of type 'Design_Linking' or greater available for 'ip_xap_349logicvcml' version '1.0'.

Please replace the netlist with the one attached or run Section 3 first followed by Sections 4.1 through 4.9. In Section 3 you will generate a netlist from the evaluation logiCVC pcore which you will then import into the PlanAhead PR project in Section 4.1.

21.2.4.2 Image tearing

Occasional horizontal image tearing can be observed when the sobel or sepia filter engines are enabled.

21.3 2 Vivado HLS Flow

Vivado HLS provides a tool and methodology for migrating algorithms coded in C, C++ or System-C from the Zynq PS onto the PL by generating RTL code. The Sobel filter IP core used in the Zynq Base TRD was generated using this approach. Similarly, the Sepia filter IP core can be generated based on the provided C-algorithm and Vivado HLS project. The following tutorial is based on XAPP890 and shows how to implement the Sobel filter HLS project -- the same methodology can be used for the Sepia filter HLS project.

Shortcut: Pre-generated Sobel and Sepia filter IP cores are available at zc702_pr_rd/hw/base_trd_prj/zynq_base_trd.srcs/sources_1/edk/xps_proj/pcores/sepia_filter_top_v1_04_a and zc702_pr_rd/hw/base_trd_prj/zynq_base_trd.srcs/sources_1/edk/xps_proj/pcores/sobel_filter_top_v1_04_a.

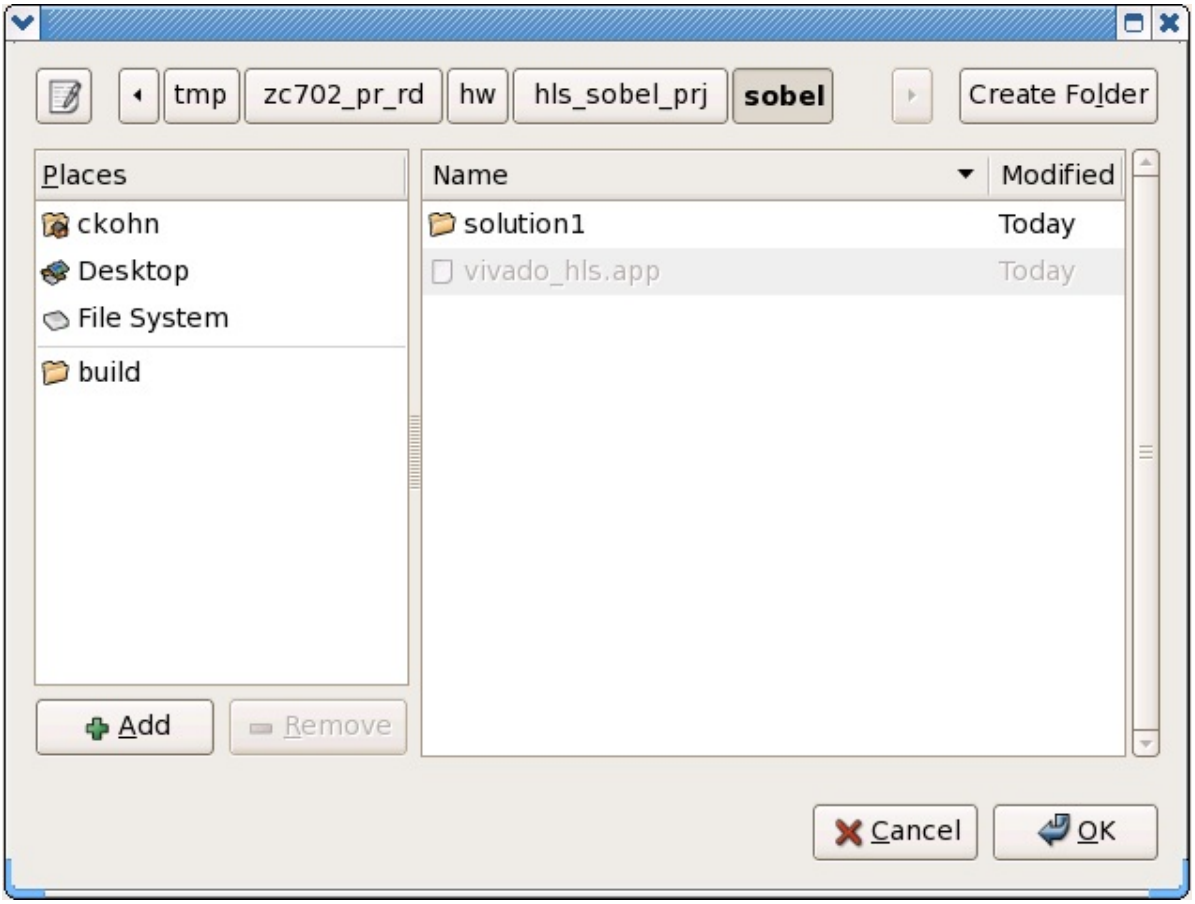
21.3.1 2.1 Synthesizing the HLS Design

Tutorial

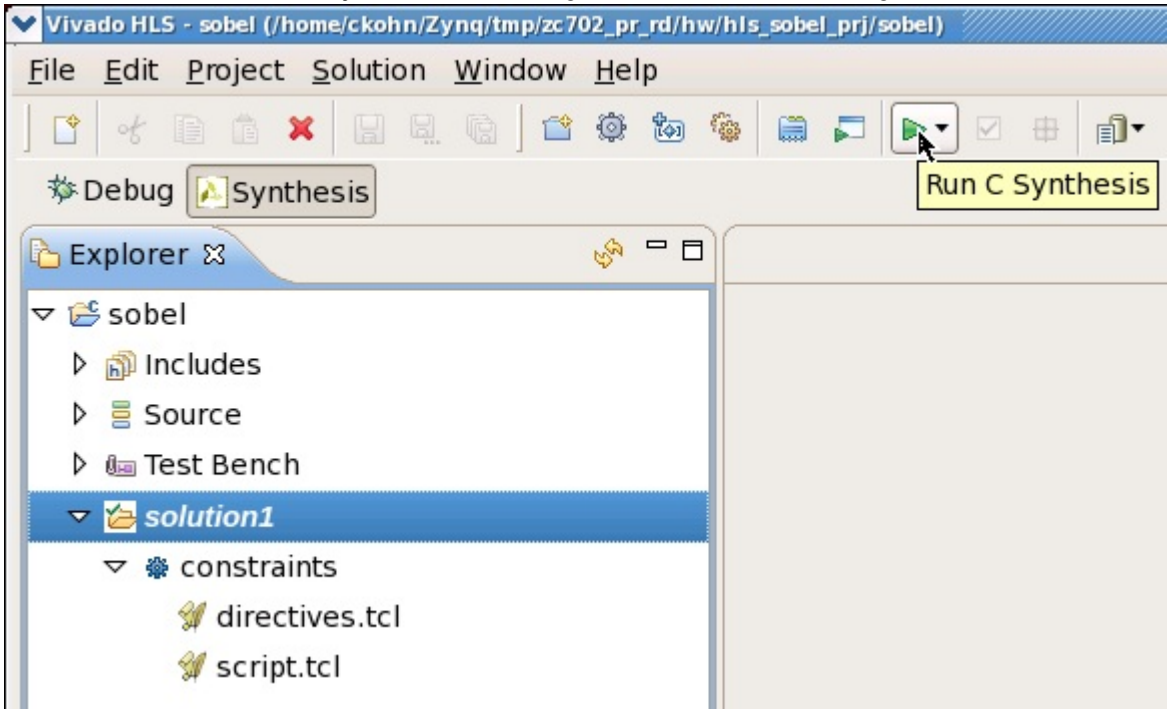
---

1. To open Vivado HLS, select **Start > All Programs > Xilinx Design Tools > Vivado 2012.4 > Vivado HLS**.
2. On the Vivado HLS welcome screen, click **Open Project** under the **Getting Started** group.

3. In the **Open Project** dialog, browse to the zc702_pr_rd/hw/hls_sobel_prj/sobel directory and click **OK**.



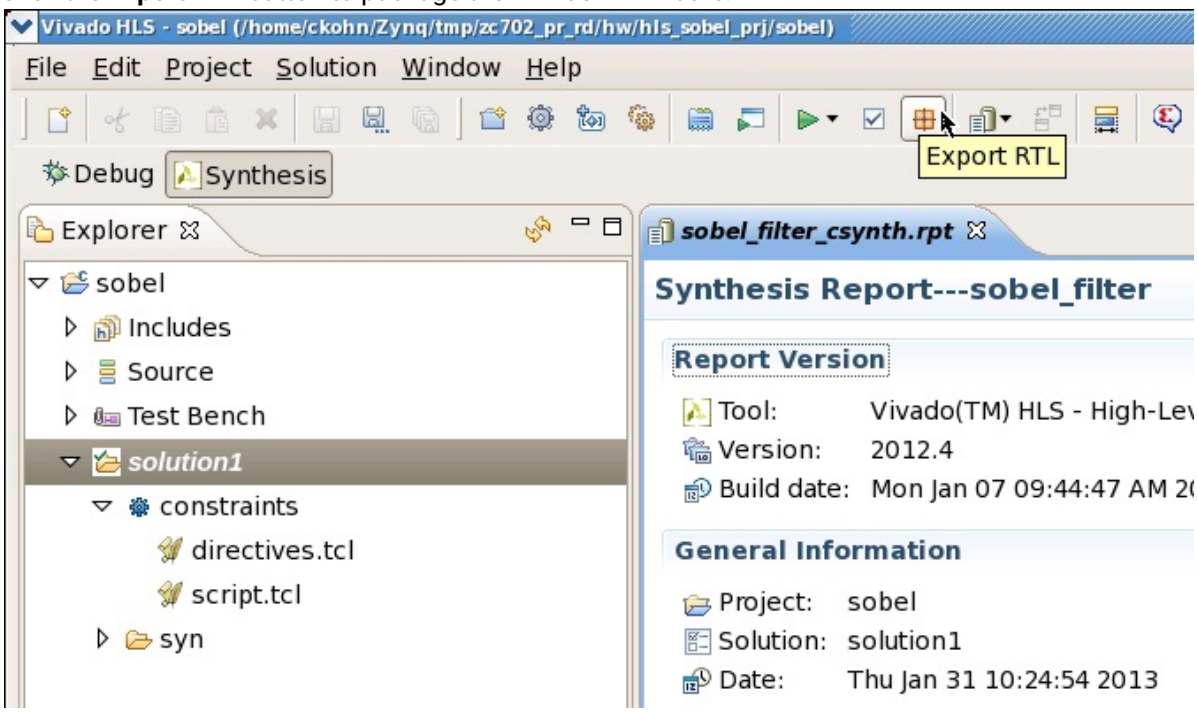
4. From the icon bar, click the **Synthesis** button to generate the RTL for the algorithm.



21.3.2 2.2 Exporting the RTL as EDK Pcore

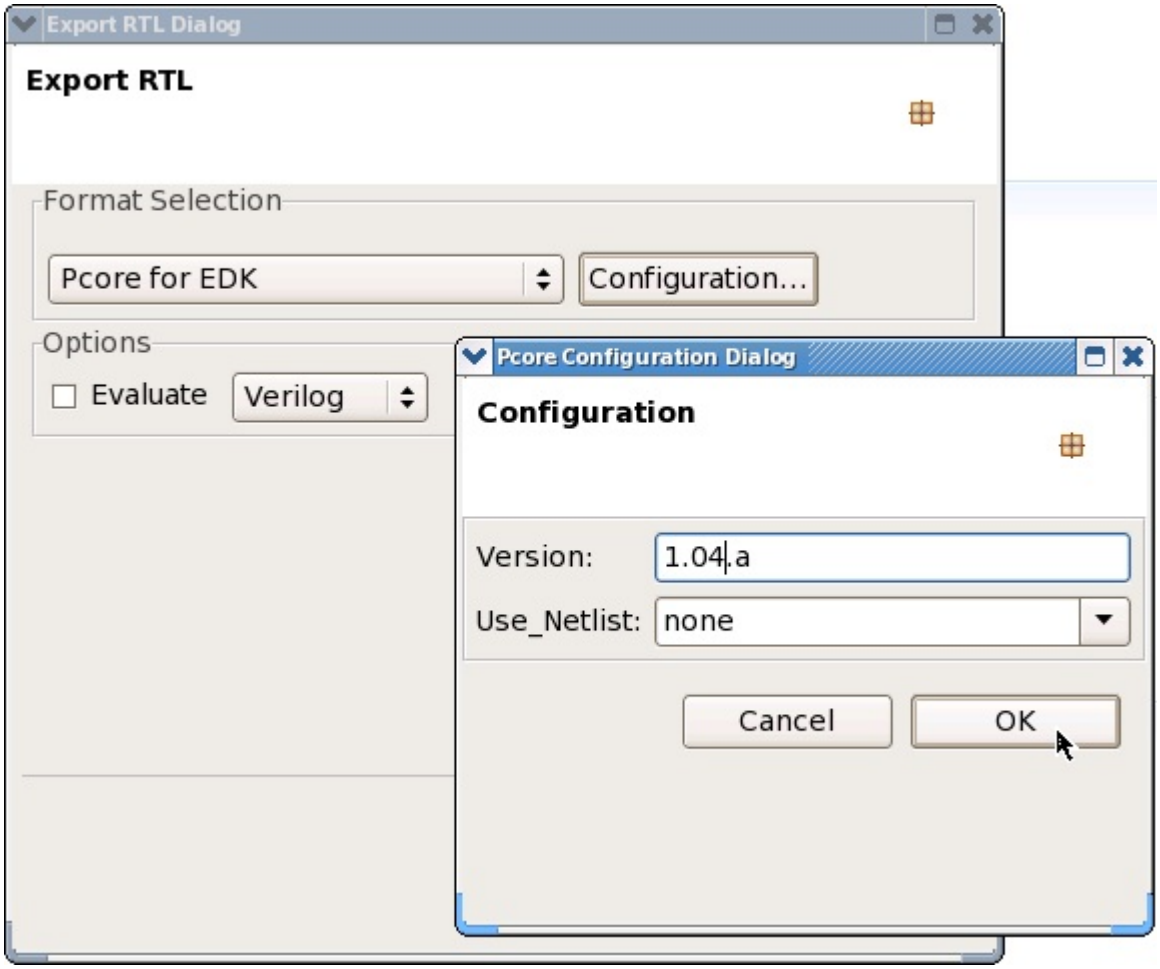
Tutorial

1. Click the **Export RTL** button to package the RTL as EDK Pcore.



2. In the dialog box, select **Pcore for EDK**.
3. Click the **Configuration...** button.

4. Enter **1.04.a** for the pcore **Version** and confirm with **OK** twice.



5. The Pcore is located in the directory zc702_pr_rd/hw/hls_sobel_prj/sobel/solution1/impl/pcores.

Repeat Sections 2.1 and 2.2 for the Sepia filter HLS project located at zc702_pr_rd/hw/hls_sepia_prj.

21.4 3 PlanAhead Base TRD Flow

The Base TRD hardware design is a PlanAhead project with an embedded XPS project. The default video filter used in the design is a Sobel filter. First you will synthesize the Base TRD with the Sobel filter using PlanAhead. Then, you will replace the Sobel filter with a Sepia filter in XPS and re-run synthesis. The generated netlists will be used as starting point for the PlanAhead PR project (see Section 4). Finally, you will export the hardware platform information so it can be imported into SDK at a later point (see Section 6.1).

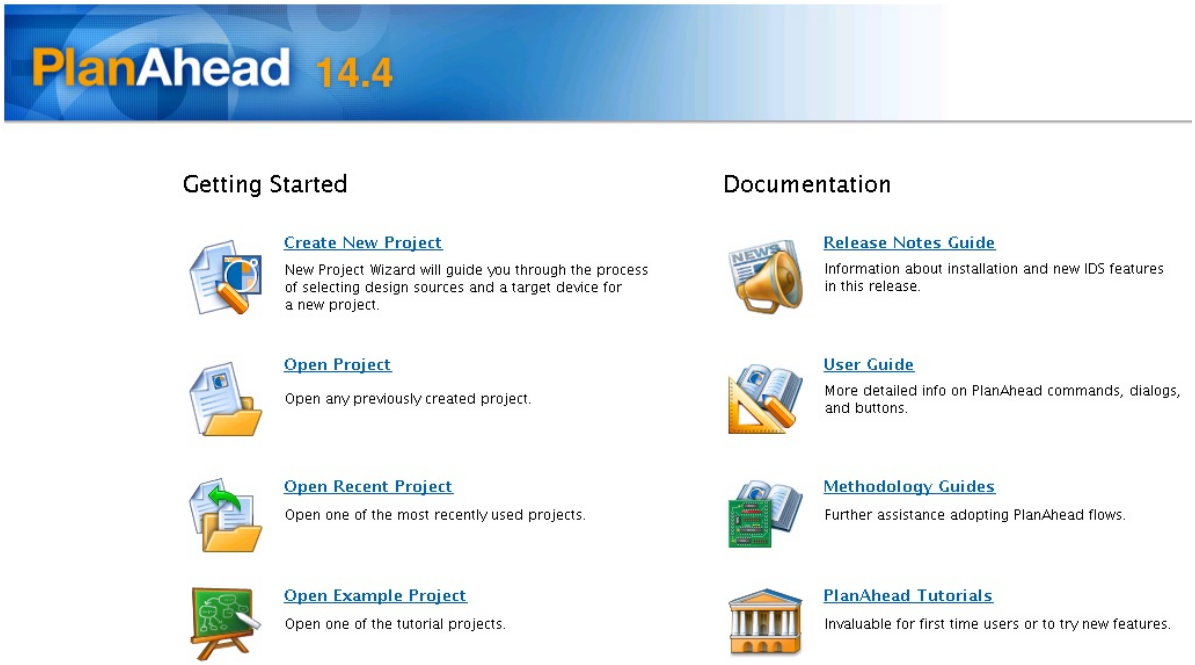
Shortcut: Pre-generated netlists are available inside the PlanAhead PR project zc702_pr_rd/hw/pr_prj. A pre-configured Hardware Platform Specification is available at zc702_pr_rd/sw/workspace/hw_platform.

21.4.1 3.1 Synthesizing the Design

Tutorial

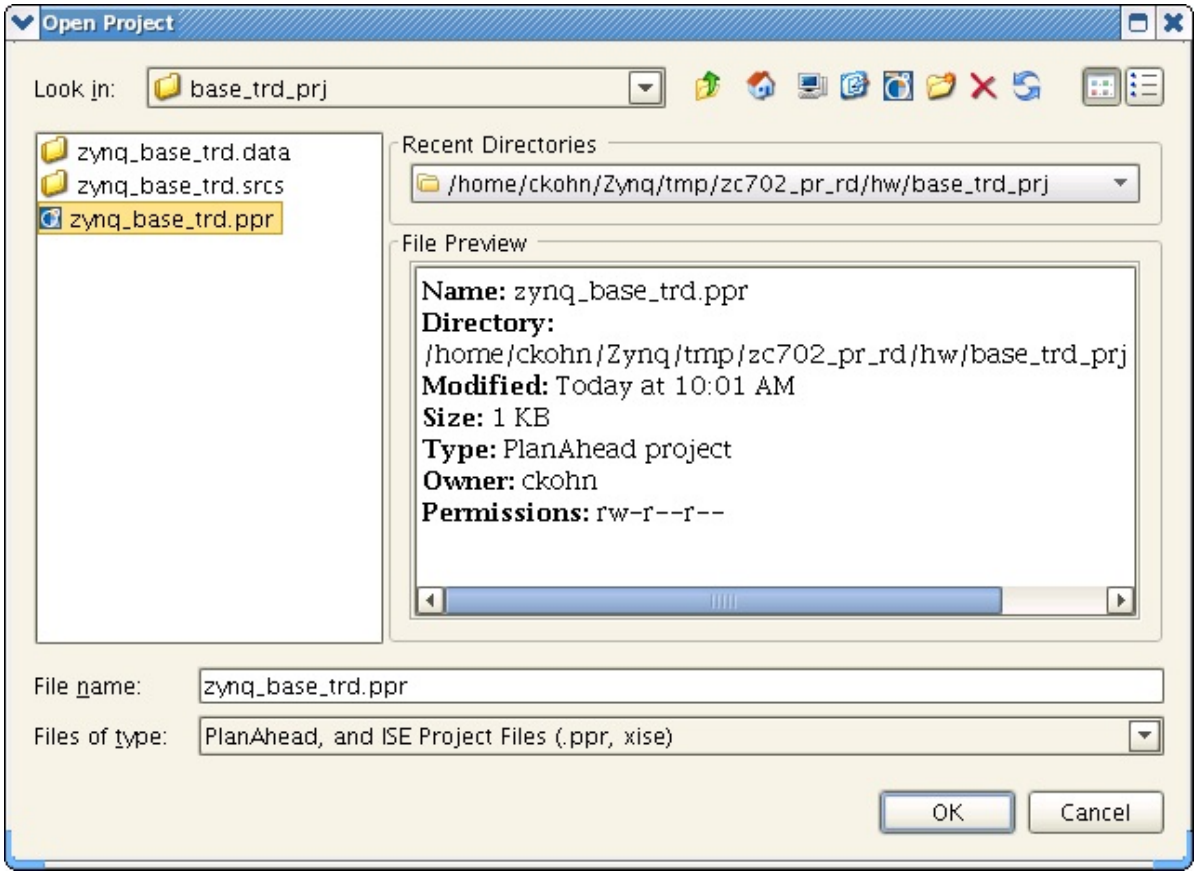
---

1. To open PlanAhead, select **Start > All Programs > Xilinx Design Tools > ISE Design Suite 14.4 > PlanAhead > PlanAhead**.
2. On the PlanAhead welcome screen, click **Open Project** under the **Getting Started** group.



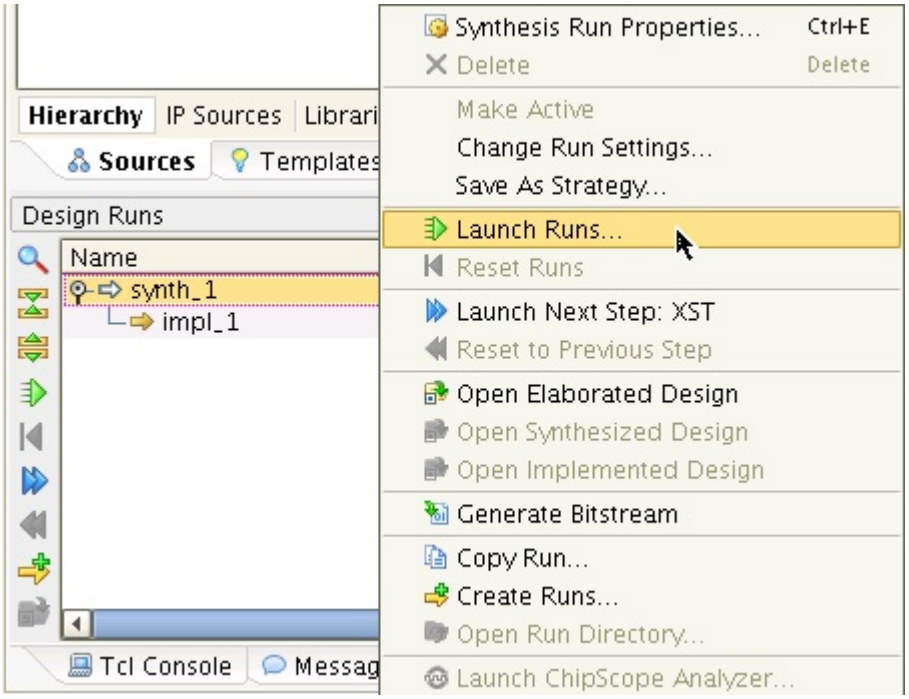
3. In the **Open Project** dialog, select the zynq_base_trd.ppr project file.

4. Click **OK**.



5. In the **Design Runs** window at the bottom, right-click **synth_1** and select **Launch Runs...**.

6. Click **OK** in **Launch Selected Runs** dialog.



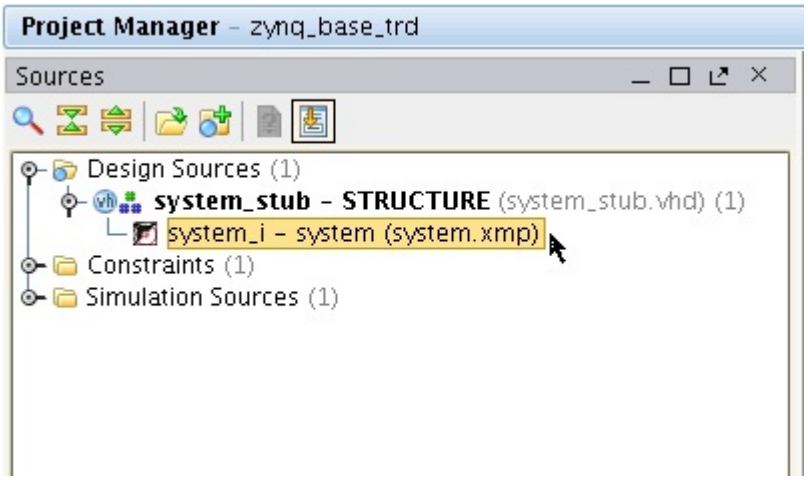
7. Click **Cancel** in the **Synthesis Completed** dialog.
8. Browse to the zc702_pr_rd/hw/base_trd_prj/zynq_base_trd.srcs/sources_1/edk/xps_proj/implementation directory and rename the file system_filter_engine_wrapper.ngc to system_filter_sobel_wrapper.ngc.

21.4.2 3.2 Replacing the Filter and Re-Synthesizing the Design

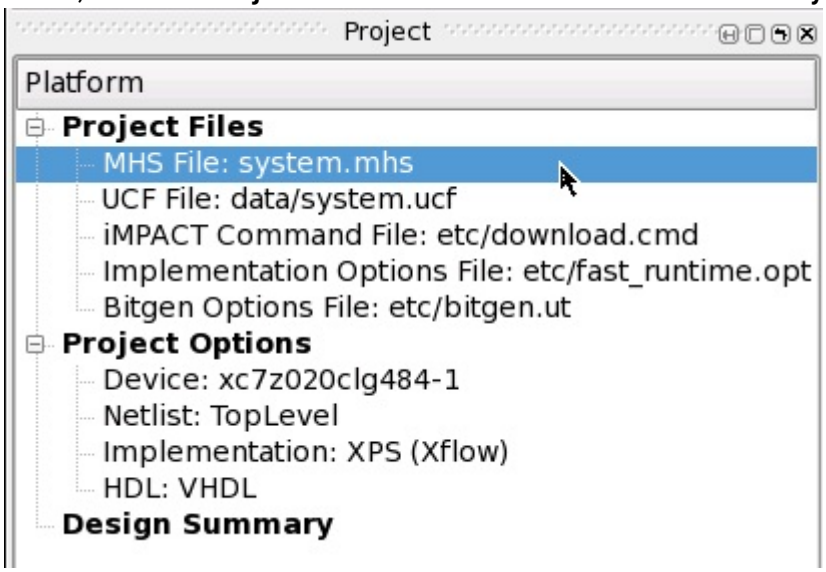
Tutorial

---

1. Double-click the **system_i - system (system.xmp)** file in the **Sources** tab of **Project Manager** to open the embedded XPS project.



2. In XPS, select the **Project** tab on the left and double-click **MHS File: system.mhs**.

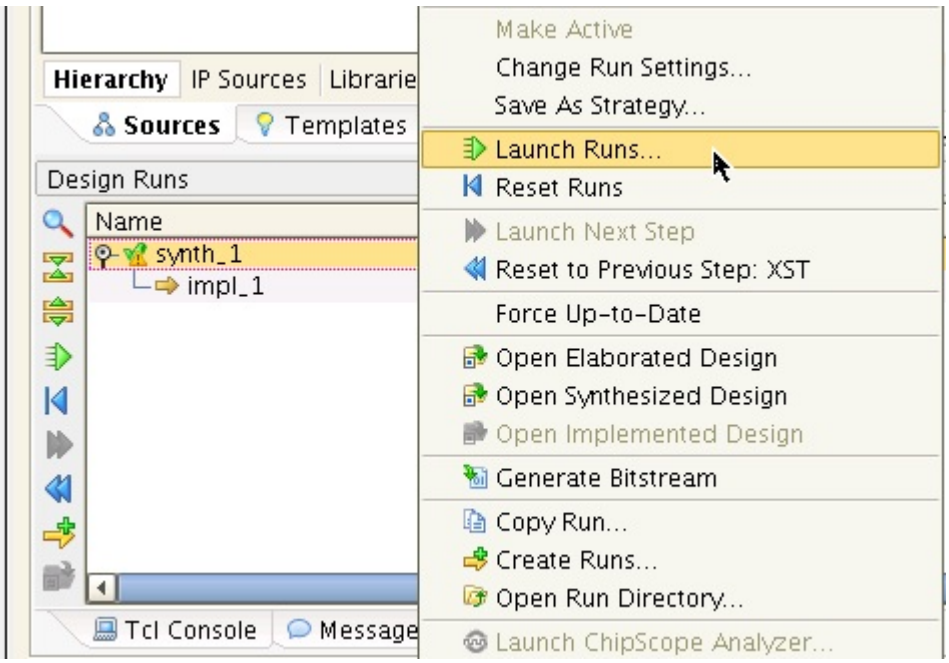


3. Scroll down or search for **sobel_filter_top** and replace with **sepia_filter_top**.

4. From the menu bar, select **File > Save** and click **Reload** when prompted to reload the project.

```
464  BEGIN sepia_filter_top
465   PARAMETER INSTANCE = FILTER_ENGINE
466   PARAMETER HW_VER = 1.04.a
467   PARAMETER C_S_AXI_CONTROL_BUS_BASEADDR = 0x400d0000
468   PARAMETER C_S_AXI_CONTROL_BUS_HIGHADDR = 0x400dffff
469   BUS_INTERFACE S_AXI_CONTROL_BUS = axi4_lite
470   BUS_INTERFACE OUTPUT_STREAM = FILTER_DMA_S2MM
471   BUS_INTERFACE INPUT_STREAM = FILTER_DMA_MM2S
472   PORT aclk = clk_150mhz
473   PORT aresetn = FILTER_RST_0
474   PORT interrupt = FILTER_0_interrupt
475  END
```

5. Close XPS and click **Yes** when prompted to continue to exit XPS.
6. Back in PlanAhead, select the **Design Runs** tab at the bottom.
   Note: The **synth_1** run has a yellow exclamation mark sign on top of a green check mark next to it, indicating that the synthesis run is completed but out-of-date. This is because we have modified the XPS project by replacing the Sobel Filter with a Sepia filter.
7. Right-click **synth_1** and select **Launch Runs...**.
8. Click **OK** twice.



9. Click **Cancel** in the **Synthesis Completed** dialog.

10. Browse to the zc702_pr_rd/hw/base_trd_prj/zynq_base_trd.srcs/sources_1/edk/xps_proj/implementation directory and rename the file system_filter_engine_wrapper.ngc to system_filter_sepia_wrapper.ngc.

21.4.3 3.3 Exporting the Hardware Platform Specification

Tutorial

---

1. From the menu bar, select **File > Export > Export Hardware for SDK...**
2. Click **OK**.

3. The generated Hardware Platform Specification is located at zc702_pr_rd/hw/base_trd_prj/zynq_base_trd.sdk/SDK/SDK_Export/hw

---

21.5 4 PlanAhead Partial Reconfiguration Design Flow

Based on the synthesized netlists from the Base TRD hardware design, you will use the PlanAhead tool to

create a reconfigurable partition, floorplan the design, add reconfigurable modules, run the implementation tools, and generate full and partial bitstreams. The promgen tool is used to convert the partial bitstreams to binary format so they can be used by the software application to configure the PL through the PCAP port.

Shortcut 1: A pre-built full bitstream (sobel configuration) is available at zc702_pr_rd/sw/boot/*/sobel.bit. Partial Sobel and Sepia bitstreams are available at zc702_pr_rd/sd/*/sobel.bin and zc702_pr_rd/sd/*/sepia.bin.

Shortcut 2: A pre-configured PlanAhead PR project is available at zc702_pr_rd/hw/pr_prj. When using this project, only the last two steps in the PR flow -- implementation and bitstream generation -- need to be run (see tutorial below).

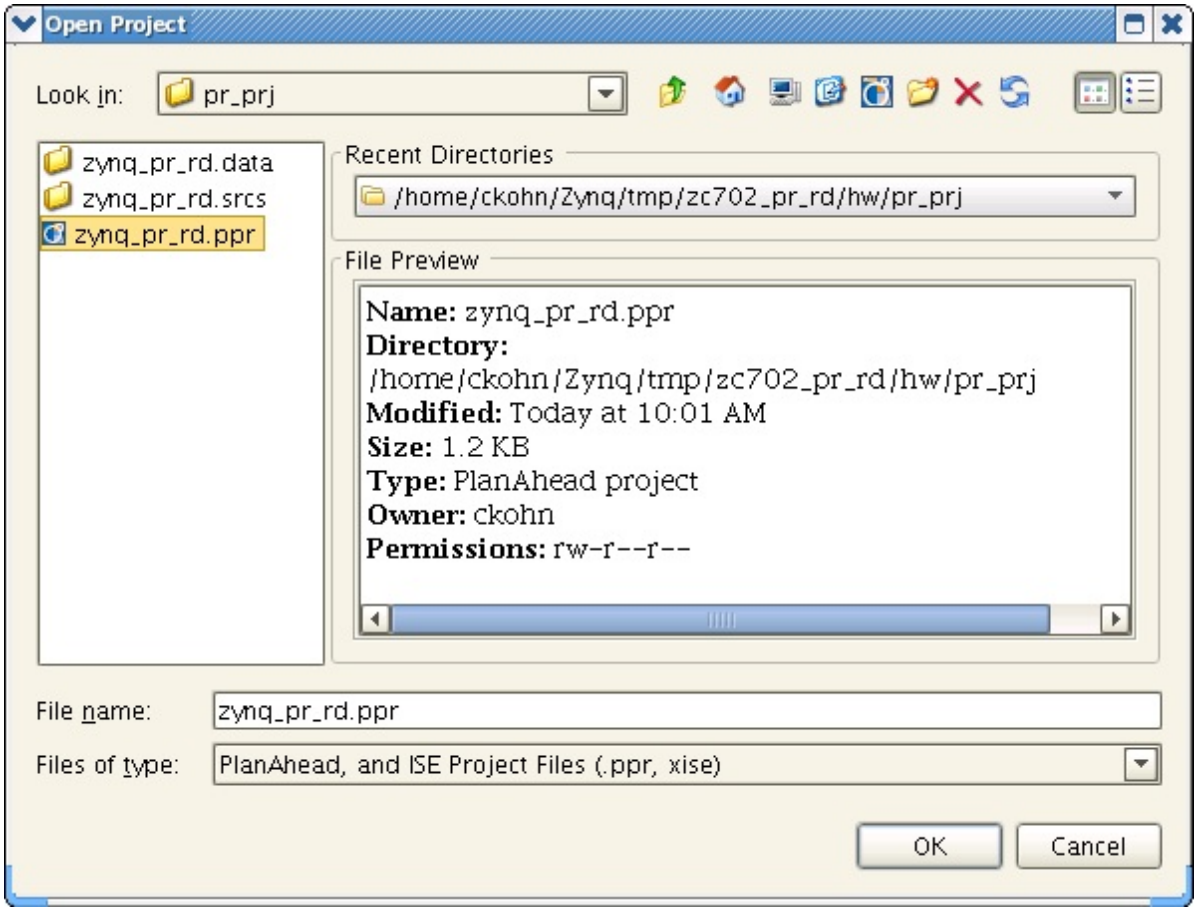
Tutorial

---

1. To open PlanAhead, select **Start > All Programs > Xilinx Design Tools > ISE Design Suite 14.4 > PlanAhead > PlanAhead**.
2. On the PlanAhead welcome screen, click **Open Project** under the **Getting Started** group.

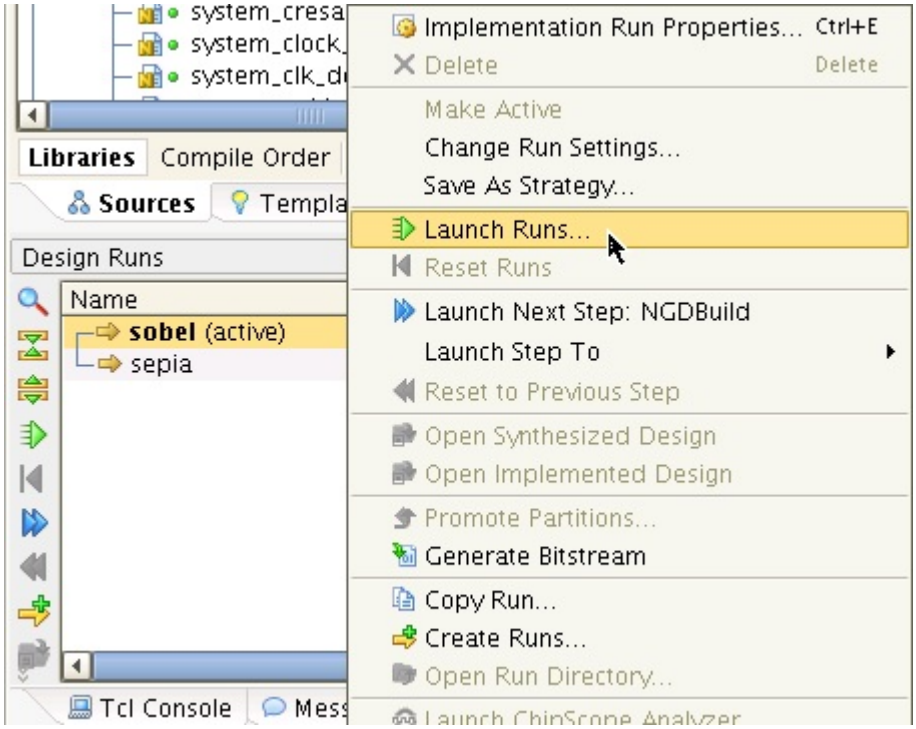
3. In the **Open Project** dialog, browse to the zc702_pr_rd/hw/pr_prj directory and select zynq_pr_rd.ppr.

4. Click **OK**.

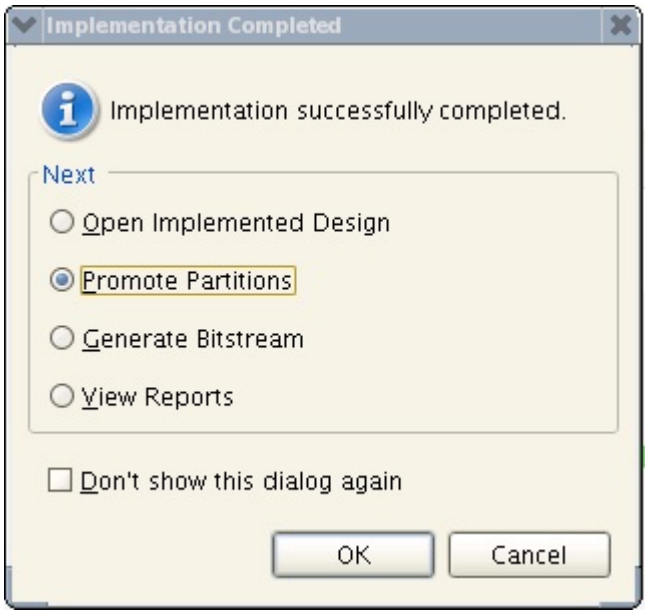


5. In the **Design Runs** window at the bottom, right-click **sobel** and select **Launch Run...**.
6. Click **OK** in the **Launch Selected Runs** dialog.

7. Click **OK** in the **Launch Runs Critical Messages** dialog to ignore the warnings.

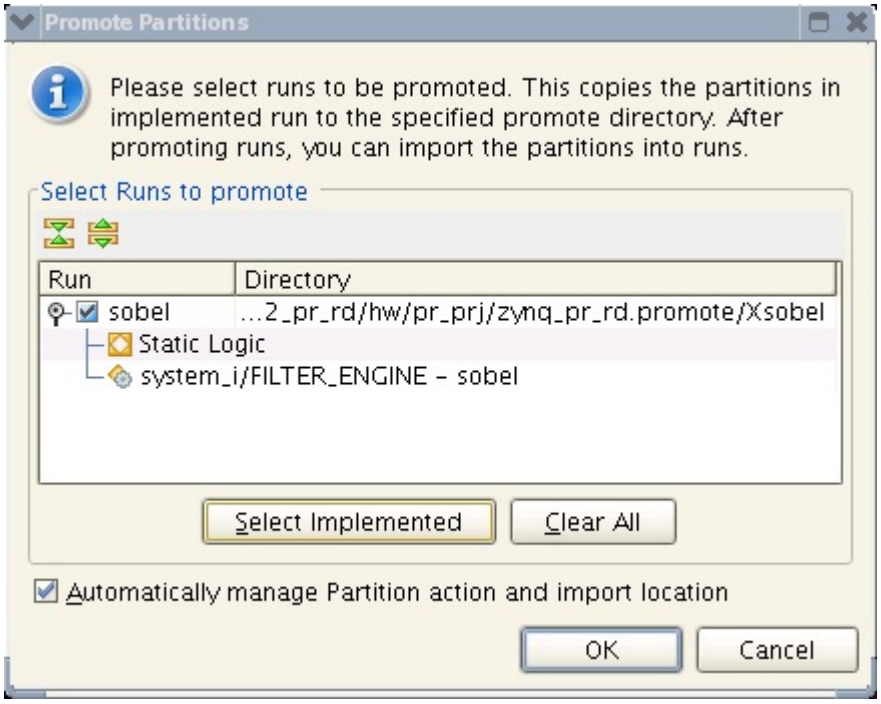


8. In the **Implementation Completed** dialog, select **Promote Partitions**.
9. Click **OK** twice.

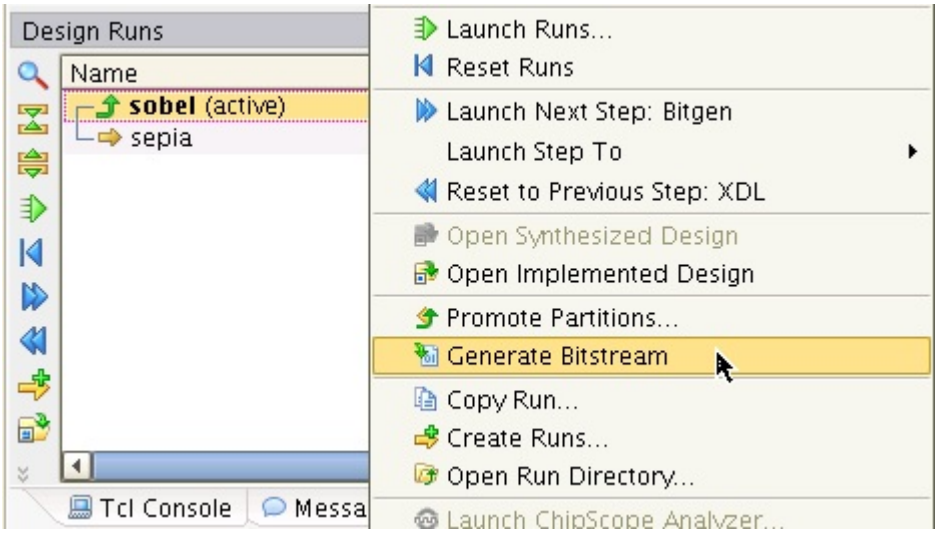


10. Click **OK** in the **Promote Partitions** dialog.

11. Click **OK** in the **Critical Messages** dialog.

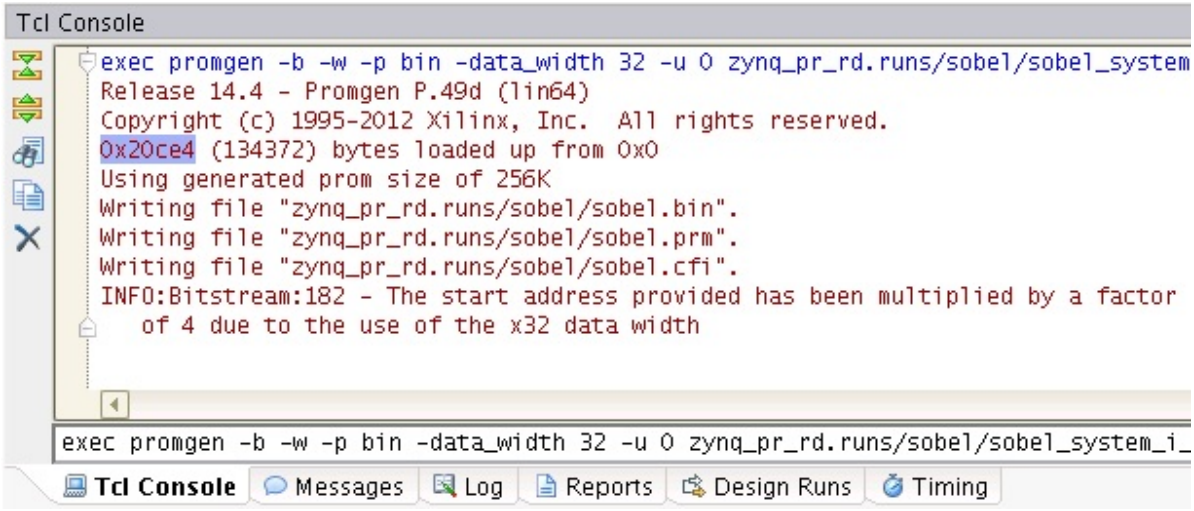


12. Select the **sobel** run, right-click and select **Generate Bitstream**.
13. Click **OK**.



14. Repeat steps 5 through 13 for the **sepia** run, except in the **Implementation Completed** dialog, click **Cancel**.
15. Select the **Tcl Console** tab at the bottom.
16. At the command line prompt, enter exec promgen -b -w -p bin -data_width 32 -u 0 zynq_pr_rd.runs/sobel/sobel_system_i_FILTER_ENGINE_sobel_partial.bit -o zynq_pr_rd.runs/sobel/sobel.bin to

convert the partial sobel bitfile to binary format.



17. Next, enter exec promgen -b -w -p bin -data_width 32 -u 0 zynq_pr_rd.runs/sepia/sepia_system_i_FILTER_ENGINE_sepia_partial.bit -o zynq_pr_rd.runs/sepia/sepia.bin to convert the partial sepia bitfile to binary format.
18. Note down the size of the generated partial binary files as indicated by the **promgen** console output -- the file size should be identical for both partial binaries.

Alternatively, complete Sections 4.1 through 4.9 for the full PR design flow tutorial.

21.5.1 4.1 Creating a PR Project and Importing the Generated Netlists

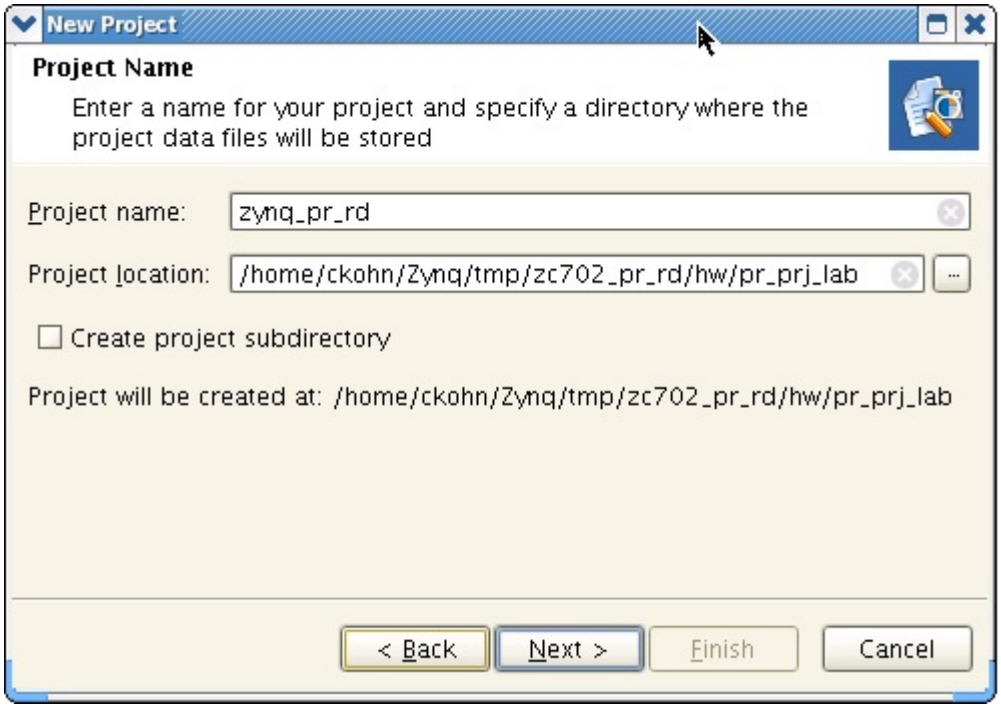
Tutorial

---

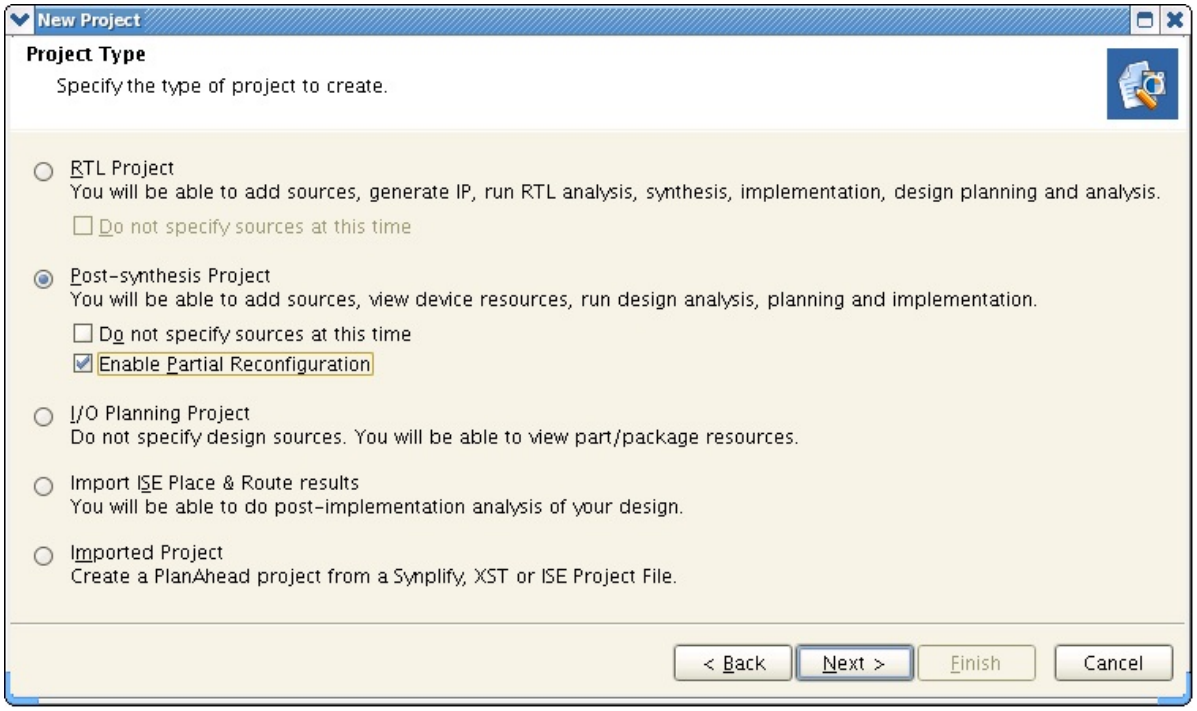
1. Create a new directory zc702_pr_rd/hw/pr_prj_lab for this tutorial.
2. To open PlanAhead, select **Start > All Programs > Xilinx Design Tools > ISE Design Suite 14.4 > PlanAhead > PlanAhead**.

3. On the PlanAhead welcome screen, click **Create New Project** under the **Getting Started** group.

4. Click **Next**.
5. In the **Project Name** dialog, change the **Project name** to **zynq_pr_rd**.
6. Browse to the zc702_pr_rd/hw/pr_prj_lab directory for **Project location** and click **Select**.
7. Click **Next**.

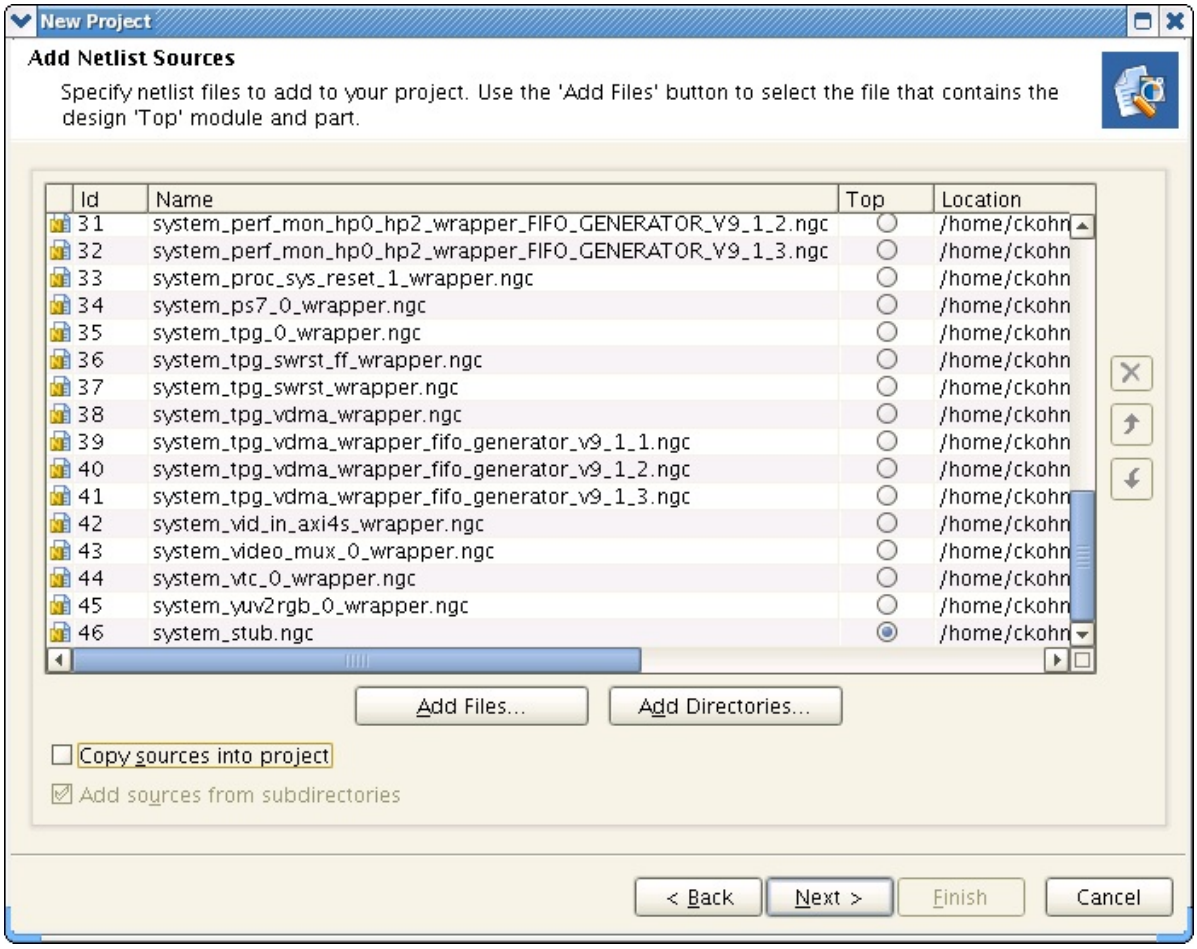


8. In the **Project Type** dialog, select **Post-synthesis Project** and check the **Enable Partial Reconfiguration** box.
9. Click **Next**.



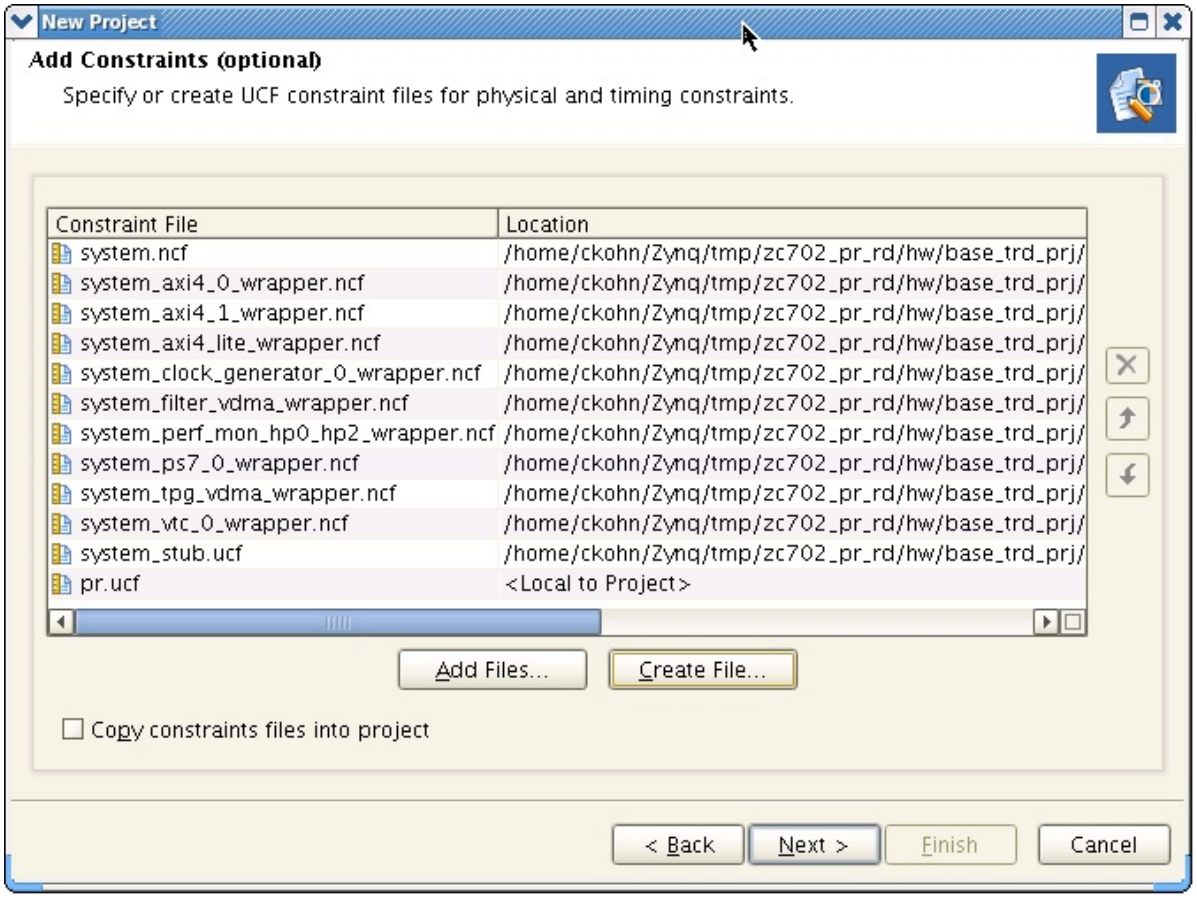
10. In the **Add Netlist Sources** dialog, click the **Add Files...** button.
11. Browse to the zc702_pr_rd/hw/base_trd_prj/zynq_base_trd.srcs/sources_1/edk/xps_proj/implementation directory.
12. Select all .ngc files except for system_filter_sepia_wrapper.ngc and system_filter_sobel_wrapper.ngc and click **OK**.
13. Click the **Add Files...** button again.
14. Browse to the zc702_pr_rd/hw/base_trd_prj/zynq_base_trd.runs/synth_1 directory.
15. Select system_stub.ngc and click **OK**.
16. Mark system_stub.ngc as top-level module by checking the radio button in the **Top** column next to the file name.

17. Click **Next**.



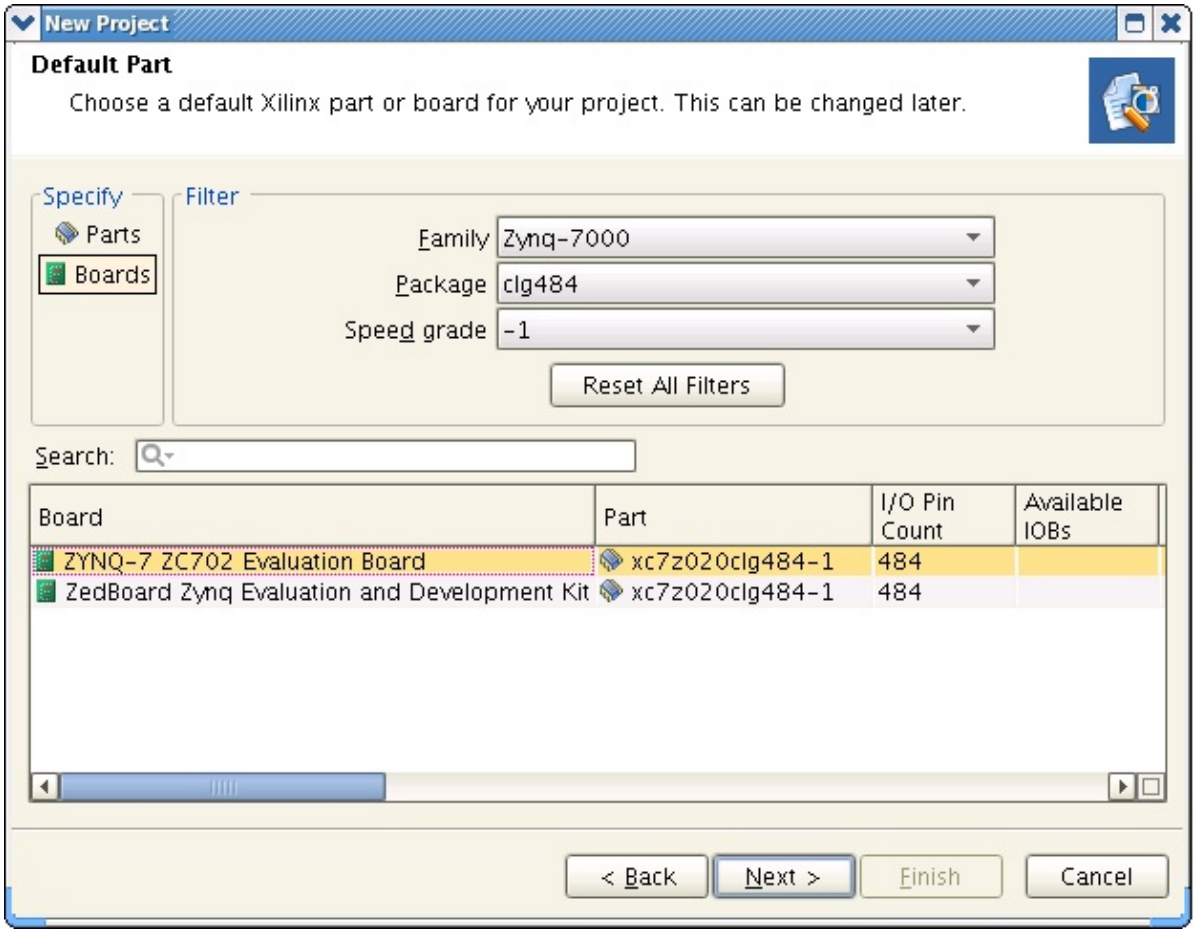
18. In the **Add Constraints (optional)** dialog, click the **Add Files...** button.
19. Browse to the zc702_pr_rd/hw/base_trd_prj/zynq_base_trd.srcs/sources_1/edk/xps_proj/implementation directory.
20. Select all .ncf files and click **OK**.
21. Click the **Add Files...** button again.
22. Browse to the zc702_pr_rd/hw/base_trd_prj/zynq_base_trd.srcs/constrs_1 directory.
23. Select system_stub.ucf and click **OK**.
24. Click the **Create File...** button.
25. Enter pr in the **File name** field and click **OK**.

26. Click **Next**.



27. In the **Default Part** dialog, select **Boards**.
28. Select **Zynq-7000** for **Family**, **CLG484** for **Package**, and **-1** for **Speed grade**.
29. Select **ZYNQ-7 ZC702 Evaluation Board** from the bottom view.
30. Click **Next**.

31. Click **Finish**.

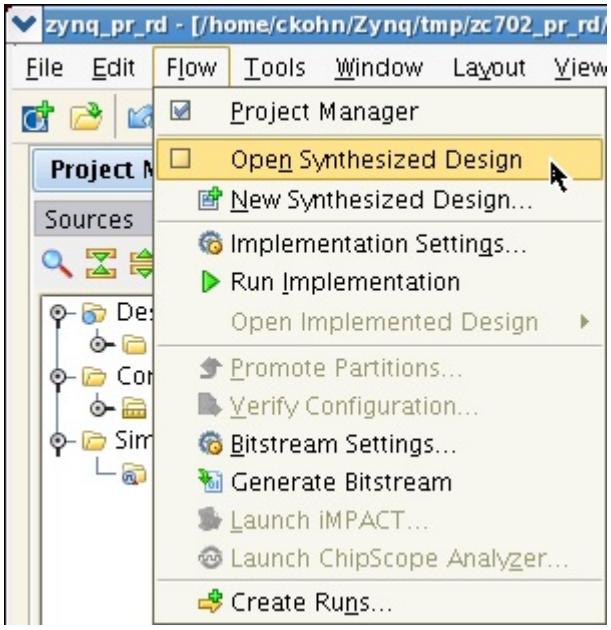


21.5.2 4.2 Defining a Reconfigurable Partition

Tutorial

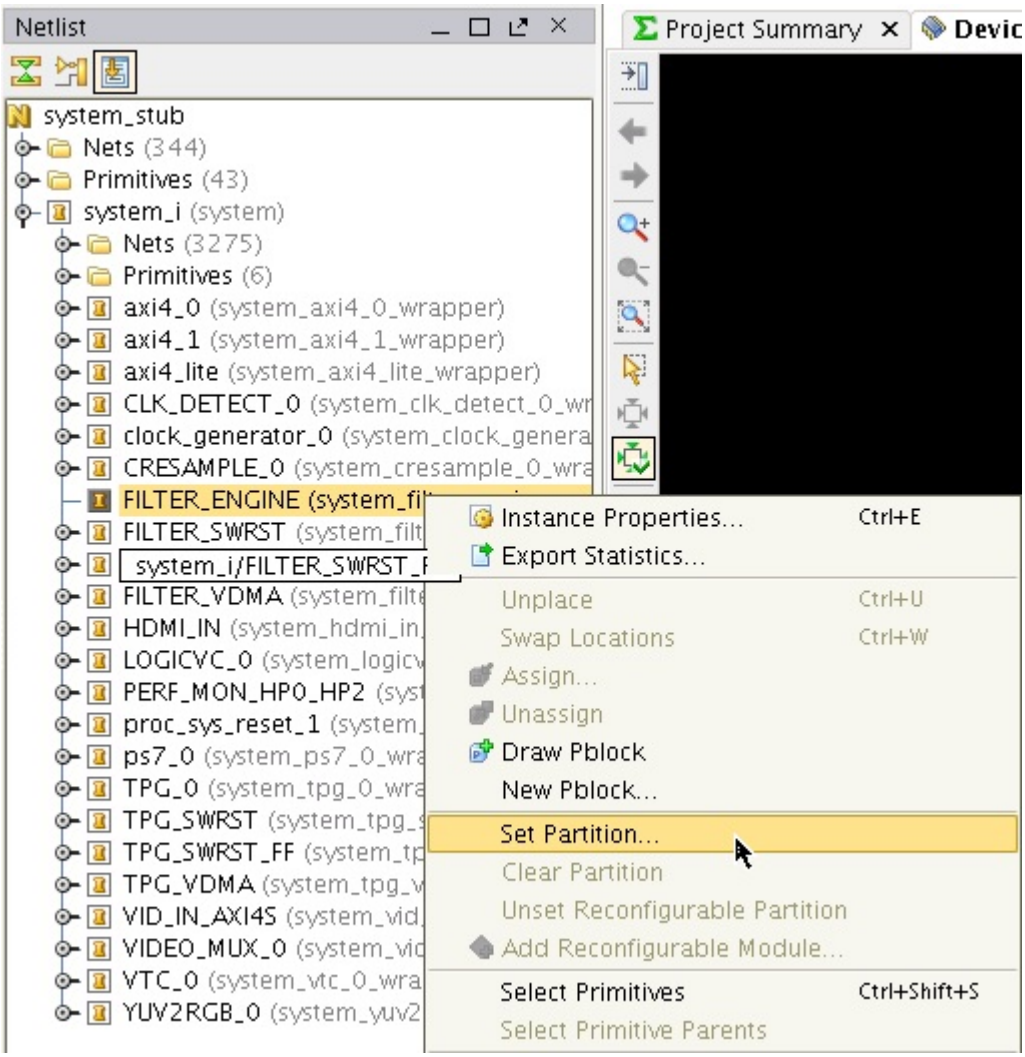
---

1. From the menu bar, select **Flow > Open Synthesized Deign**.



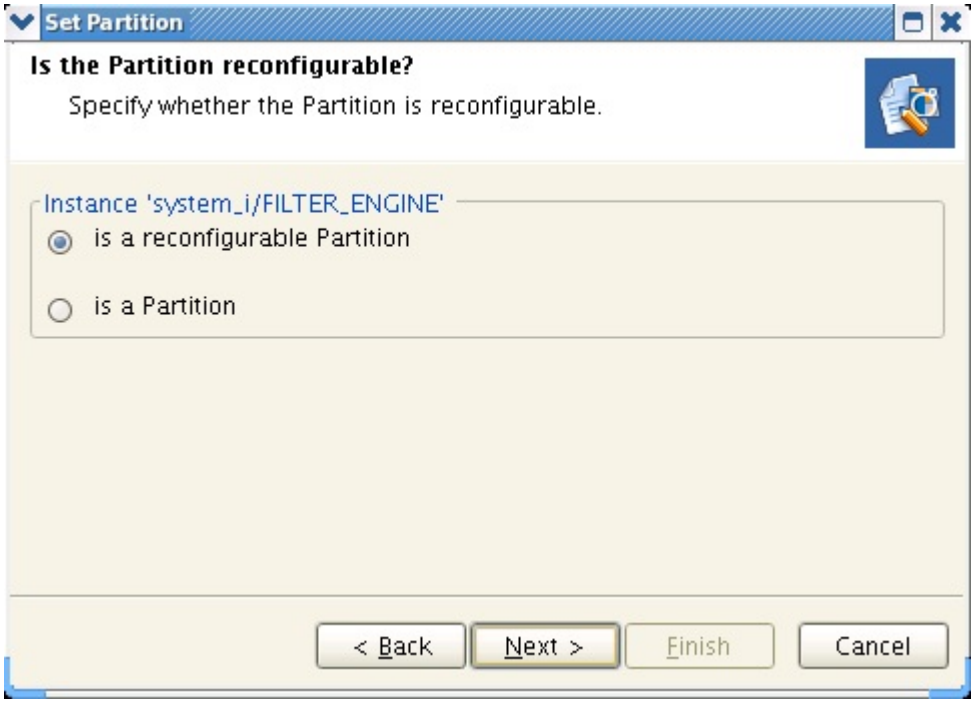
2. The **Undefined Modules Found** and the **Critical Messages** windows can be ignored. Click **OK** twice.
   Note: The missing netlist refers to the block that will be our reconfigurable module and hence marked as black box for now.
3. In the **Netlist** view of **Synthesized Design**, right-click the **FILTER_ENGINE** module and select **Set Partition...**.

4. Click **Next**.

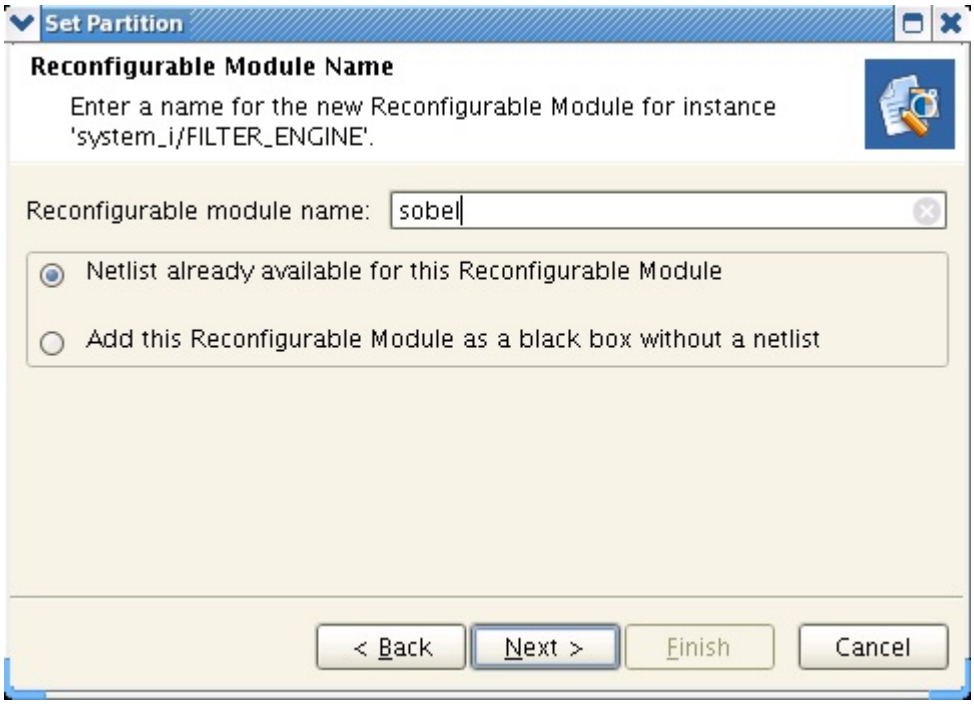


5. In the **Set Partition** dialog, select **is a reconfigurable Partition**.

6. Click **Next**.

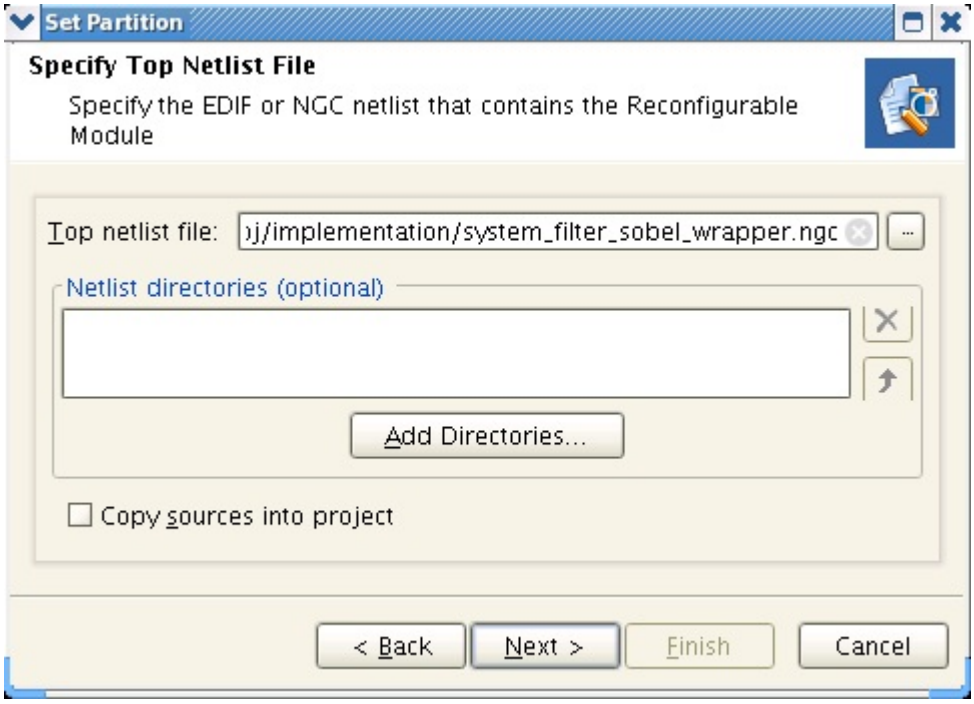


7. Enter **sobel** for **Reconfigurable Module Name** and select **Netlist already available for this Reconfigurable Module**.
8. Click **Next**.



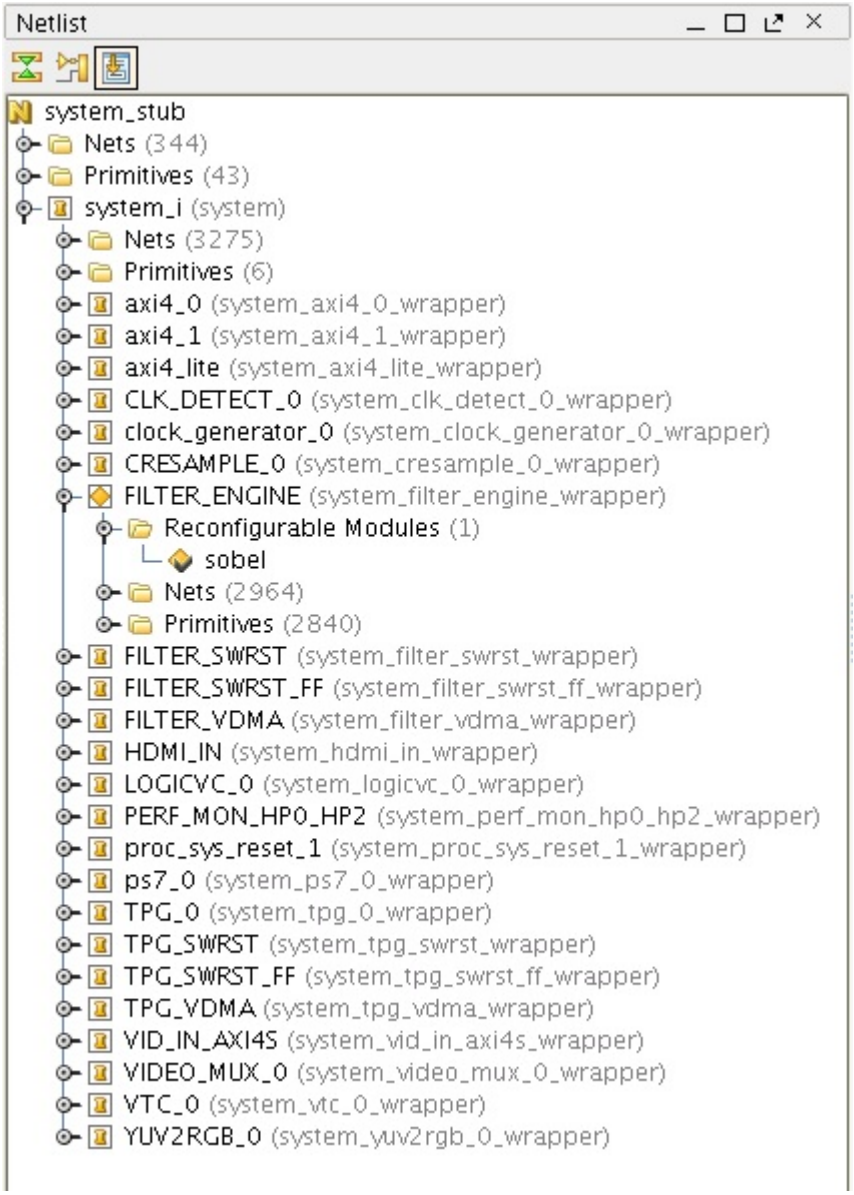
9. For **Top netlist file**, browse to the zc702_pr_rd/hw/base_trd_prj/zynq_base_trd.srcs/sources_1/edk/xps_proj/implementation directory, select the file system_filter_sobel_wrapper.ngc and click **OK**.

10. Click **Next**.



11. Skip the **Add Constraints** dialog and click **Next**.
12. Click **Finish**.
13. The **Netlist** view is updated showing a diamond next to the **FILTER_ENGINE** module and the entry **sobel** under **Reconfigurable Modules**.
    Note: The check mark inside the diamond next to **sobel** indicates that this module is the currently

active reconfigurable module.

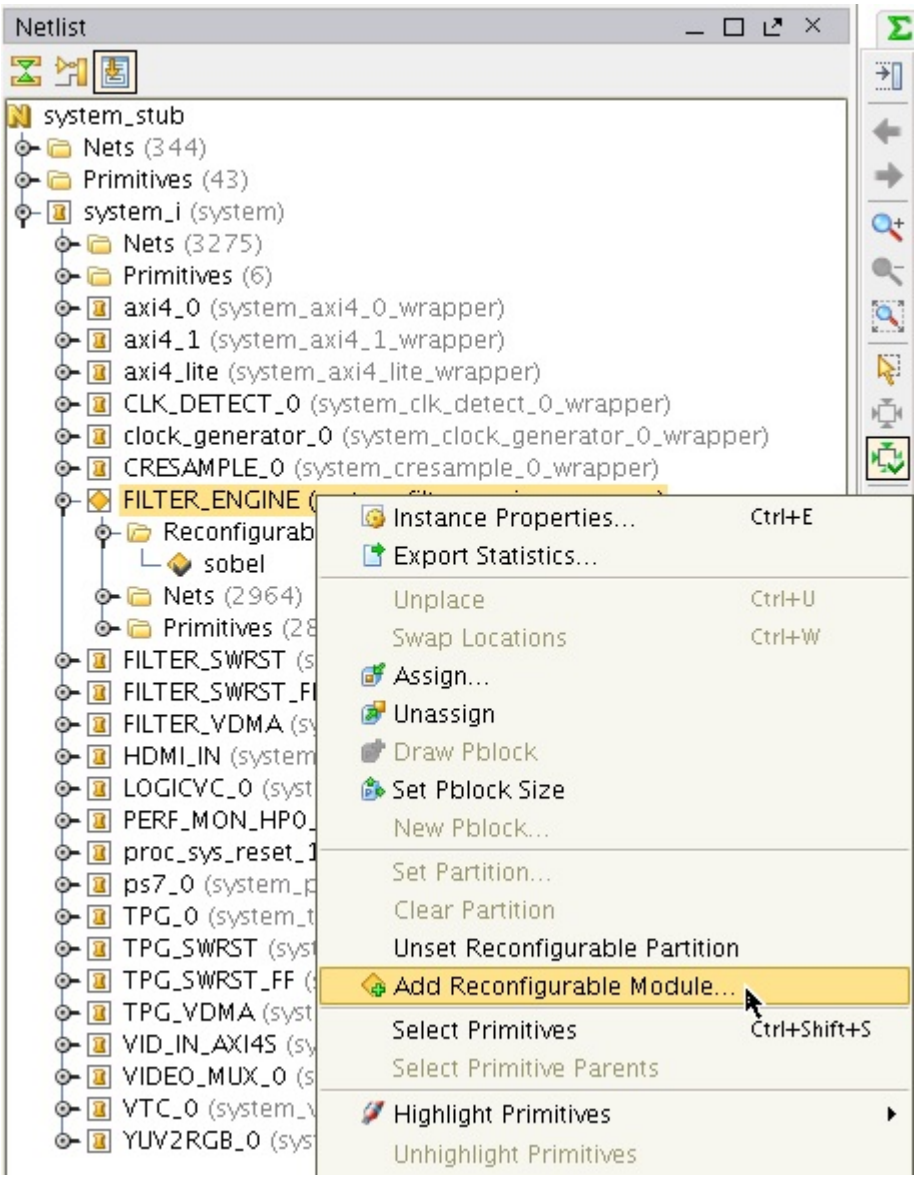


21.5.3 4.3 Adding a Reconfigurable Module

Tutorial

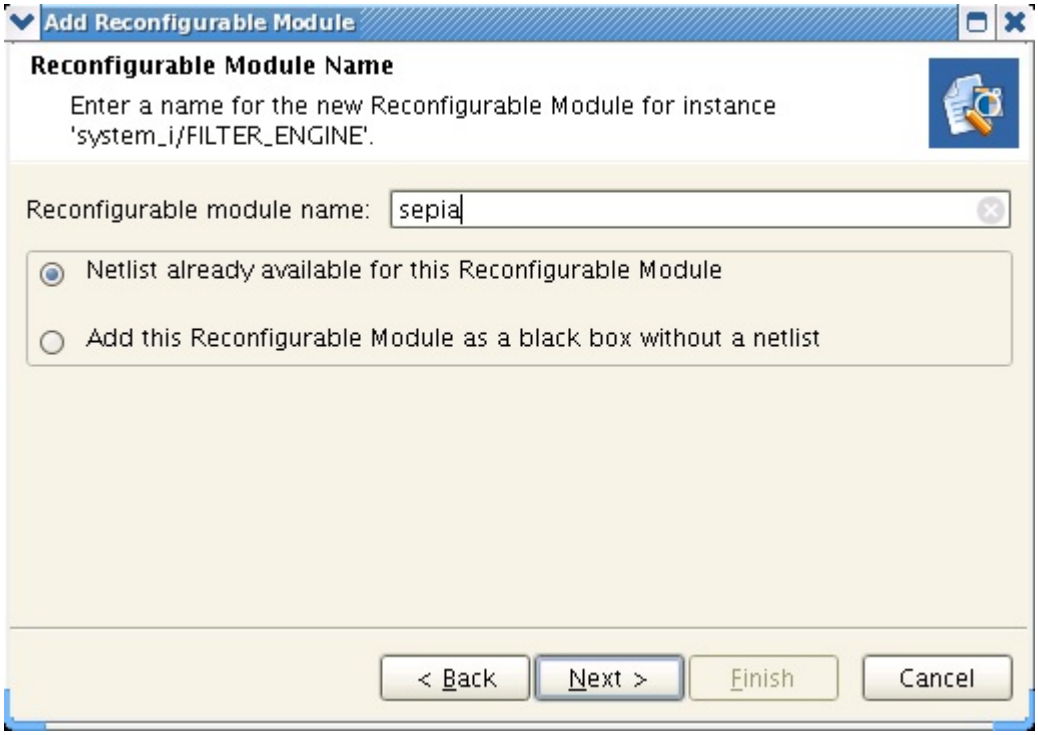
---

1. In the **Netlist** view of **Synthesized Design**, right-click the **FILTER_ENGINE** module and select **Add Reconfigurable Module...**.
2. Click **Next**.

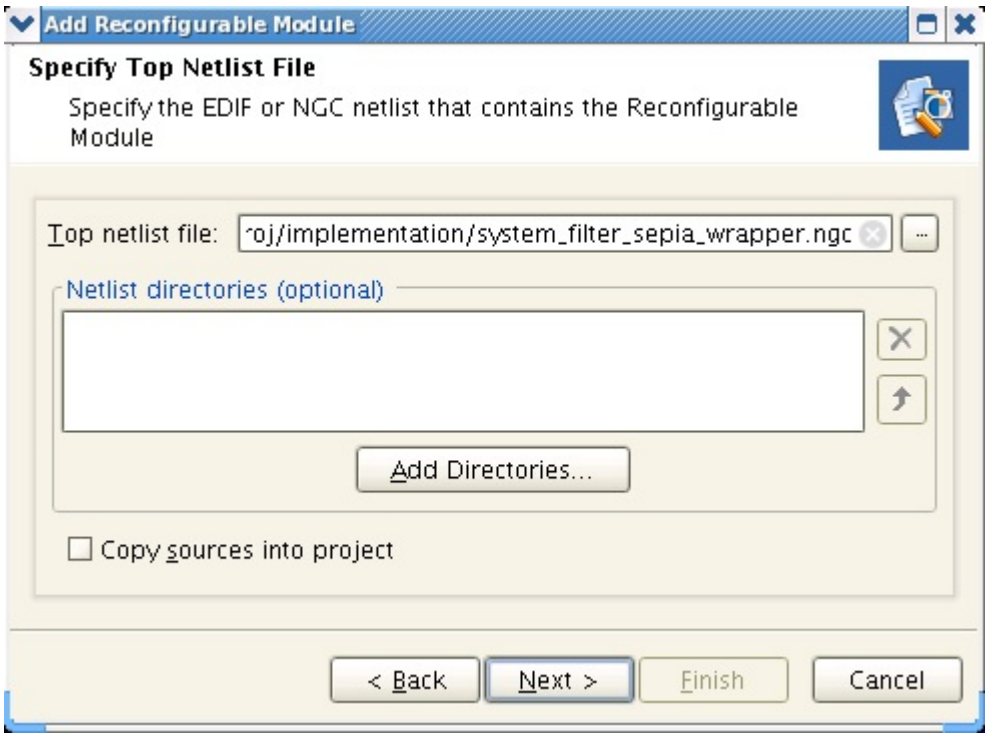


3. Enter **sepia** for **Reconfigurable Module Name** and select **Netlist already available for this Reconfigurable Module**.

4. Click **Next**.

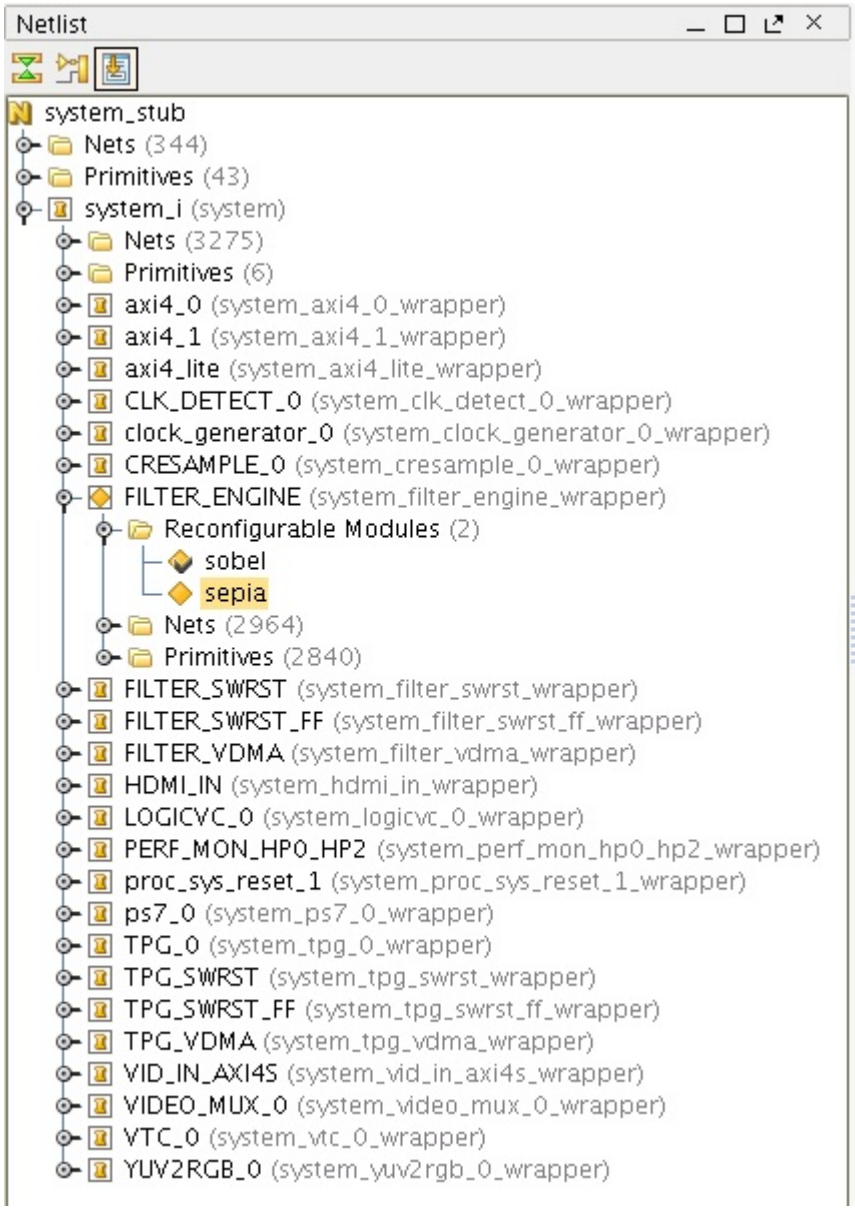


5. Browse to the zc702_pr_rd/hw/base_trd_prj/zynq_base_trd.srcs/sources_1/edk/xps_proj/implementation directory, select system_filter_sepia_wrapper.ngc and click **OK**.
6. Click **Next**.



7. Skip the **Add Constraints** dialog and click **Next**.

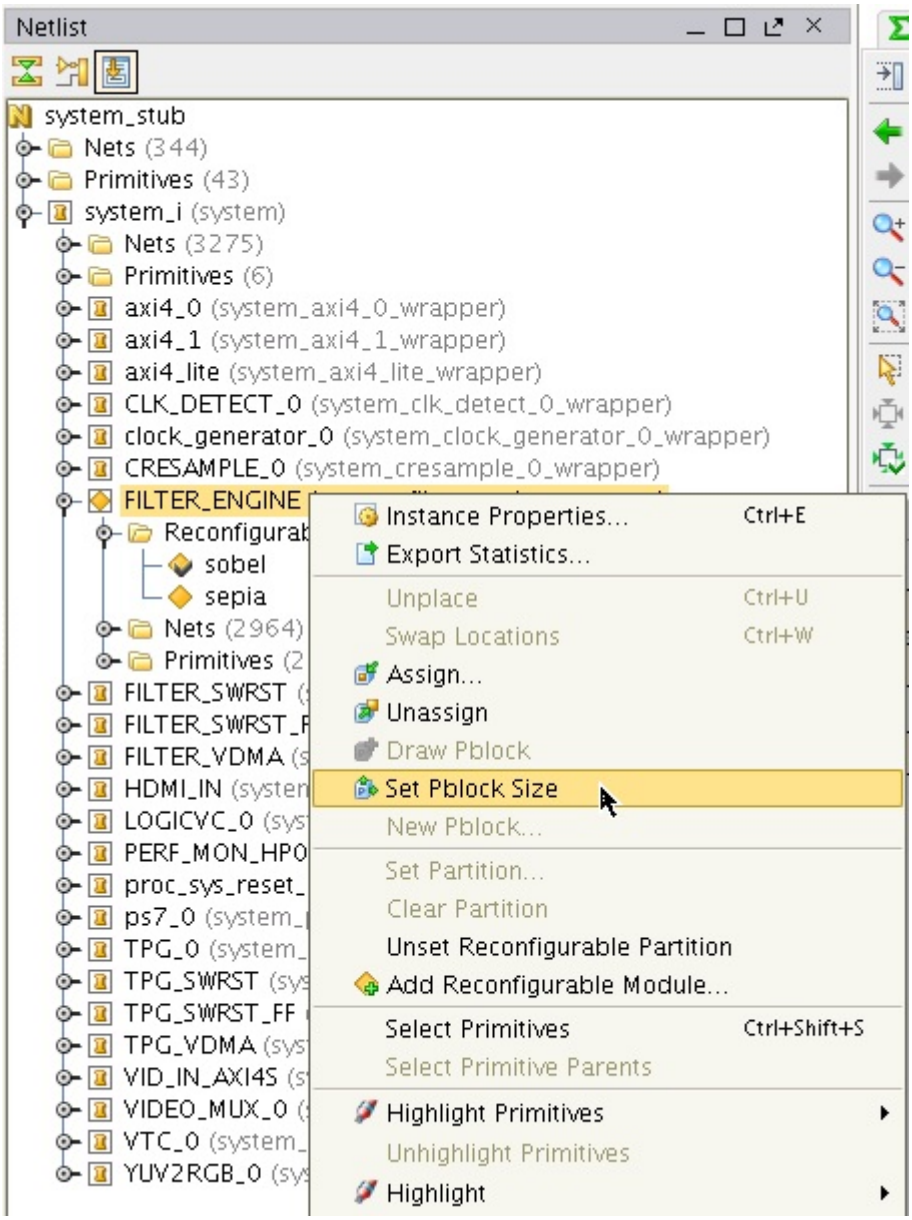
8. Click **Finish**.
9. The **Netlist** view is updated showing the entry **sepia** under **Reconfigurable Modules**.
   Note: The **sepia** module does not have a check mark inside the diamond next to it since there can be only one active reconfigurable module.



21.5.4 4.4 Floorplanning the Reconfigurable Partition

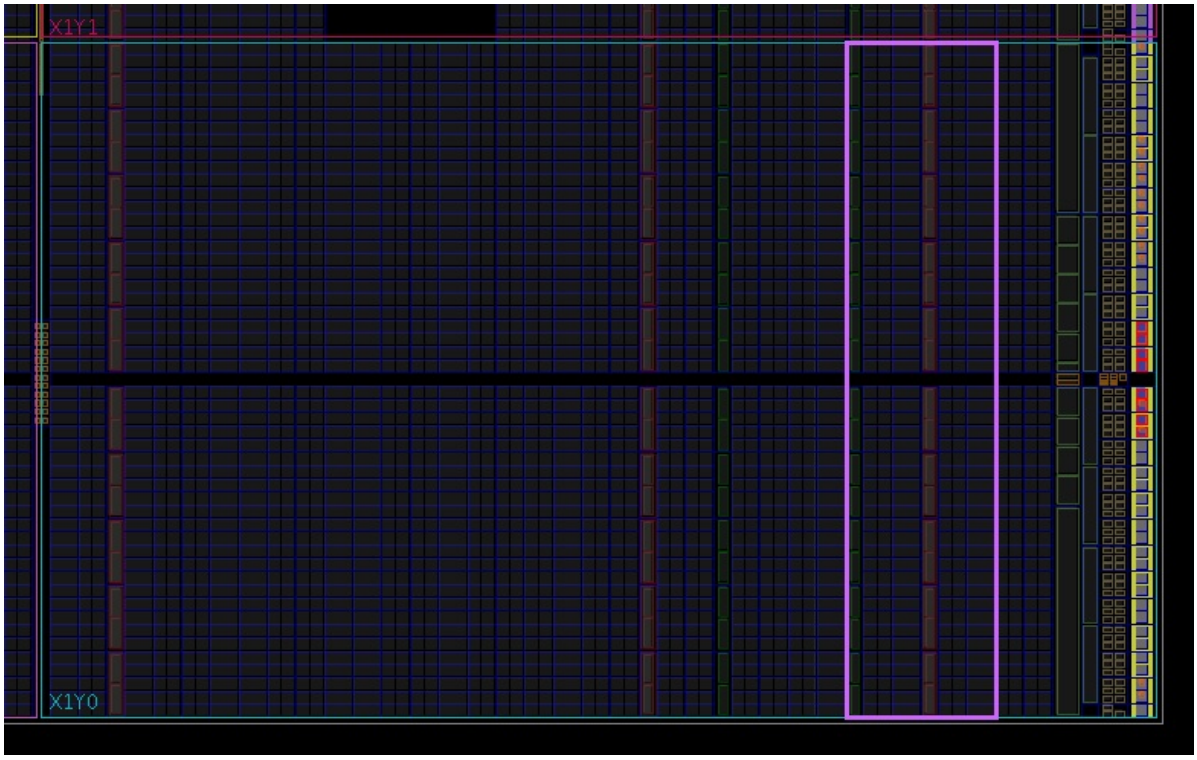
Tutorial

1. In the **Netlist** view of **Synthesized Design**, right-click the **FILTER_ENGINE** module and select **Set Pblock Size**.

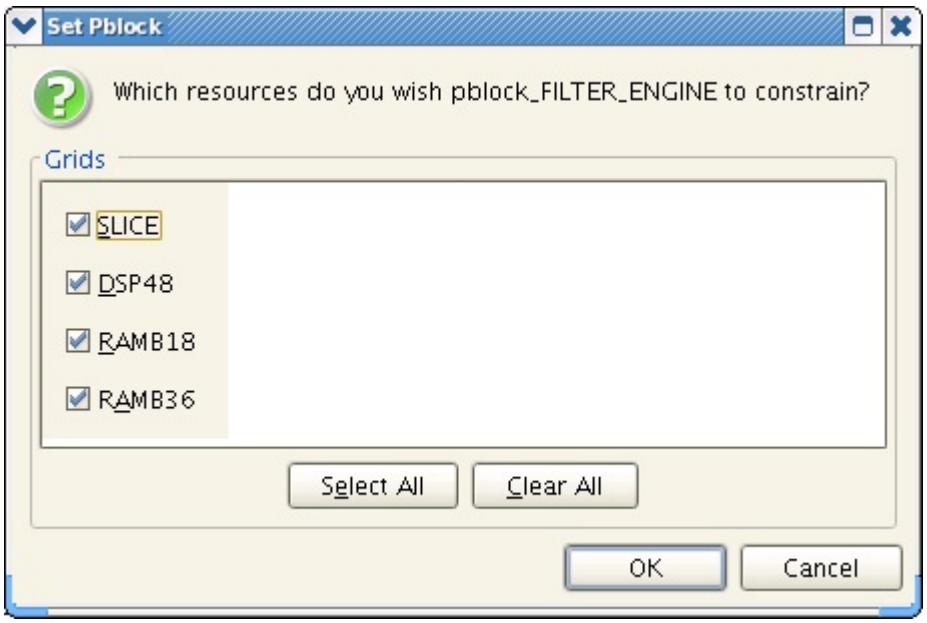


2. In the **Device** window, zoom into clock region **X1Y0** at the bottom right.
3. Click and drag the mouse cursor to draw a box that bounds **SLICE_X102Y0:SLICE_X109Y49** (bottom left to top right) as shown in the figure.
   Note: Inside the Pblock, the blue rectangles denote Slices, the column of green rectangles on the left

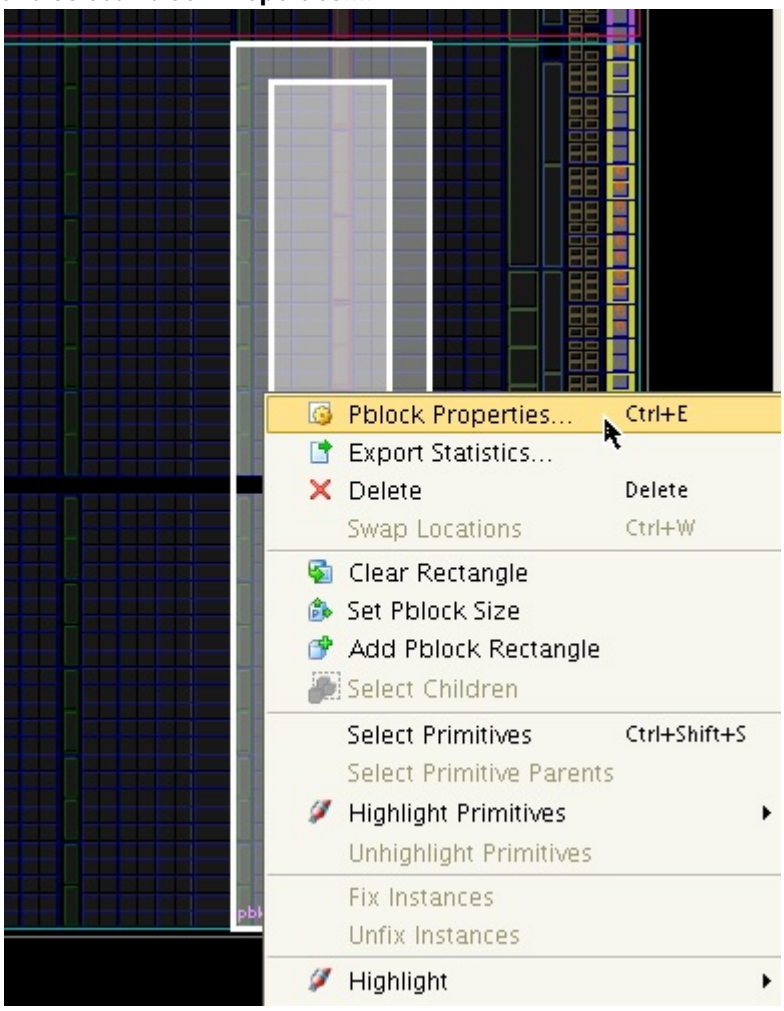
denote DSPs, the column of orange/red rectangles on the right denote BRAMs.



4. In the **Set Pblock** dialog, make sure that all of **SLICE**, **DSP48**, **RAMB18**, and **RAMB36** are checked.
5. Click **OK**.

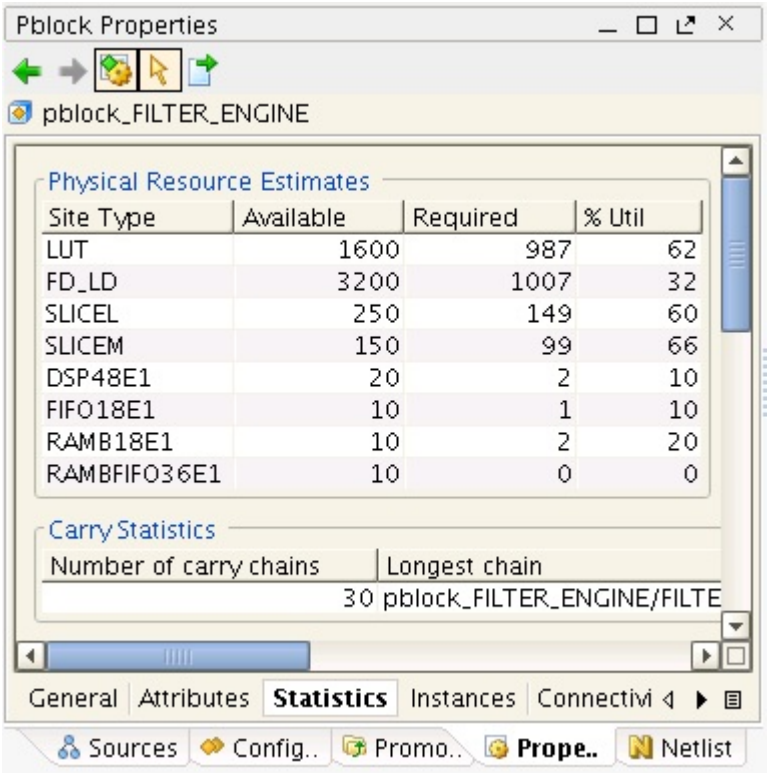


6. In the Device window, select the previously drawn Pblock such that it is highlighted, then right-click and select **Pblock Properties...**.



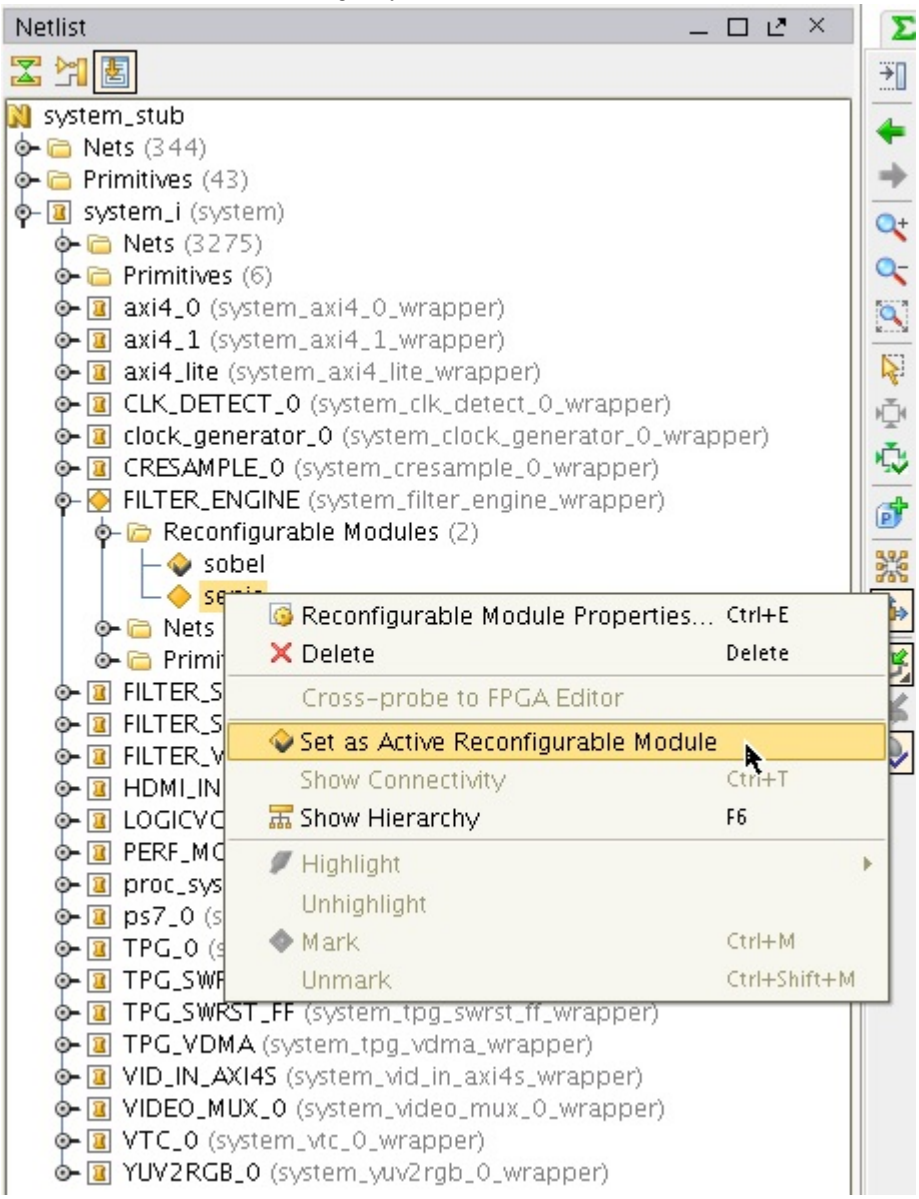
7. In the **Pblock Properties** window, select the **Statistics** tab. In the **Physical Resource Estimates** section, check that the utilization for the various site types is lower than 100%. We have now verified

that the drawn Pblock is big enough to hold the **sobel** reconfigurable module.



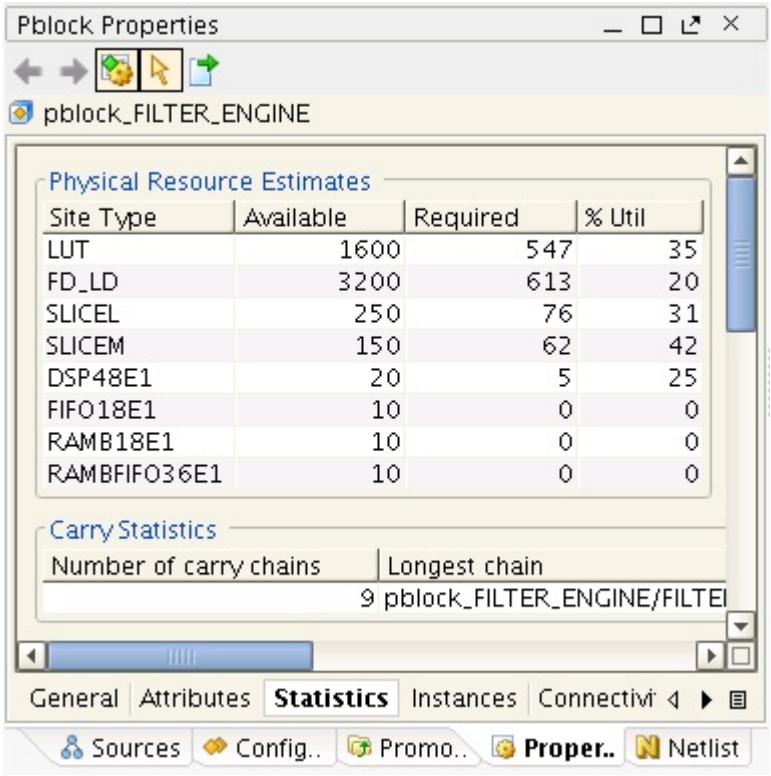
8. In the **Netlist** view of **Synthesized Design**, right-click the **sepia** reconfigurable module and select **Set as Active Reconfigurable Module**.

9. Click **Save** when prompted to save the **Synthesized Design**.
   Note: This will save the area group constraints for the created Pblock to the constraints file.



10. Repeat steps 5 and 6 to pull up the **Pblock Properties** window and check the utilization for the **sepia** reconfigurable module. How does it compare to the **sobel** reconfigurable module in terms of required

LUTs, DSPs, and BRAMs?



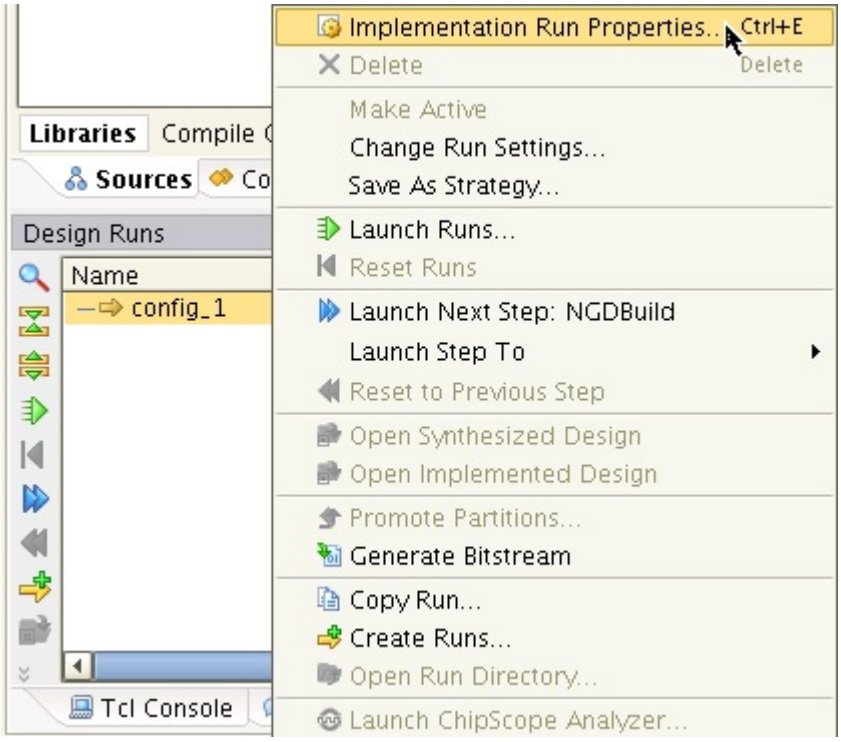
21.5.5 4.5 Creating, Implementing, and Promoting the Sobel Configuration

Tutorial

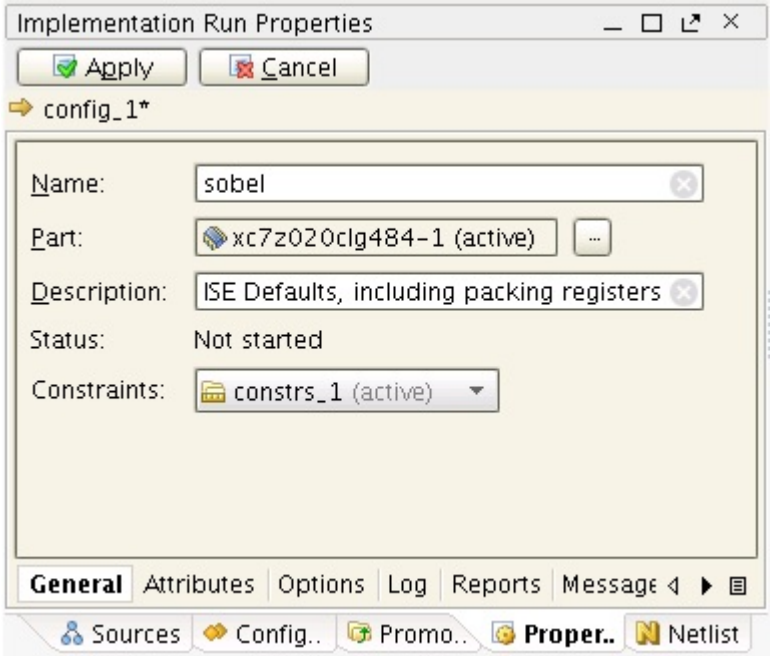
---

1. Select the **Design Runs** tab at the bottom of the screen.

2. Right-click **config_1** and select **Implementation Run Properties...**.

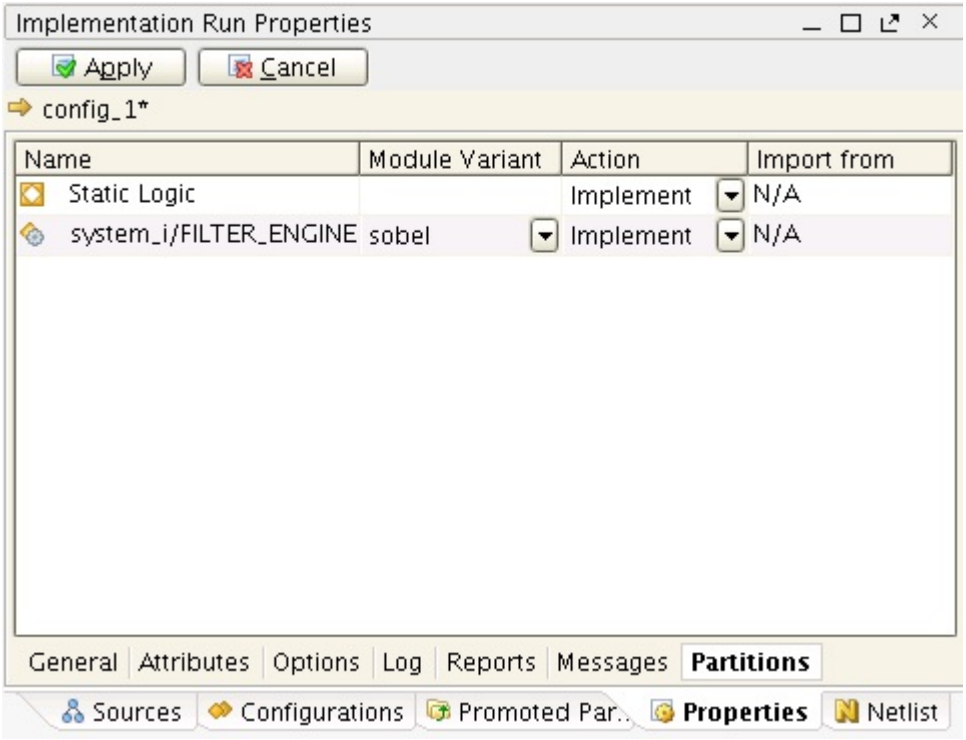


3. In the **Properties** window, select the **General** tab and change **Name** to **sobel**.

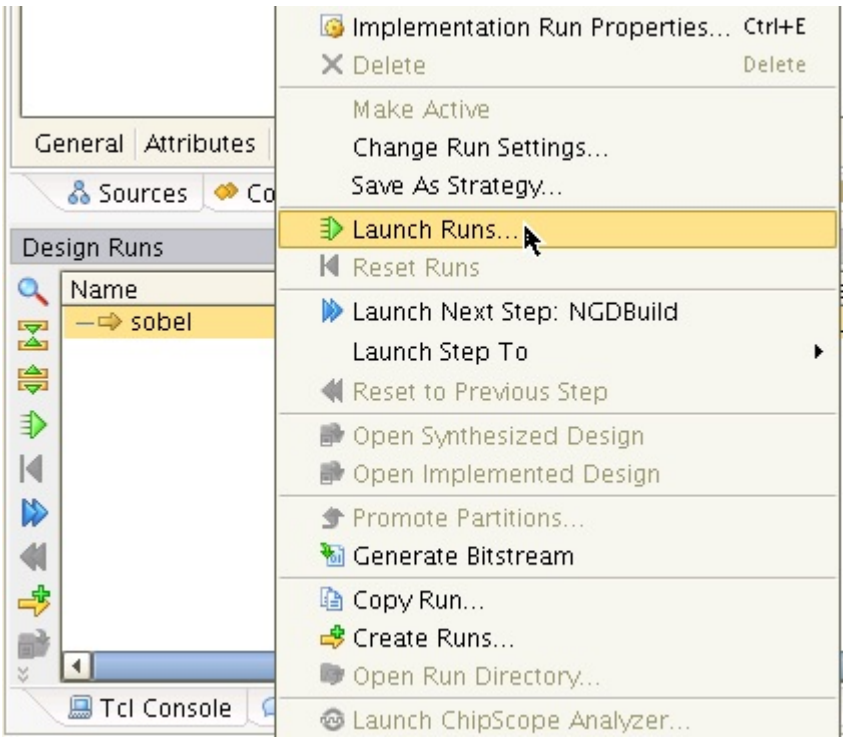


4. In the **Partitions** tab, make sure that **sobel** is selected under **Module Variant** and **Implement** under **Action**.

5. Click **Apply** to save the changes.

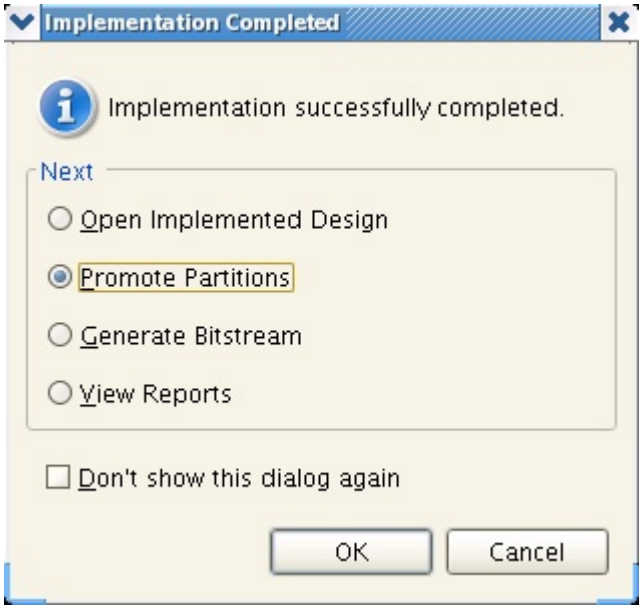


6. Back in the **Design Runs** tab at the bottom, right-click **sobel** and select **Launch Runs...**
7. Click **OK**.



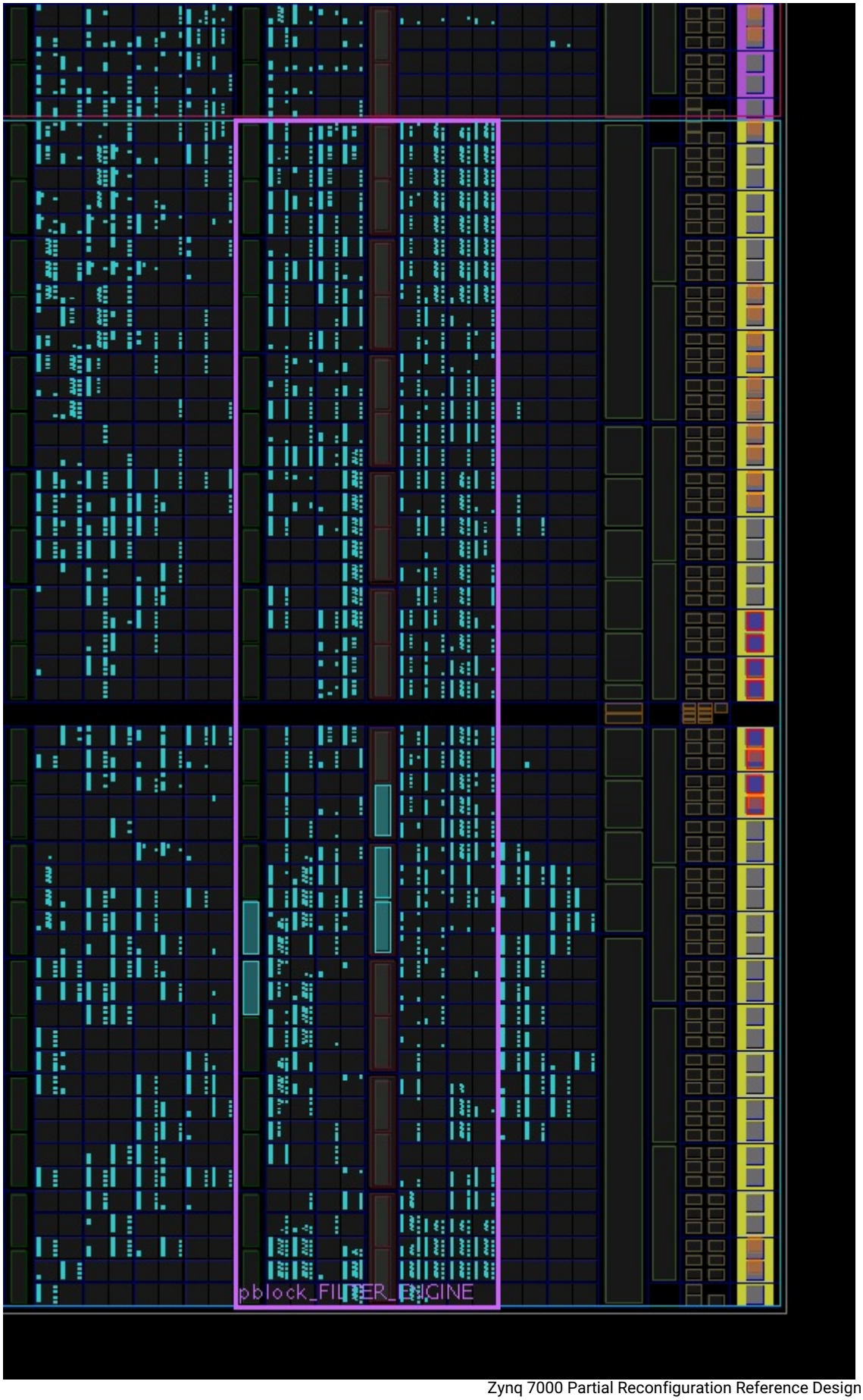
8. Click **OK** in the **Launch Run Critical Messages** dialog to ignore the warnings.

9. In the **Implementation Completed** dialog, select **Promote Partitions**.
10. Click **OK** twice.



11. Optional: To open the implemented design, select **Flow > Open Implemented Design > sobel** from the menu bar and examine the routed design in the **Device** window. How does it compare to the post-

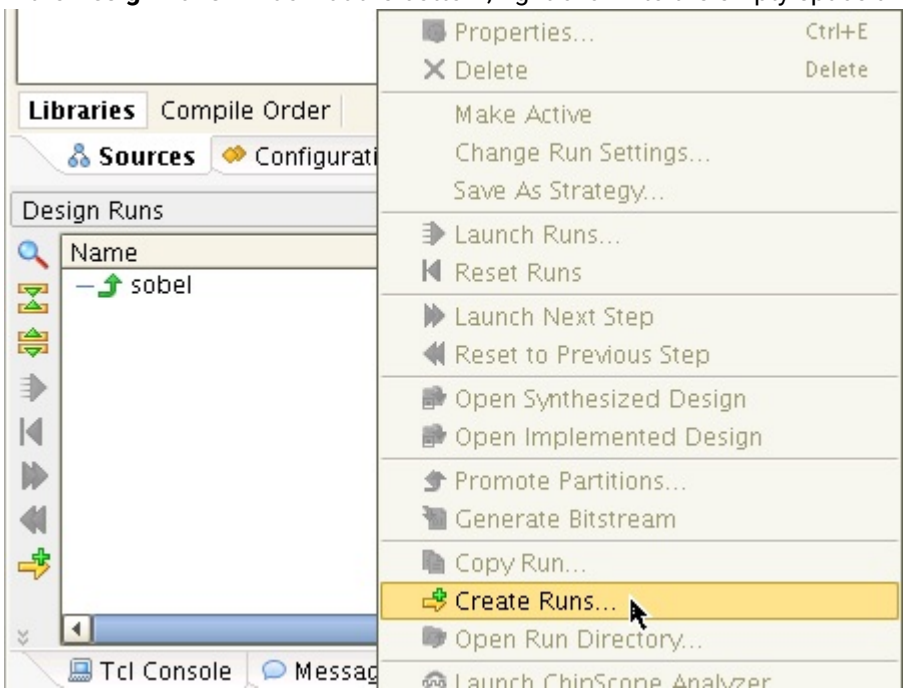
synthesis physical resources estimates?


pblock_FILTER_ENGINE

21.5.6 4.6 Creating and Implementing the Sepia Configuration

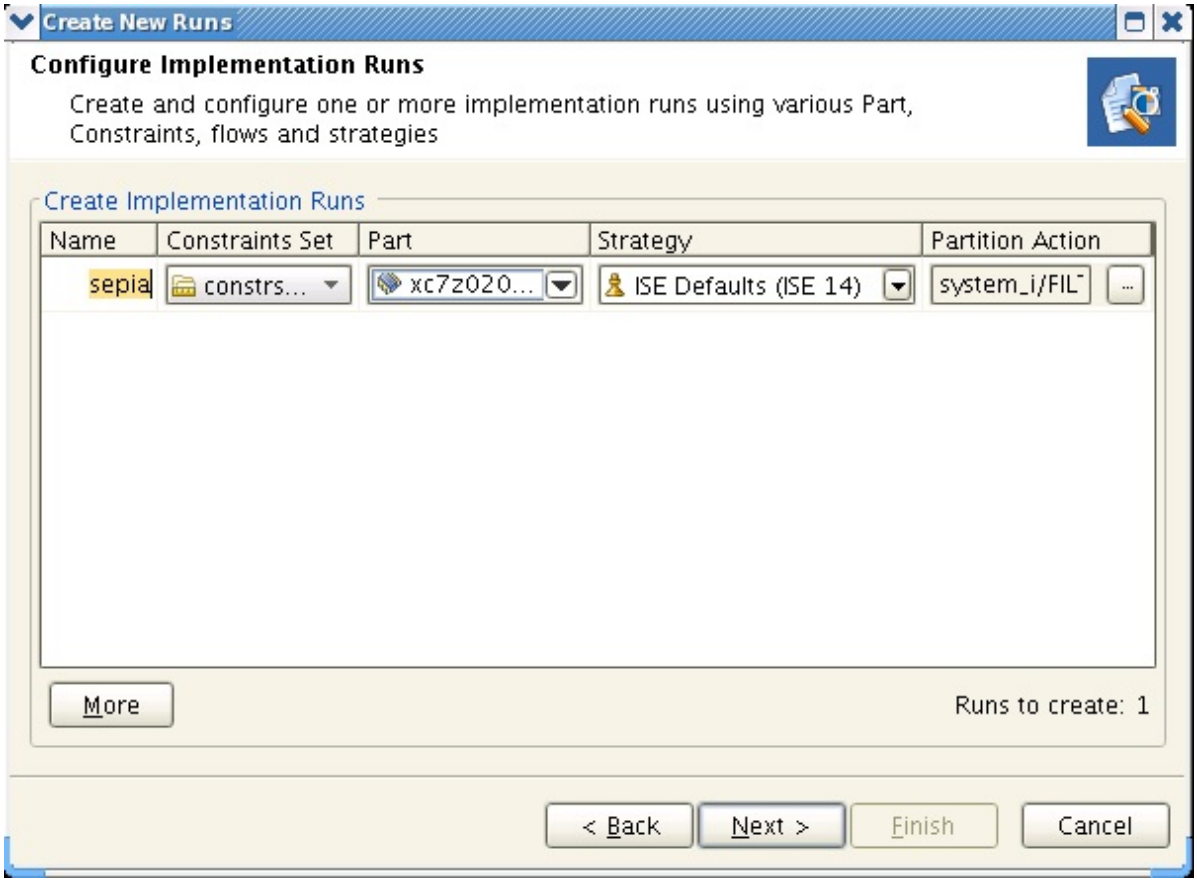
Tutorial

1. In the **Design Runs** window at the bottom, right-click into the empty space and select **Create Runs...**.



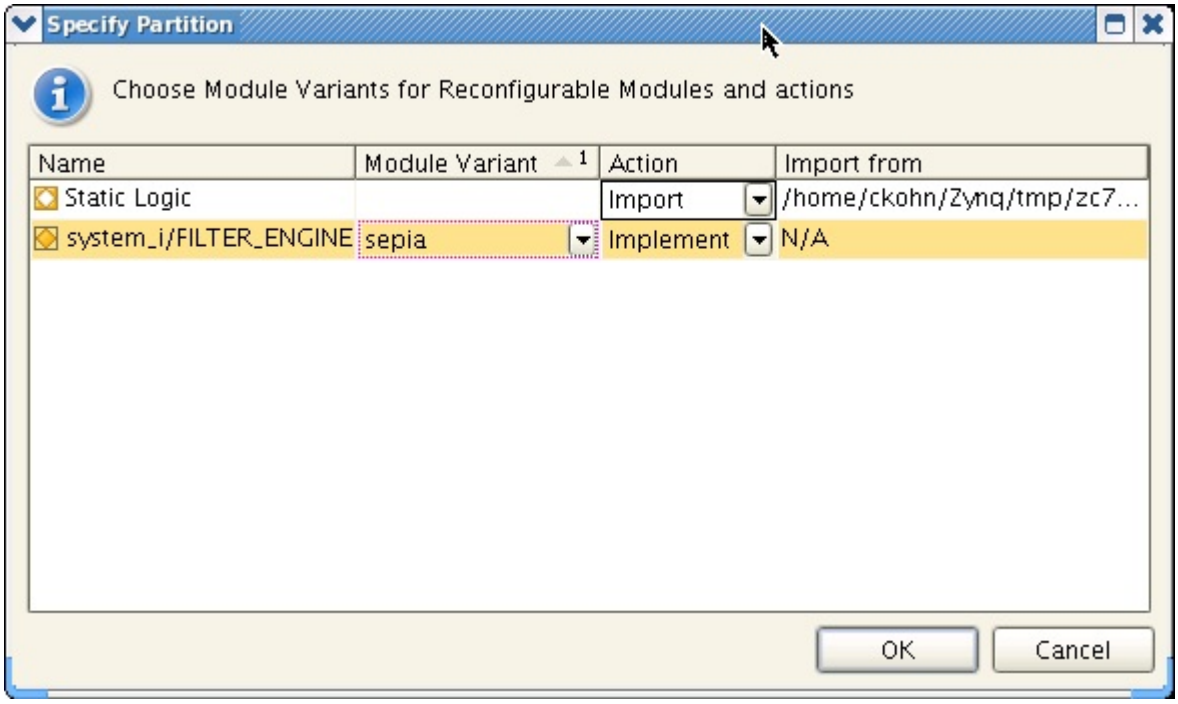
2. Click **Next**.

3. In the **Create Implementation Runs** dialog, change **Name** to **sepia**.



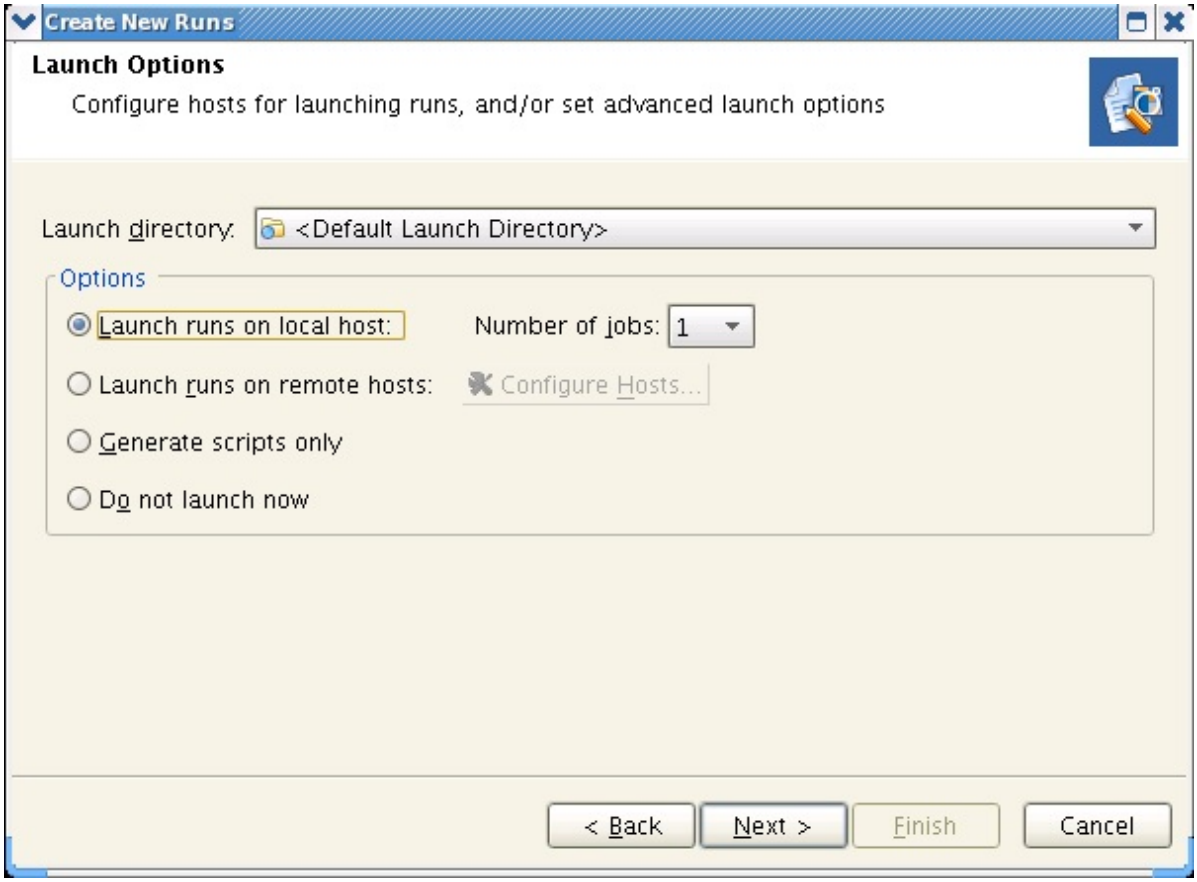
4. Click the **...** icon under **Partition Action**.
5. In the **Specify Partition** dialog, select **sepia** under **Module Variant** and **Implement** under **Action**.
6. Make sure that for **Static Logic**, **Action** is set to **Import**.
   Note: This will import the static logic from the promoted **sobel** design run instead of re-implementing it.

7. Click **OK**.



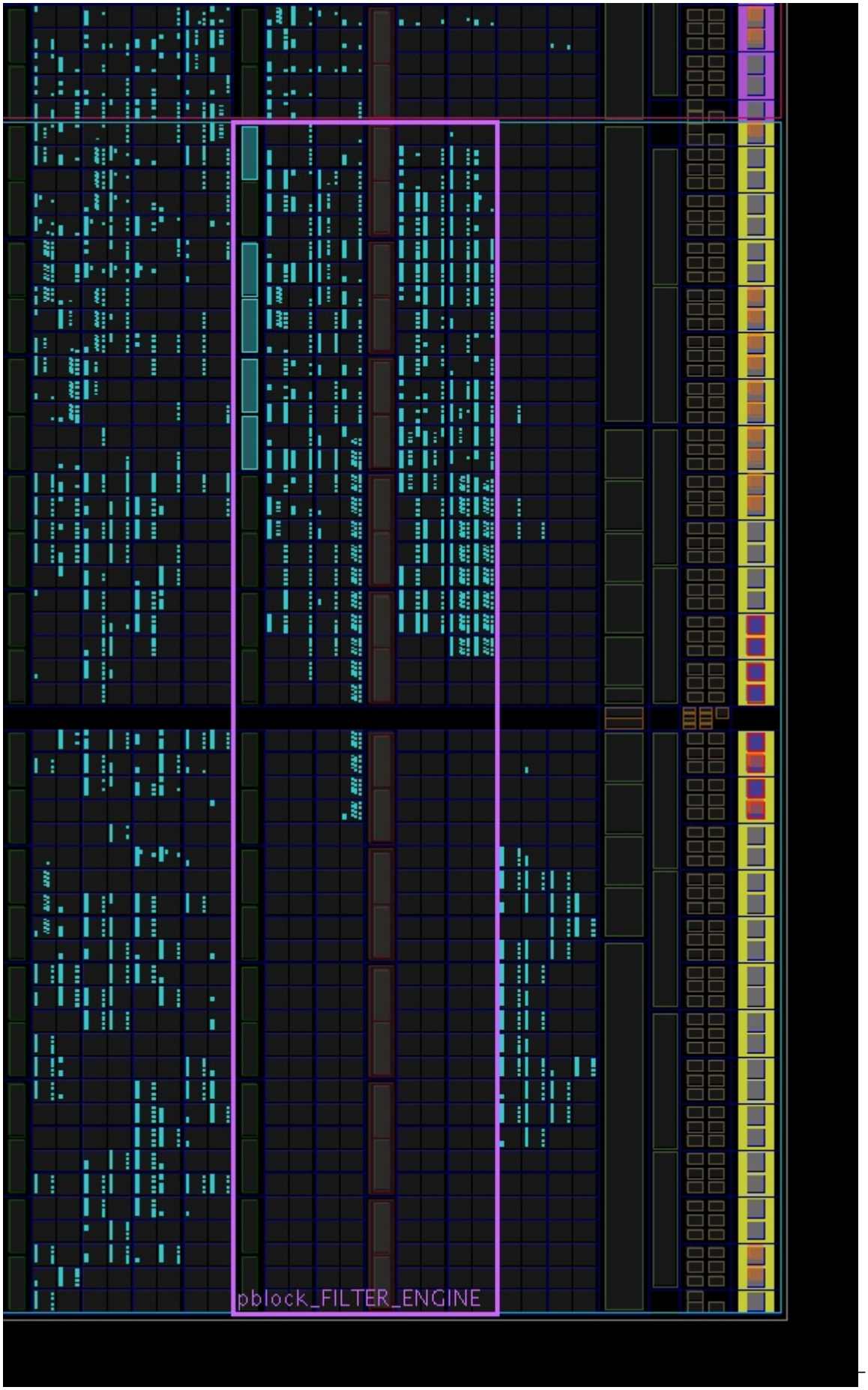
8. Click **Next**.
9. In the **Launch Options** dialog, select **Launch runs on local host**.

10. Click **Next**.



11. Click **Finish**.
12. Click **OK** in the **Launch Run Critical Messages** dialog to ignore the warnings.
13. Optional: In the **Implementation Completed** dialog, select **Open Implemented Design** and examine the routed design in the **Device** window. How does it compare to the **sepia** design? How does it

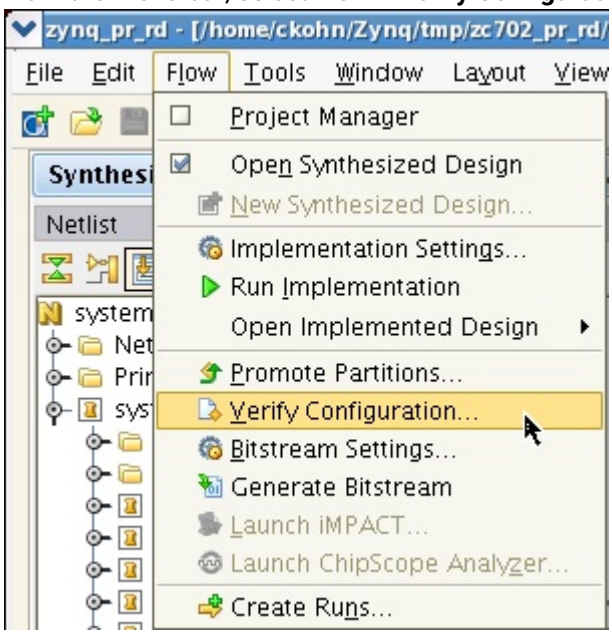
compare to the post-synthesis physical resources estimates?


pblock_FILTER_ENGINE

21.5.7 4.7 Running the Verify Configuration Utility

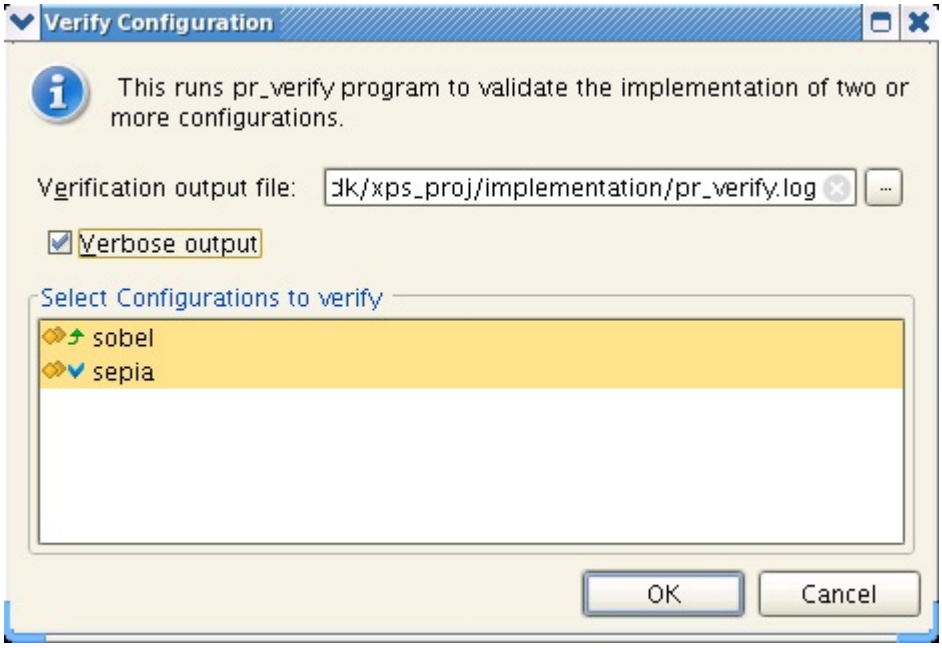
Tutorial

1. From the menu bar, select **Flow > Verify Configuration...**



2. Check the **Verbose output** box and make sure that both **sobel** and **sepia** configurations are highlighted.

3. Click **OK**.



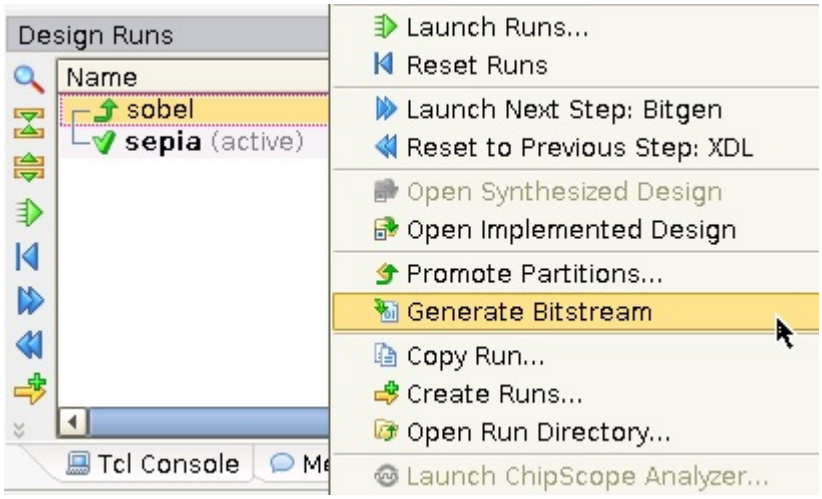
21.5.8 4.8 Generating Full and Partial Bitstreams

Tutorial

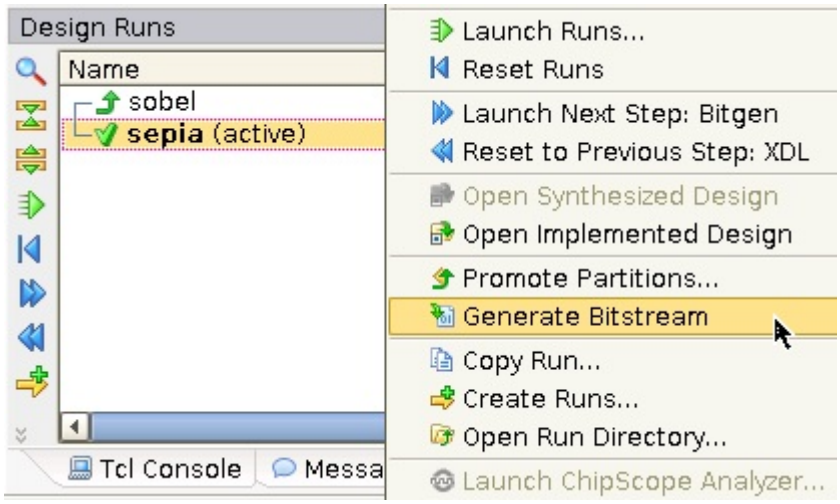
---

1. In the **Design Runs** window, select the **sobel** run, right-click and select **Generate Bitstream**.
   Note: The green up arrow next to **sobel** indicates that this run's configuration has been implemented and promoted, so its static logic can be imported by other configurations.

2. Click **OK**.



3. In the **Design Runs** window, select the **sepia** run, right-click and select **Generate Bitstream**. Note: The green check mark next to **sepia** indicates that this run's configuration has been implemented.
4. Click **OK**.



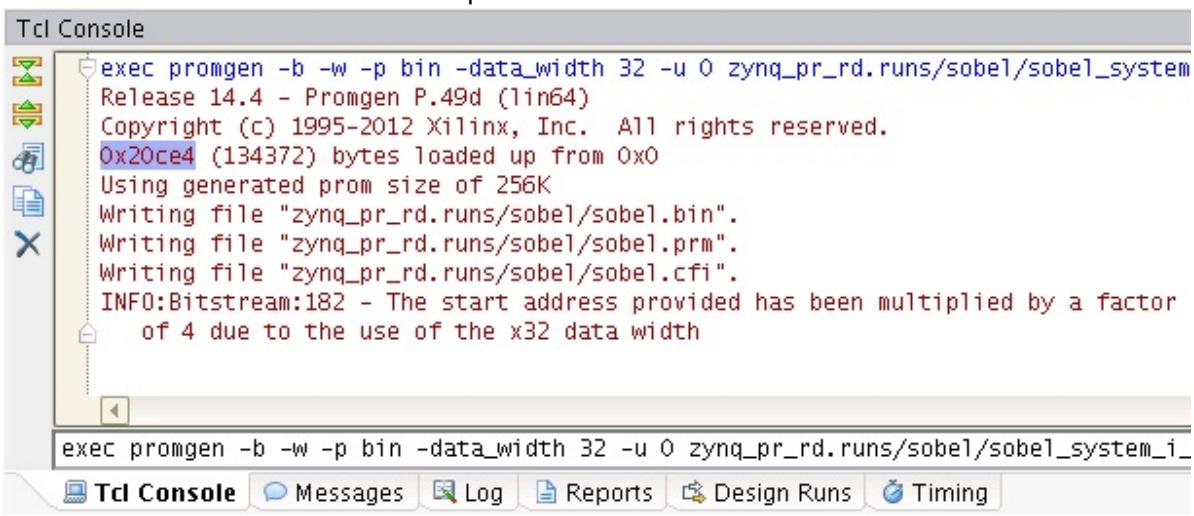
21.5.9 4.9 Converting Partial Bitstreams to Binary Format

Tutorial

---

1. Select the **Tcl Console** tab at the bottom.

2. At the command line prompt, enter exec promgen -b -w -p bin -data_width 32 -u 0 zynq_pr_rd.runs/sobel/sobel_system_i_FILTER_ENGINE_sobel_partial.bit -o zynq_pr_rd.runs/sobel/sobel.bin to convert the partial sobel bitfile to binary format.
3. Next, enter exec promgen -b -w -p bin -data_width 32 -u 0 zynq_pr_rd.runs/sepia/sepia_system_i_FILTER_ENGINE_sepia_partial.bit -o zynq_pr_rd.runs/sepia/sepia.bin to convert the partial sepia bitfile to binary format.
4. Note down the size of the generated partial binary files as indicated by the **promgen** console output -- the file size should be identical for both partial binaries.



21.6 5 Linux Components

This section describes how to build Linux specific components i.e. the second stage boot loader u-boot, the Linux kernel image and device tree blob, and the Linux root file system. To complete this section, you are required to have a Linux development PC with the ARM GNU cross compile tool chain and the Git tool installed. Make sure you have your PATH and CROSS_COMPILE environment variables set correctly. You can use the corkscrew tool if you are having difficulties accessing Xilinx git repositories from behind a firewall.

21.6.1 5.1 Building the u-boot Second Stage Boot Loader

This section explains how to download the sources, configure, and build the u-boot second stage boot loader. For additional information, refer to the Xilinx Zynq u-boot wiki.

Shortcut: A pre-compiled u-boot executable is available at zc702_pr_rd/sw/boot/linux/u-boot.

Tutorial

Clone the latest Zynq u-boot git repository from the Xilinx git server.

```
> git clone git://github.com/xilinx/u-boot-xlnx.git
> cd u-boot-xlnx
```

Create a new branch named *zynq_pr_rd_14.4* based on the *xilinx-v14.4* tag.

```
> git checkout -b zynq_pr_rd_14.4 xilinx-v14.4
```

Configure u-boot for the Zynq ZC702 Base TRD.

```
> make ARCH=arm zynq_zc70x_config
```

Build the u-boot boot loader. The generated U-boot executable can be found at u-boot-xlnx/u-boot.

```
> make ARCH=arm
```

Rename the u-boot executable to u-boot.elf.

```
> mv u-boot u-boot.elf
```

21.6.2 5.2 Building the Linux Kernel Image and Device Tree Blob

This section explains how to download the sources, configure, patch, and build the Linux kernel image and the device tree blob. For additional information, refer to the Xilinx Zynq Linux wiki.

21.6.2.1 Linux Kernel Image

Shortcut: A pre-compiled Linux kernel image is available at zc702_pr_rd/sd/linux/uImage.

Tutorial

Clone the latest Zynq Linux kernel git repository from the Xilinx git server.

```
> git clone git://github.com/xilinx/linux-xlnx.git
```

```
> cd linux-xlnx
```

Create a new branch named *zynq_pr_rd_14.4* based on the *xilinx-v14.4* tag.

```
> git checkout -b zynq_pr_rd_14.4 xilinx-v14.4
```

Apply the PR RD specific patch on top of the Base TRD tag. The patch includes:

- Xilinx VDMA driver
- Xilinx Sobel/Sepia filter driver
- Zynq Base TRD config file (zynq_base_trd_defconfig)
- Zynq Base TRD device tree file (zynq_base_trd.dts)
- Mouse sensitivity patch

```
> cp zc702_pr_rd/sw/patch/zc702_pr_rd_14_4.patch linux-xlnx  // copy the PR RD patch
> git apply --stat zc702_pr_rd_14_4.patch                    // display contents of
patch
> git apply --check zc702_pr_rd_14_4.patch                   // check if patch can be
applied
> git am
zc702_pr_rd_14_4.patch                                  // apply the patch
```

Configure the Linux kernel for the Zynq ZC702 Base TRD (same config as the PR RD).

```
> make ARCH=arm zynq_base_trd_defconfig
```

Build the Linux kernel. The generated kernel image can be found at linux-xlnx/arch/arm/boot/uImage.

```
> make ARCH=arm uImage
```

21.6.2.2 Linux Device Tree Blob

Shortcut: A pre-compiled Linux device tree blob is available at zc702_pr_rd/sd/linux/devicetree.dtb.

Tutorial

Compile the Base TRD device tree file. The output of this step is a device tree blob which can be found at linux-xlnx/devicetree.dtb.

```
> ./scripts/dtc/dtc -I dts -O dtb -o devicetree.dtb ./arch/arm/boot/dts/
zynq_base_trd.dts
```

21.6.3 5.3 Building the Linux Root File System

For instructions on how to build a Zynq Root File System, please refer to the Xilinx Zynq Root File System Creation wiki.

Note: These steps are just an example; they will not recreate the ramdisk shipped with this reference design.

Shortcut: A pre-built ramdisk image is available at zc702_pr_rd/sd/linux/uramdisk.image.gz.

At the end of the etc/init.d/rcS script, a hook was added to execute a customized user script named init.sh. Our implementation of this script is located at zc702_pr_rd/sd/linux/init.sh and takes care of the following design specific initialization:

- Mount the cross-compiled Qt/Qwt libraries image file (located at zc702_pr_rd/sd/linux/qt_lib.img)
- Create Xilinx VDMA device node
- Create Xilinx filter device node
- Auto-start the Qt GUI based software application after boot-up

To remove the auto-start feature of the Qt application, simply comment the line ./run_sobel.sh -qt inside zc702_pr_rd/sd/linux/init.sh.

21.7 6 SDK Flow

This section describes how to use SDK to compile the standalone and Linux based software applications that control the video data path and the dynamic partial reconfiguration. You will also learn how to compile the First Stage Boot Loader (FSBL) and how to create a standalone or Linux Zynq boot image. For detailed information on SDK, the Zynq boot image format and boot process, refer to UG821 .

21.7.1 6.1 Creating a Hardware Platform Specification

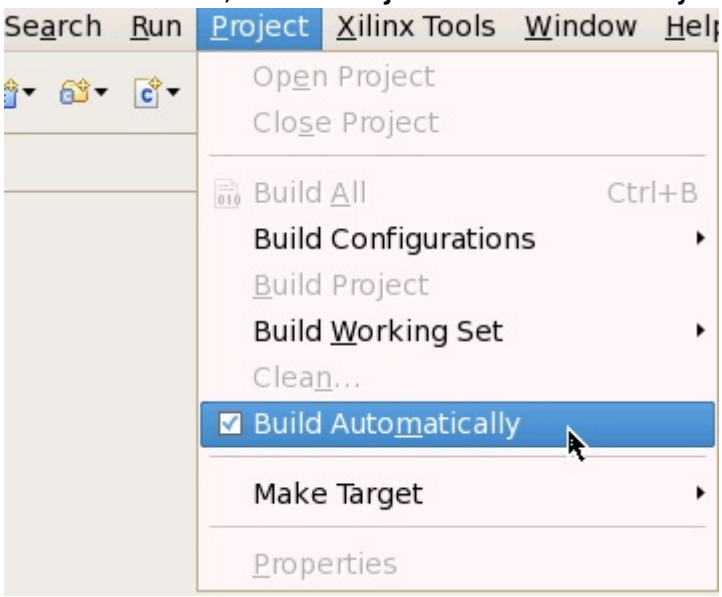
The Hardware Platform Specification is obtained by running PlanAhead's **Export to SDK** tool (see Section 3.3). It generates an XML file (system.xml) that describes the hardware system including PS and PL components and C source files that initialize the PS (ps7_init.c/h). Follow the steps below to create a Hardware Platform Specification SDK project.

Shortcut: A pre-generated Hardware Platform Specification SDK project is available at zc702_pr_rd/sw/workspace/hw_platform.

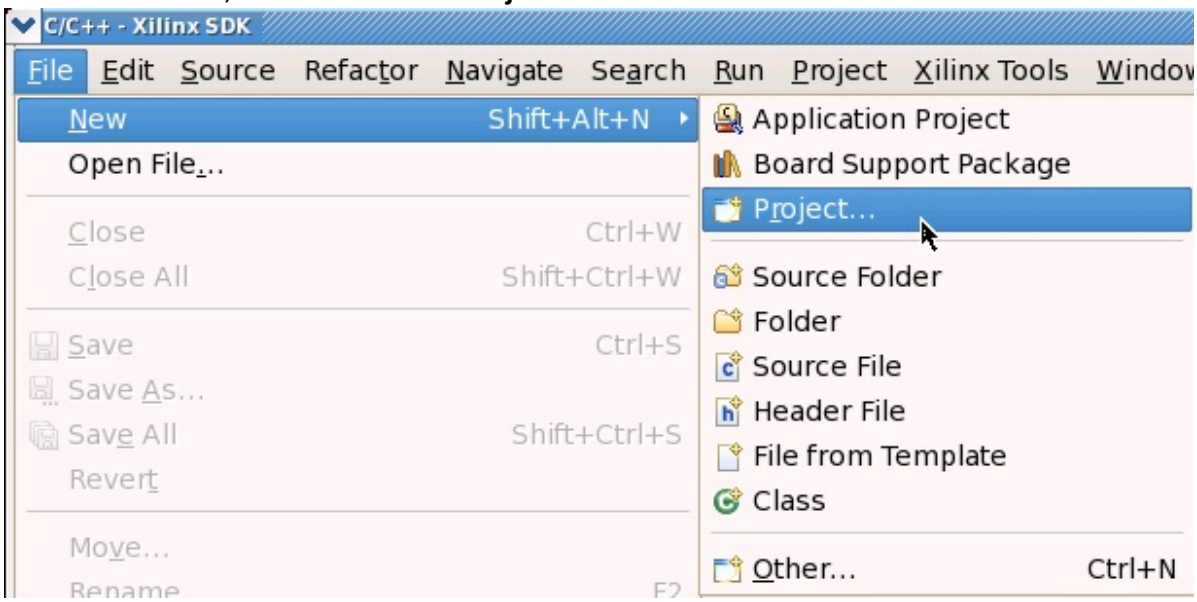
Tutorial

---

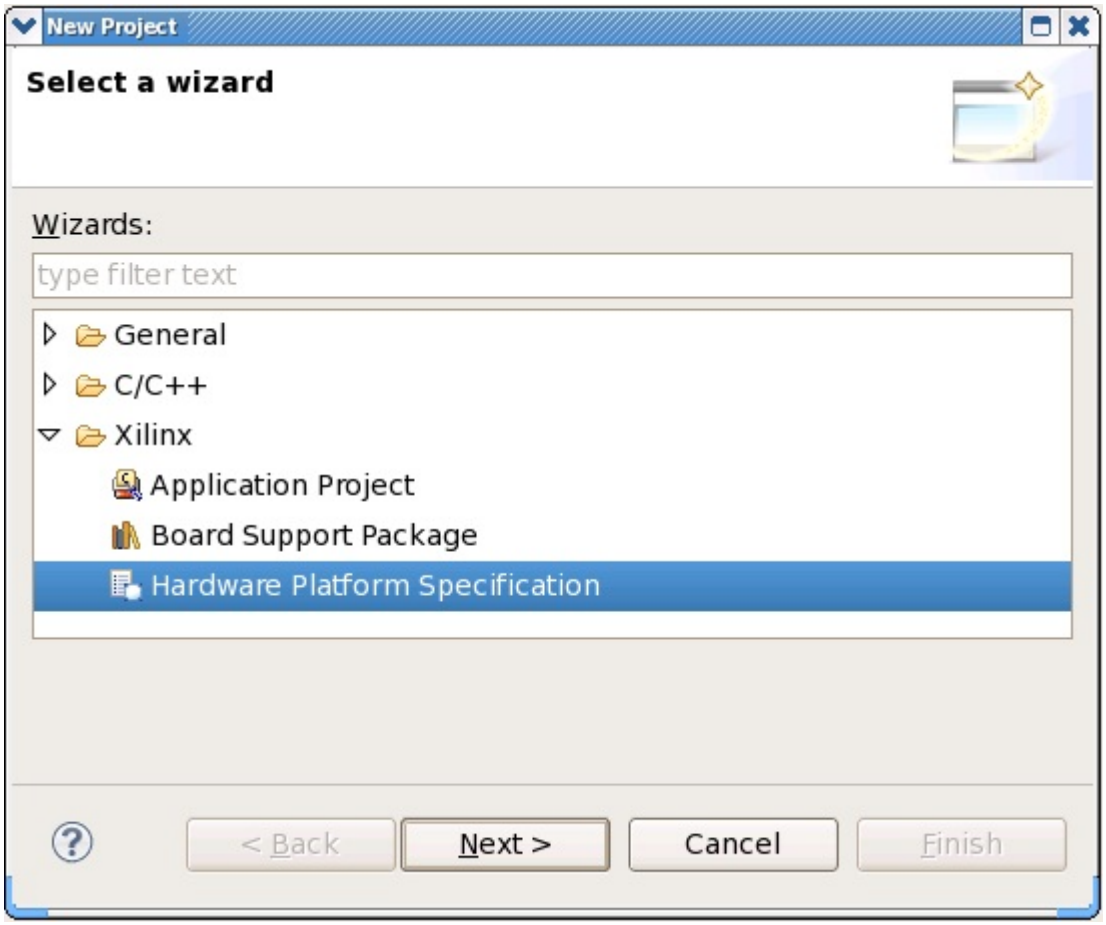
1. To open SDK, select **Start > All Programs > Xilinx Design Tools > ISE Design Suite 14.4 > EDK > Xilinx Software Development Kit**.
2. **Browse** to the zc702_pr_rd/sw/workspace directory for **Workspace** and click **OK**.
3. Click **OK**.
4. Close the welcome screen.
5. From the menu bar, uncheck **Project > Build Automatically**.



6. From the menu bar, select **File > New > Project**.

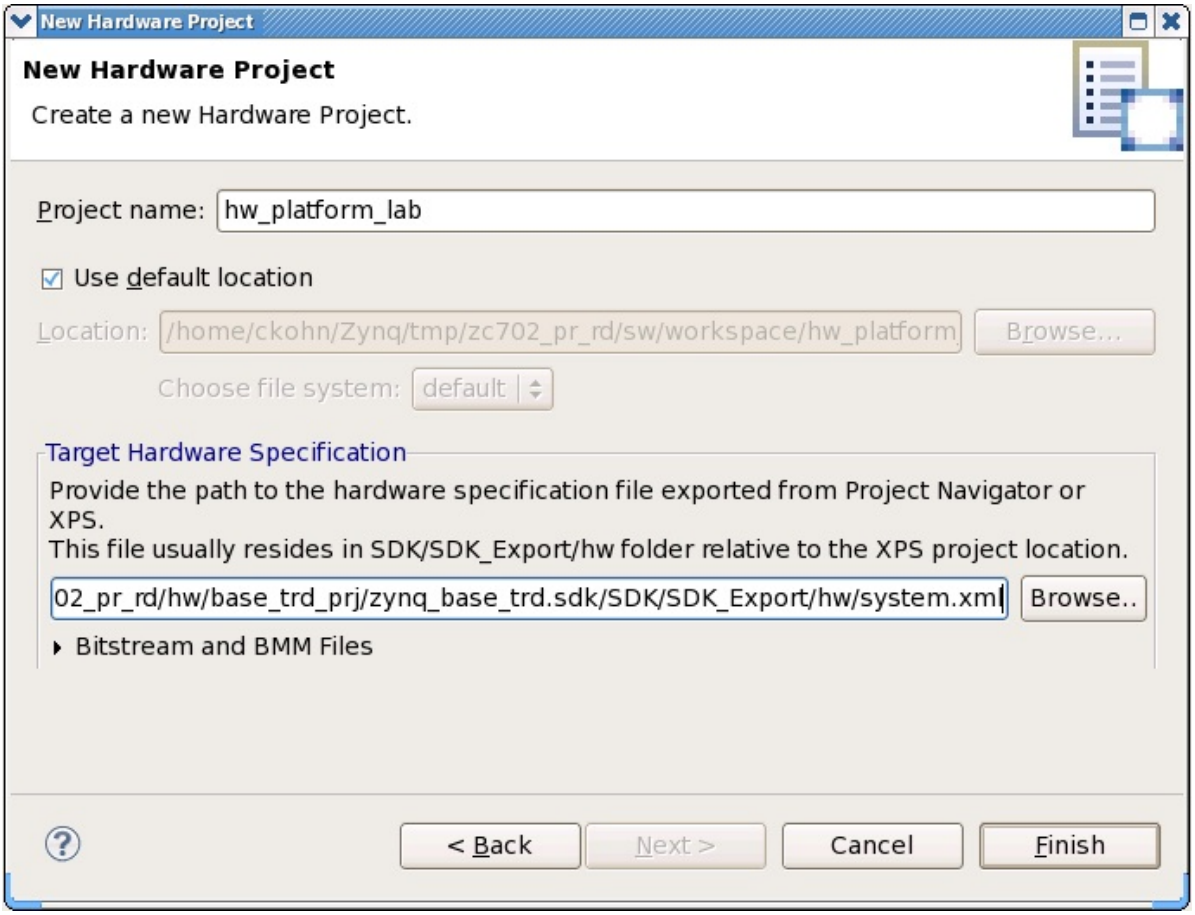


7. In the **New Project** wizard, select **Xilinx > Hardware Platform Specification**.
8. Click **Next**.

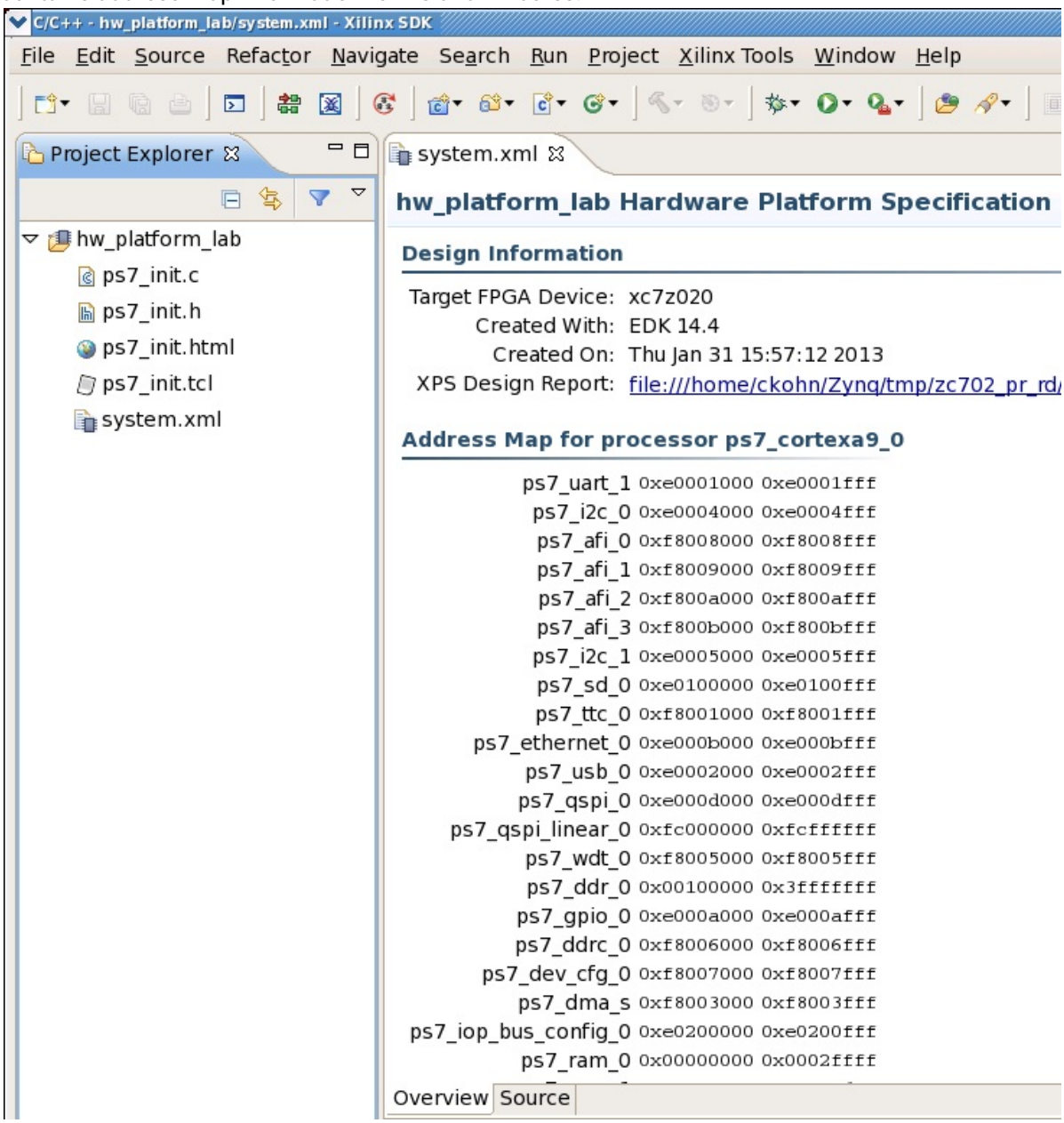


9. Enter a **hw_platform_lab** in the **Project Name** field and **Browse** to the export location of the hardware specification file (zc702_pr_rd/hw/base_trd_prj/zynq_base_trd.sdk/SDK/SDK_Export/hw/system.xml).

10. Click **Finish**.



11. You can see the imported hardware platform files in the **Project Explorer**. The system.xml file contains address map information for PS and PL cores.



21.7.2 6.2 Generating a Board Support Package

The Board Support Package (BSP) uses the information in the Hardware Platform Specification to assign drivers to the hardware components in the design. All PS drivers and most of the PL drivers are shipped with the SDK tool suite. Custom PL drivers and software services are provided as local user repository at

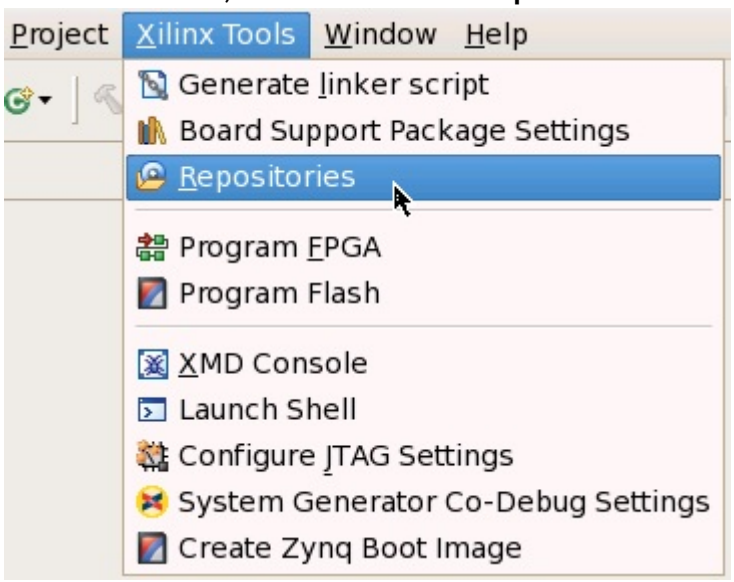
zc702_pr_rd/sw/repo. The BSP is linked to the main application as a library.

Shortcut: A pre-generated Board Support Package SDK project is available at zc702_pr_rd/sw/workspace/standalone_bsp.

Tutorial

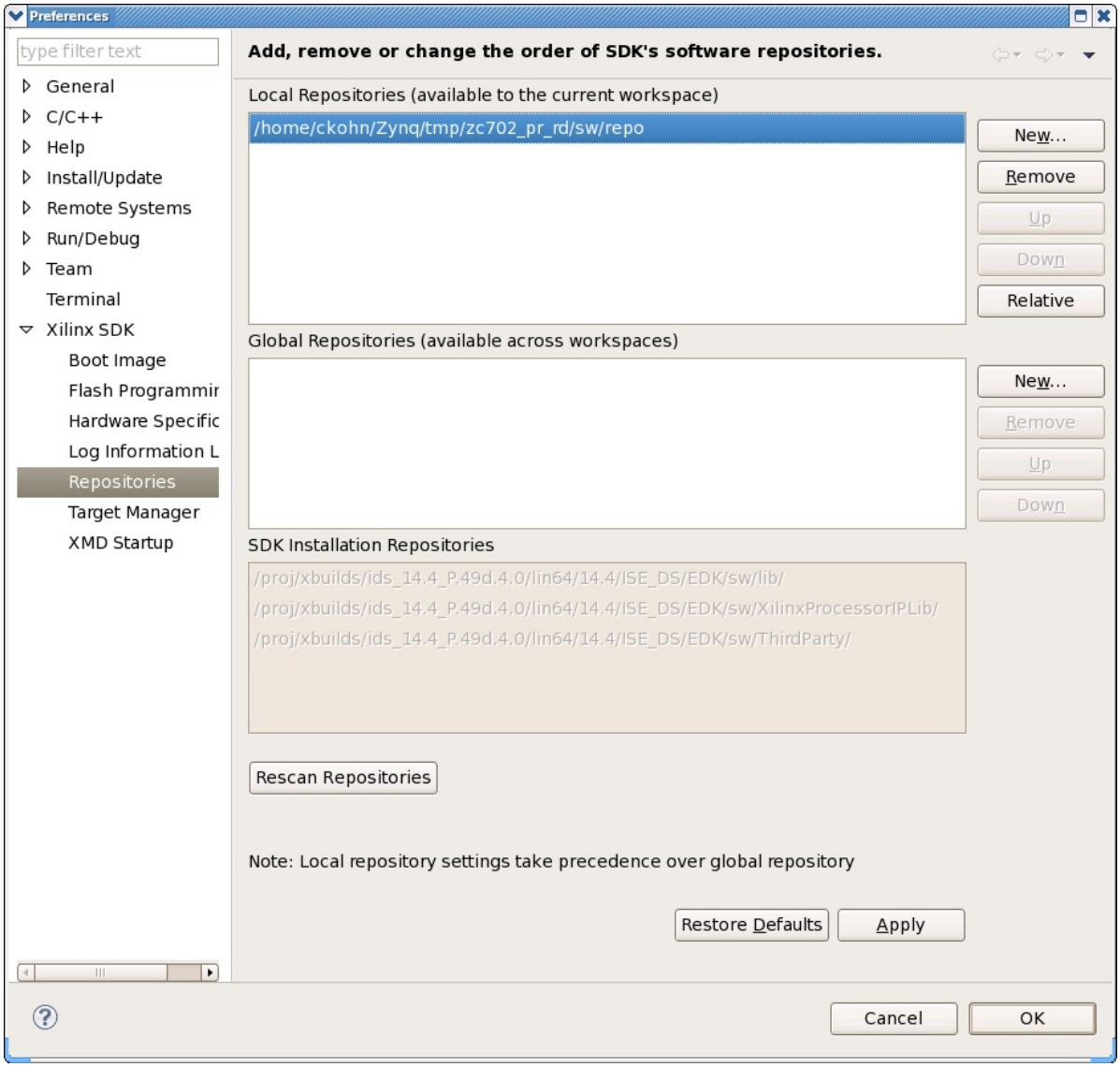
---

1. From the menu bar, select **Xilinx Tools > Repositories**.

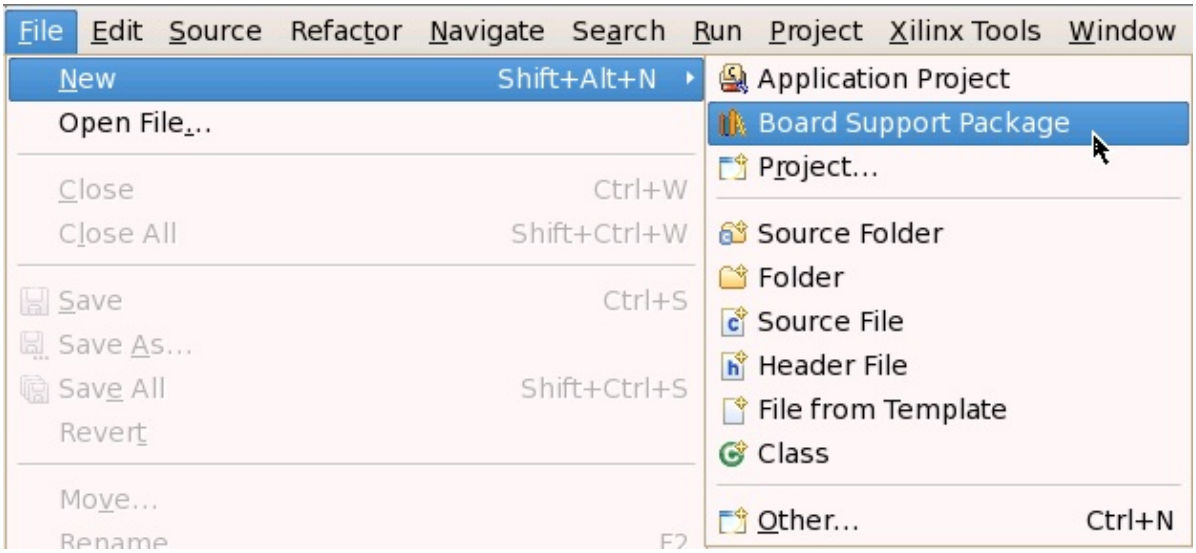


2. Next to the **Local Repositories** text field, select **New...**, browse to the zc702_pr_rd/sw/repo directory, and click **OK**.

3. Click **OK**.

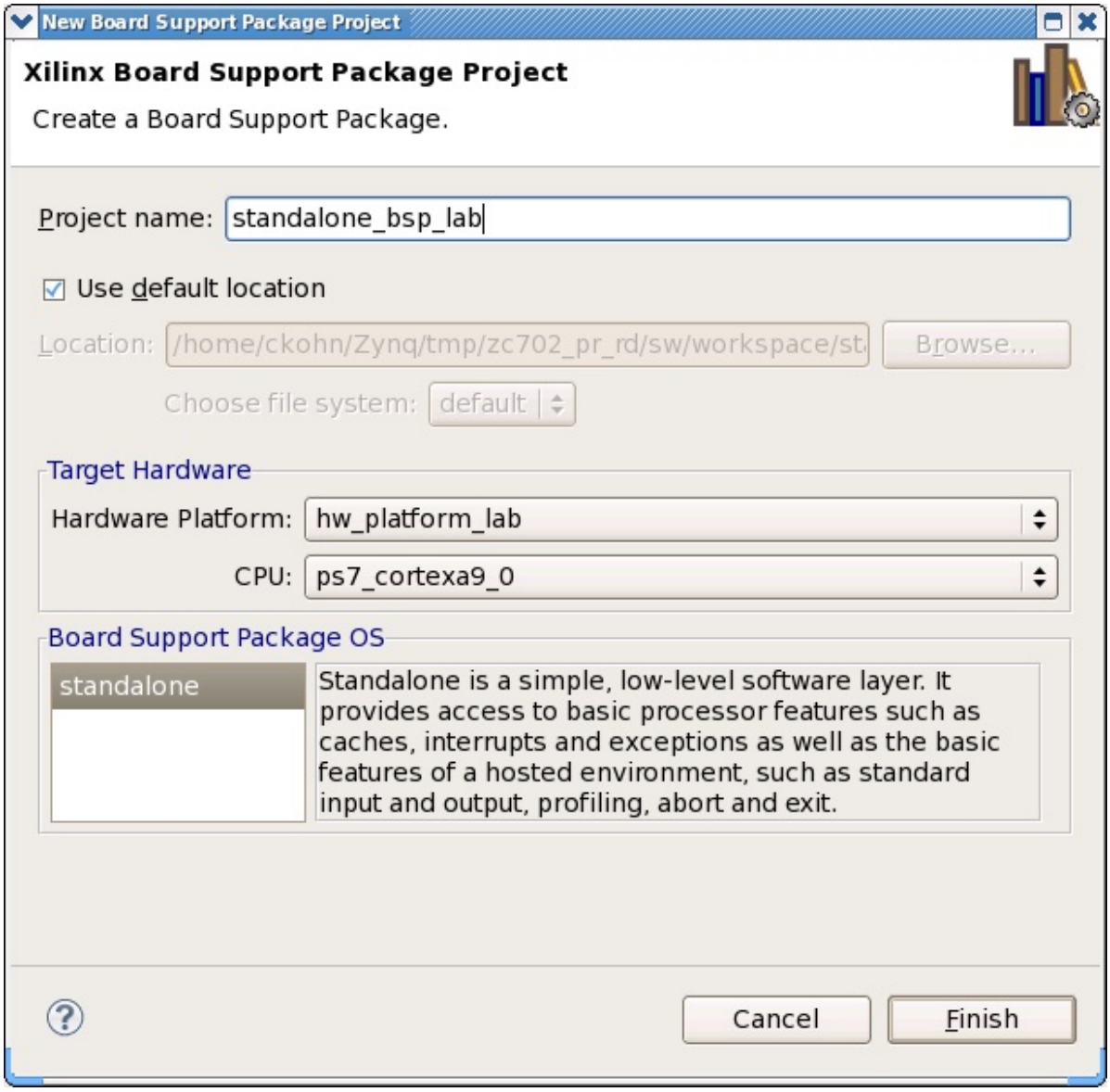

Preferences
type filter text
General
C/C++
Help
Install/Update
Remote Systems
Run/Debug
Team
Terminal
Xilinx SDK
Boot Image
Flash Programmir
Hardware Specific
Log Information L
Repositories
Target Manager
XMD Startup
Add, remove or change the order of SDK's software repositories.
Local Repositories (available to the current workspace)
/home/ckohn/Zynq/tmp/zc702_pr_rd/sw/repo
New...
Remove
Up
Down
Relative
Global Repositories (available across workspaces)
New...
Remove
Up
Down
SDK Installation Repositories
/proj/xbuilds/ids_14.4_P.49d.4.0/lin64/14.4/ISE_DS/EDK/sw/lib/
/proj/xbuilds/ids_14.4_P.49d.4.0/lin64/14.4/ISE_DS/EDK/sw/XilinxProcessorIPLib/
/proj/xbuilds/ids_14.4_P.49d.4.0/lin64/14.4/ISE_DS/EDK/sw/ThirdParty/
Rescan Repositories
Note: Local repository settings take precedence over global repository
Restore Defaults
Apply
Cancel
OK

4. From the menu bar, select **File > New > Xilinx Board Support Package**.



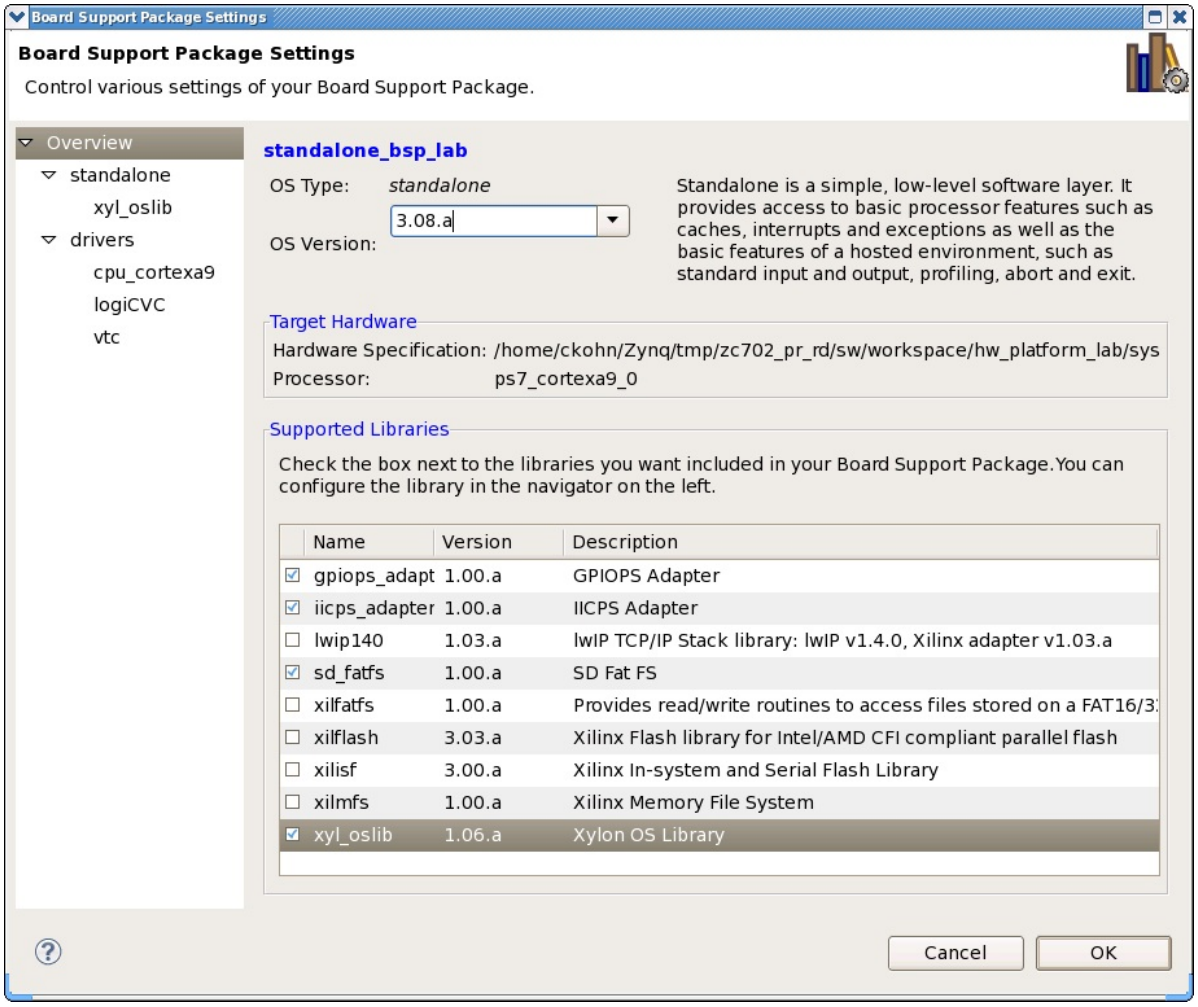
5. Enter **standalone_bsp_lab** in the **Project Name** field and select hw_platform_lab as **Hardware Platform** and ps7_cortexa9_a0 as **CPU**.

6. Click **Finish**.



7. In the **Board Support Package Settings** window, select **Overview** on the left and check the following libraries: gpiops_adapter, iicps_adapter, sd_fatfs, and xyl_oslib. The libraries are provided in the user repository.

8. Click **OK**.



21.7.3 6.3 Compiling the Standalone Software Application

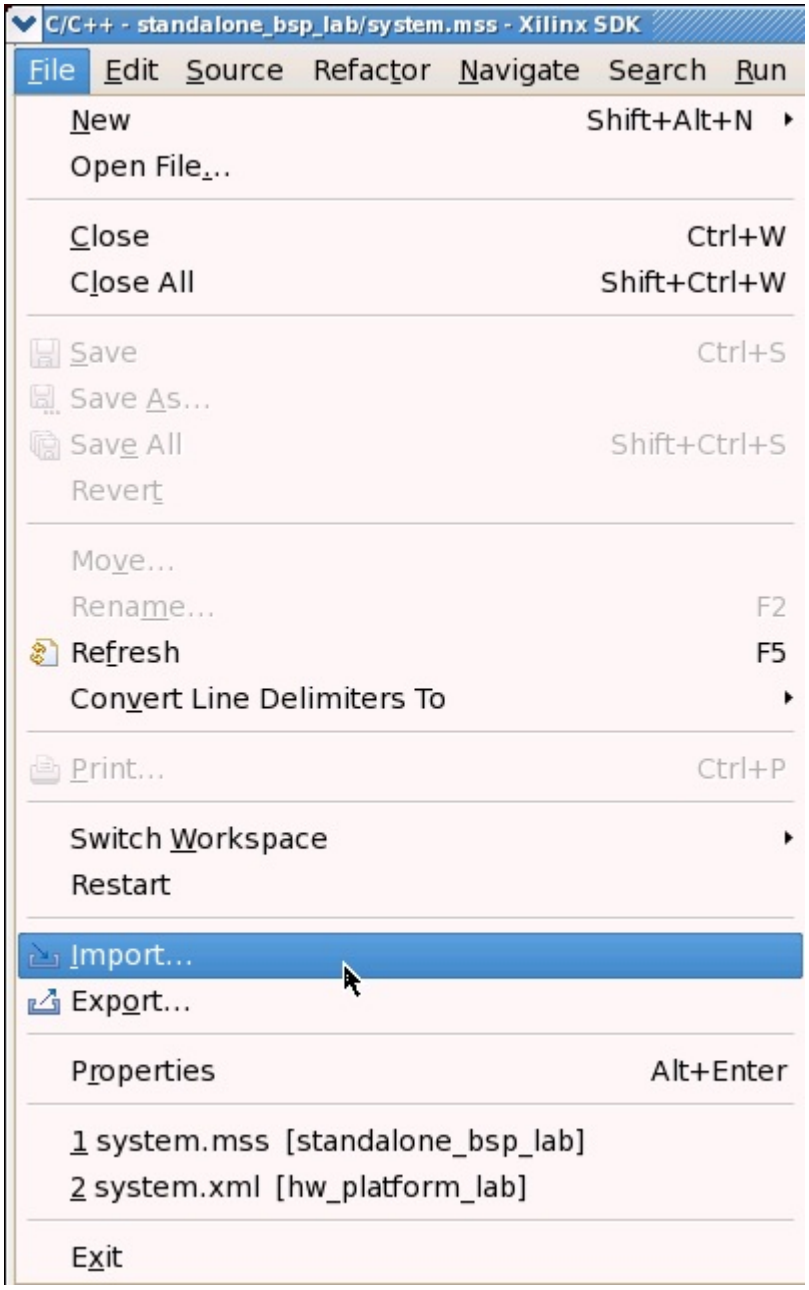
Compiling the standalone or bare-metal software application for Partial Reconfiguration requires Hardware Platform Specification (6.1) and BSP (6.2) SDK projects. This tutorial uses the pre-generated Hardware Platform and BSP SDK projects.

Shortcut: A pre-compiled executable is available at zc702_pr_rd/sw/boot/standalone/filter_std.elf.

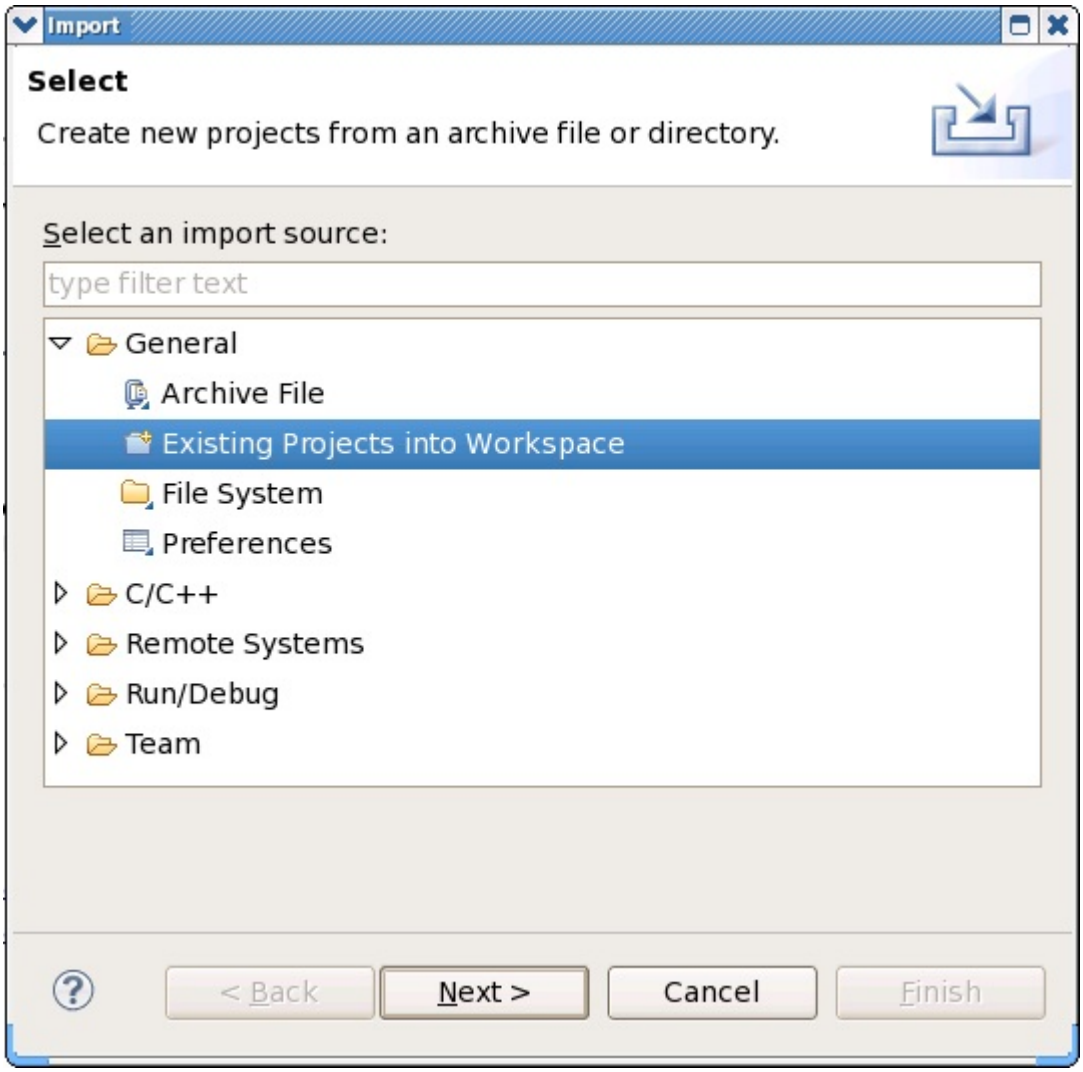
Tutorial

---

1. From the menu bar, select **File > Import...**.

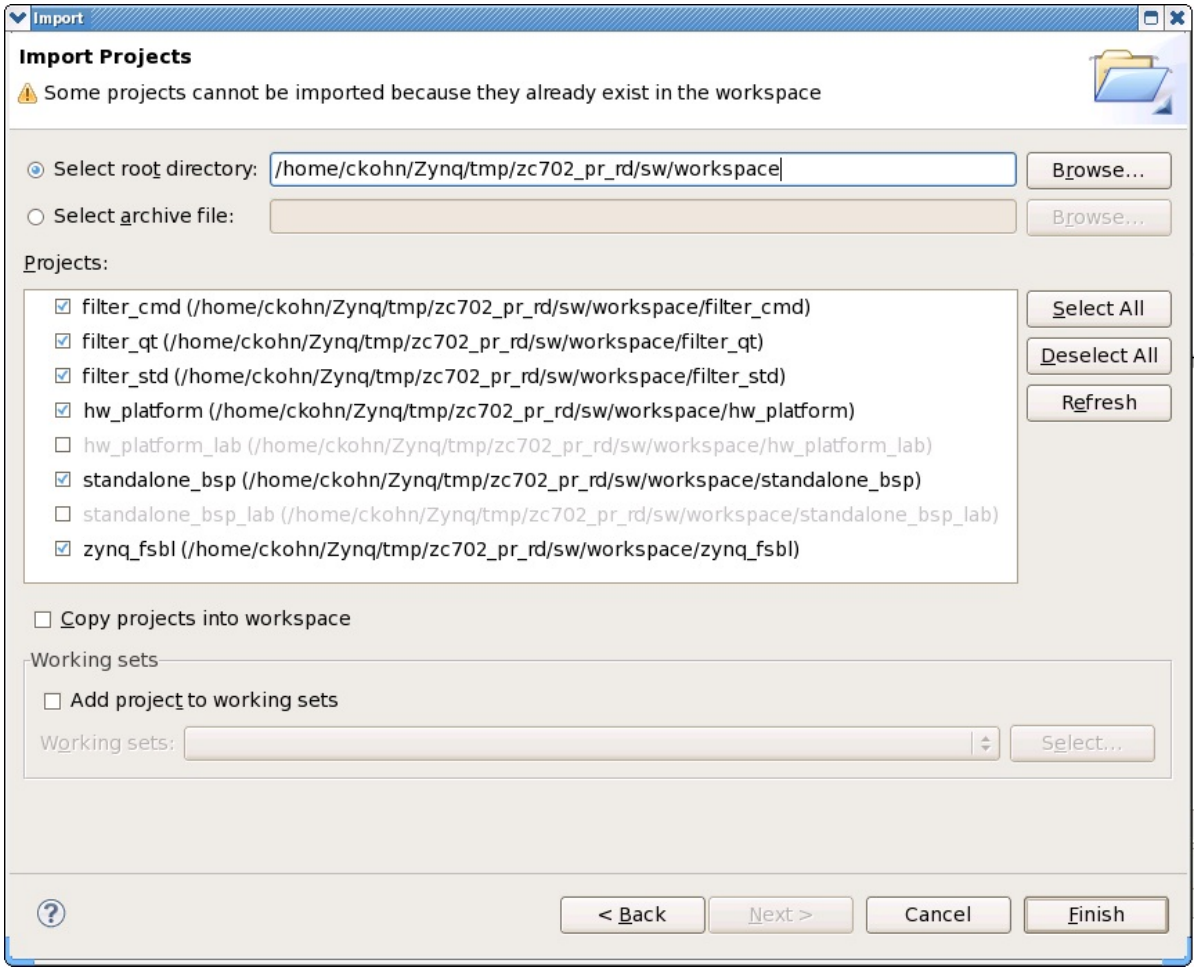


2. In the **Import** dialog, expand the **General** folder, select **Existing Projects into Workspace**, and click **Next**.



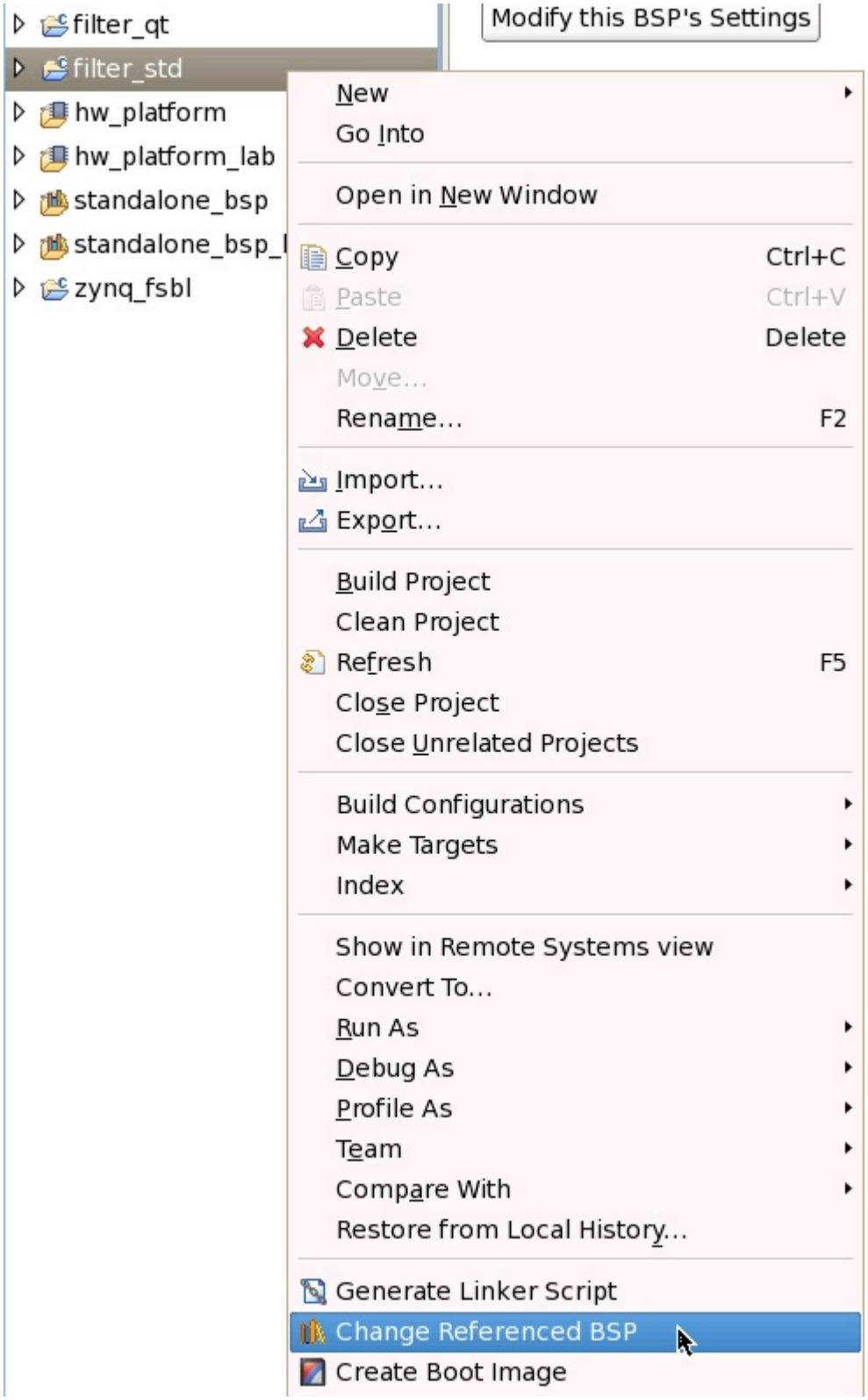
3. Next to the **Select root directory** text field, **Browse** to the zc702_pr_rd/sw/workspace directory, and click **OK**.
4. Check all projects for import. Some of them are only needed in subsequent sections.

5. Click **Finish**.

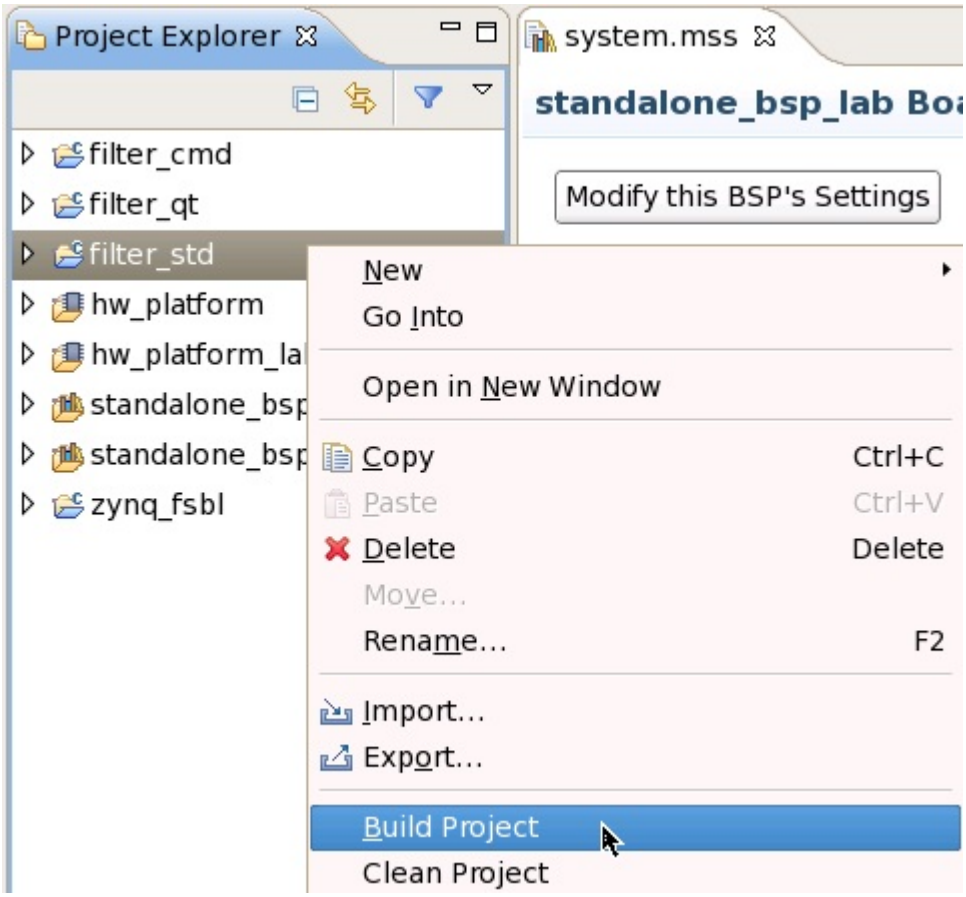


6. Optional: If you want to use your own Hardware Platform Specification and BSP projects instead of the imported ones, you need to change the referenced BSP first. Right-click **filter_std** and select

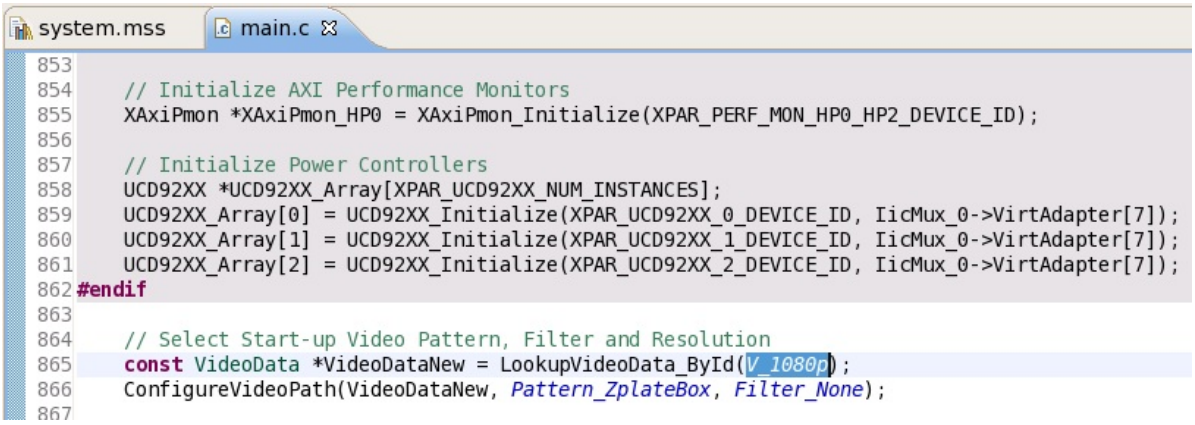
Change Referenced BSP, then select your BSP and click **OK**.



7. Right-click **filter_std** and select **Build Project**.



8. To change the video resolution, expand the filter_std project and open the file src/main.c. Search for the string V_1080p and change the parameter passed into the LookupVideoData_ById() function to V_720p. The corresponding data structure is defined in the file(s) src/video_common.h/c.
Note: this application supports and has been tested with 1080p and 720p video resolutions. Use any other resolution at your own risk.



9. Save the file.
10. Re-compile the standalone application.

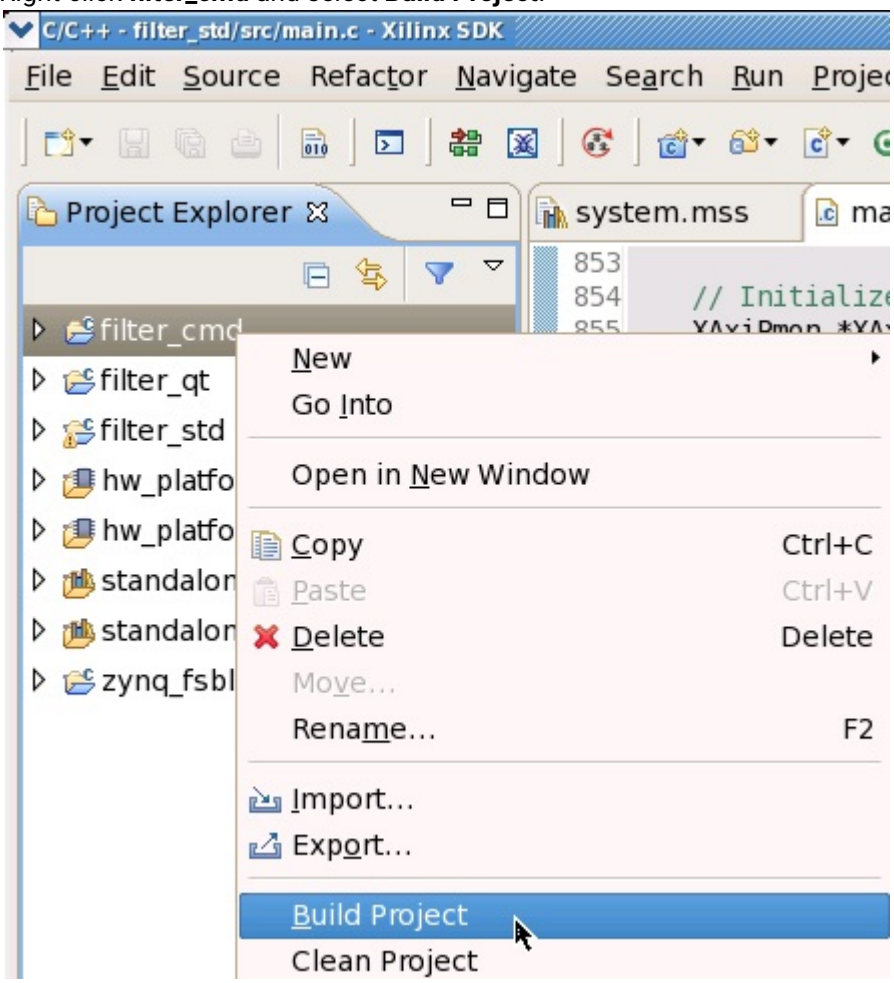
21.7.4 6.4 Compiling the Linux Command Line Software Application

The command line Linux software application has no dependencies on any other SDK projects and can be compiled by itself.

Shortcut: A pre-compiled executable is available at zc702_pr_rd/sd/linux/filter_cmd.

Tutorial

1. This tutorial assumes that the project has already been imported into SDK (see steps in Section 6.3).
2. Right-click **filter_cmd** and select **Build Project**.



21.7.5 6.5 Compiling the Linux Qt Software Application

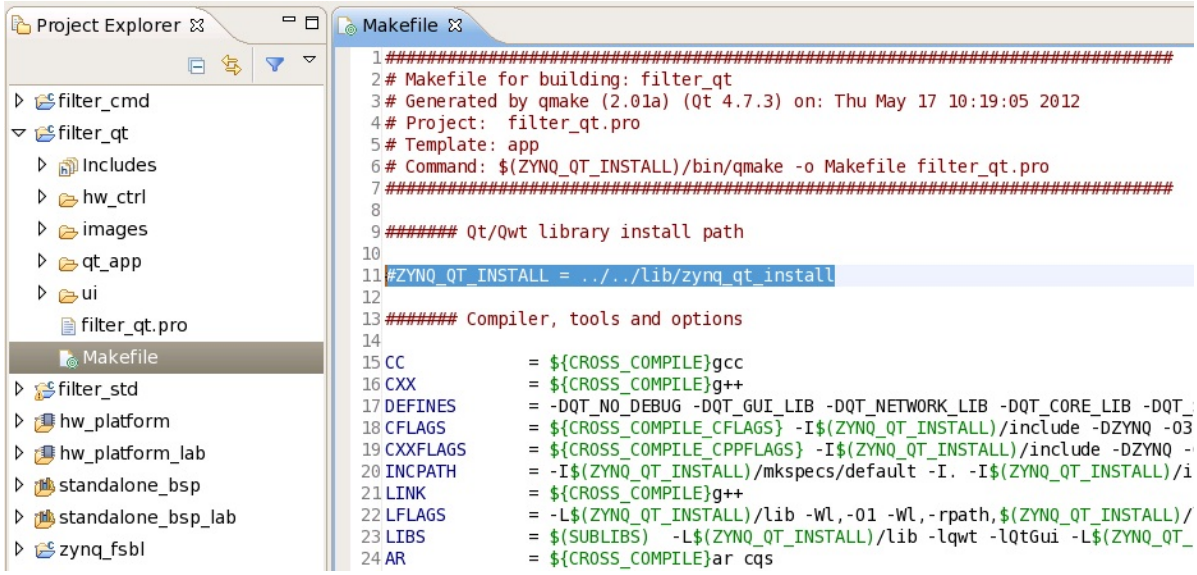
The Qt Linux software application offers a GUI for navigation, built on top of the embedded Qt framework and the Qwt widget library. To complete this step, you are required to have a Linux development PC with the ARM GNU cross compile tool chain installed. For a detailed tutorial on how to build the Qt/Qwt libraries, refer to the Zynq Qt/Qwt Libraries - Build Instructions wiki. The Qt/Qwt libraries are required both, at run time and compile time. The pre-compiled Qt/Qwt image file is sufficient when executing the application on the Zynq platform. When compiling the application, in most cases you will need to recompile the Qt/Qwt libraries on your host platform since some of the generated Qt utilities are host-specific.

Shortcut: A pre-compiled Qt/Qwt libraries image file is available at zc702_pr_rd/sd/linux/qt_lib.img along with a pre-compiled executable at zc702_pr_rd/sd/linux/filter_qt. The executable also requires an images directory zc702_pr_rd/sd/linux/images/ containing data flow graphs that are displayed when the application is executed.

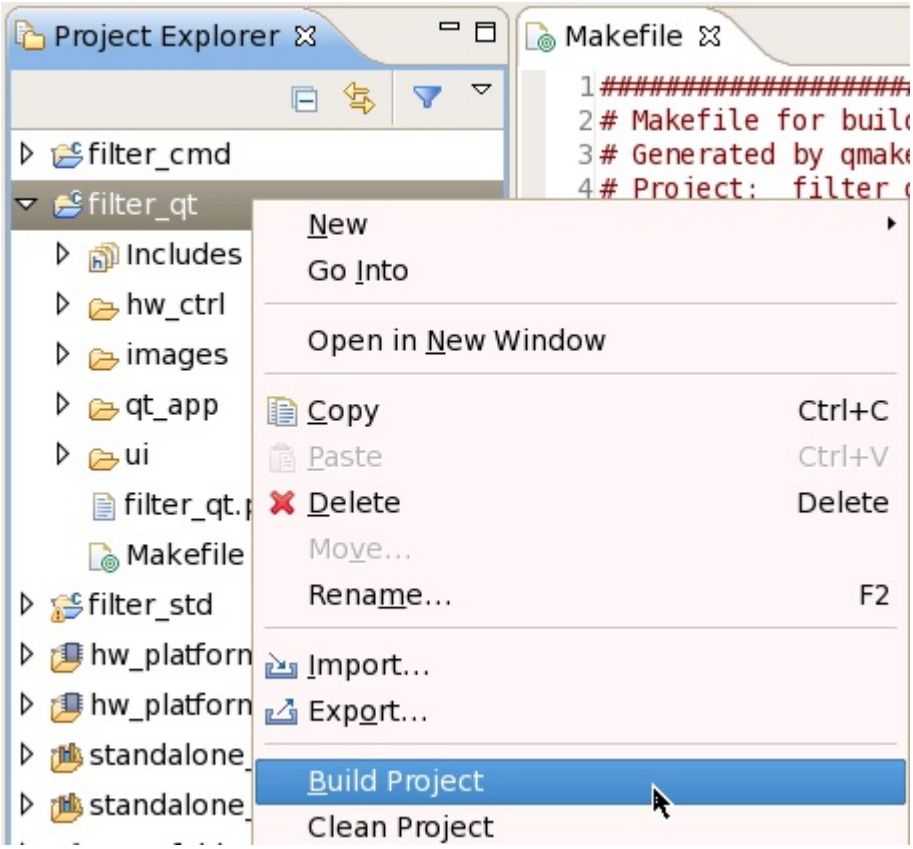
Tutorial

---

1. This tutorial assumes that the project has already been imported into SDK (see steps in Section 6.3).
2. Expand the filter_qt SDK project in the **Project Explorer** and double-click the Makefile to open it. Uncomment the highlighted line in the Makefile and set the ZYNQ_QT_INSTALL variable to your Qt/Qwt libraries installation path. Alternatively, you can create and point the ZYNQ_QT_INSTALL environment variable to your Qt/Qwt libraries installation path.
3. Save the file.



4. Right-click the **filter_qt** project and select **Build Project**.



21.7.6 6.6 Compiling the First Stage Boot Loader

Compiling the FSBL requires Hardware Platform Specification (6.1) and BSP (6.2) SDK projects. This tutorial uses the pre-generated Hardware Platform and BSP SDK projects. The FSBL SDK project has two build configurations, one named standalone, the other named linux. The linux configuration is a customized version of the standard Xilinx FSBL which adds the following functionality:

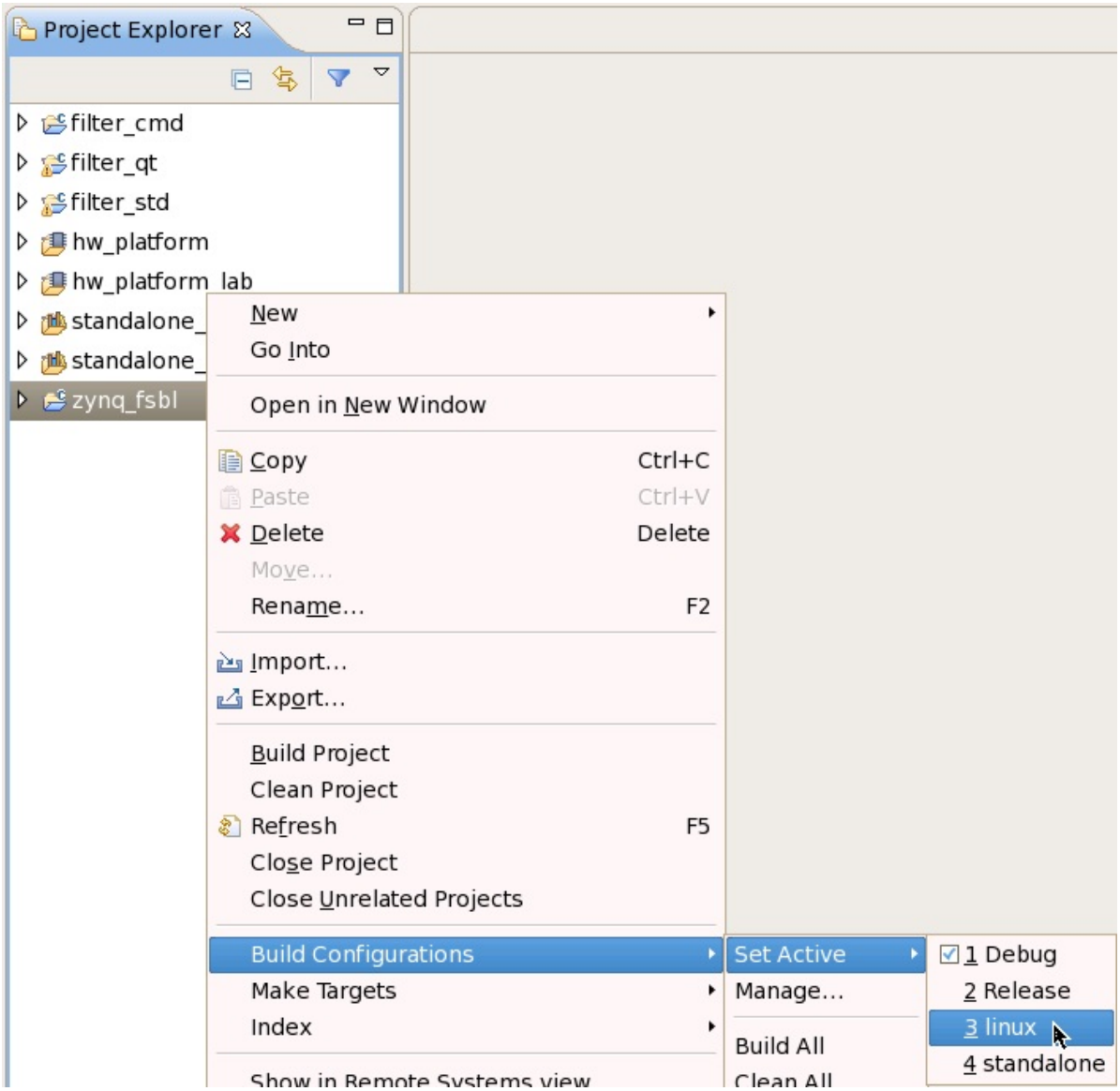
- I2C initialization sequence for HDMI transmitter (ADV7511) on ZC702 base board
- I2C FMC-IMAGEON daughter card detection sequence
- I2C initialization sequence for HDMI receiver (ADV7611) on Avnet FMC-IMAGEON

In the standalone case, the corresponding initialization is done inside the application itself instead of the FSBL.

Shortcut: Pre-compiled FSBL executables are available at zc702_pr_rd/sw/boot/standalone/zynq_fsbl.elf and zc702_pr_rd/sw/boot/linux/zynq_fsbl.elf.

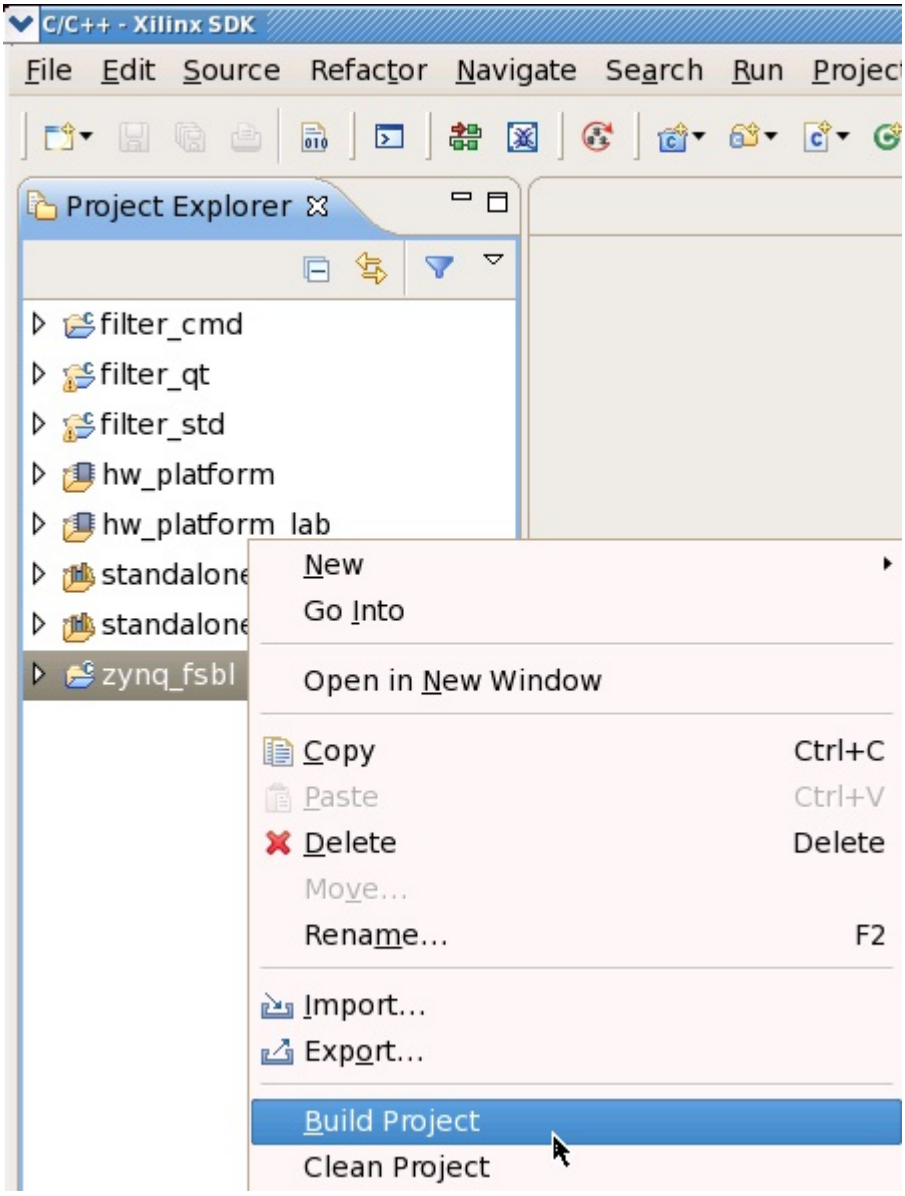
Tutorial

1. This tutorial assumes that the Hardware Platform Specification, BSP, and FSBL projects have already been imported into SDK (see steps in Section 6.3).
2. Right-click the **zynq_fsbl** project and select **Build Configurations > Set Active > linux**.



3. Optional: If you want to use your own Hardware Platform Specification and BSP projects instead of the imported ones, you need to change the referenced BSP first (see steps in Section 6.3).

4. Right-click the **zynq_fsbl** project and select **Build Project**.



5. Repeat the previous two steps with the active configuration set to standalone.

21.7.7 6.7 Creating a Standalone Boot Image

Creating a standalone boot image requires the following components:

zynq_fsbl.elf	FSBL - standalone configuration (6.6)

sobel.bit	Full bitstream - sobel configuration (4.8)
filter_std.elf	Standalone software application (6.3)

All required files to build the boot image are available at zc702_pr_rd/sw/boot/standalone.

Shortcut: A pre-generated standalone boot image is available at zc702_pr_rd/sd/standalone/BOOT.bin.

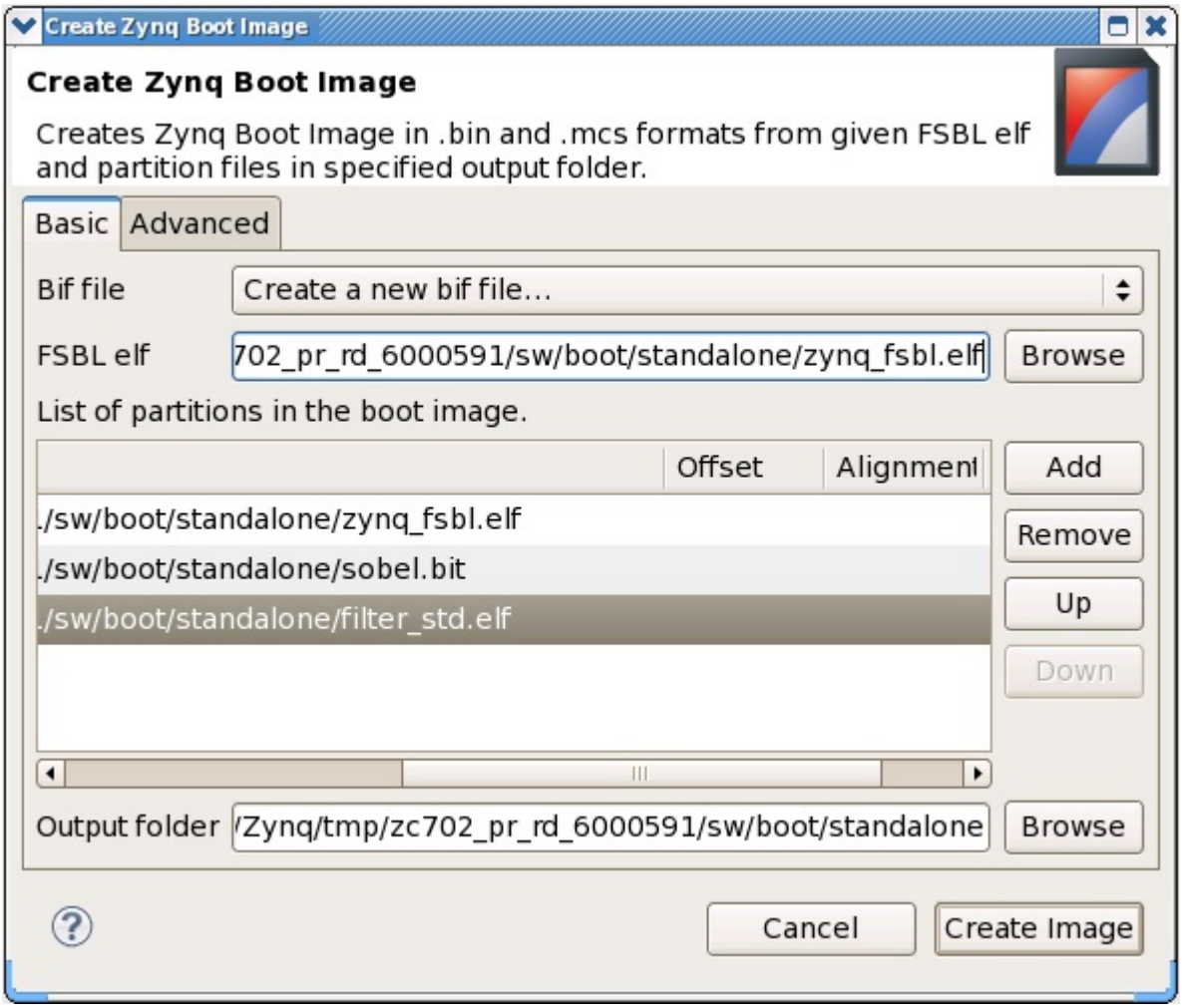
Tutorial

---

1. From the menu bar, select **Xilinx Tools > Create Zynq Boot Image**.

2. In the **Create Zynq Boot Image** dialog, next to the **FSBL elf** text field, **Browse** to the zc702_pr_rd/sw/boot/standalone directory, select zynq_fsbl.elf, and click **OK**.
3. Next to the **List of partitions in the boot image** text field, click **Add**, browse to the zc702_pr_rd/sw/boot/standalone directory, select sobel.bit, and click **OK**.
4. Click **Add** again, browse to the zc702_pr_rd/sw/boot/standalone directory, select filter_std.elf and click **OK**.
5. Next to the **Output folder** text field, **Browse** to the zc702_pr_rd/sw/boot/standalone directory and click **OK**.

6. Click **Create Image**.



7. Navigate to the zc702_pr_rd/sw/boot/standalone directory and rename the generated boot image from filter_std.bin to BOOT.bin.

21.7.8 6.8 Creating a Linux Boot Image

Creating a Linux boot image requires the following components:

zynq_fsbl.elf	FSBL - linux configuration (6.6)
sobel.bit	Full bitstream - sobel configuration (4.8)

u-boot.elf	u-boot - second stage boot loader (5.1)

All required files to build the boot image are available at zc702_pr_rd/sw/boot/linux.

Shortcut: A pre-generated Linux boot image is available at zc702_pr_rd/sd/linux/BOOT.bin.

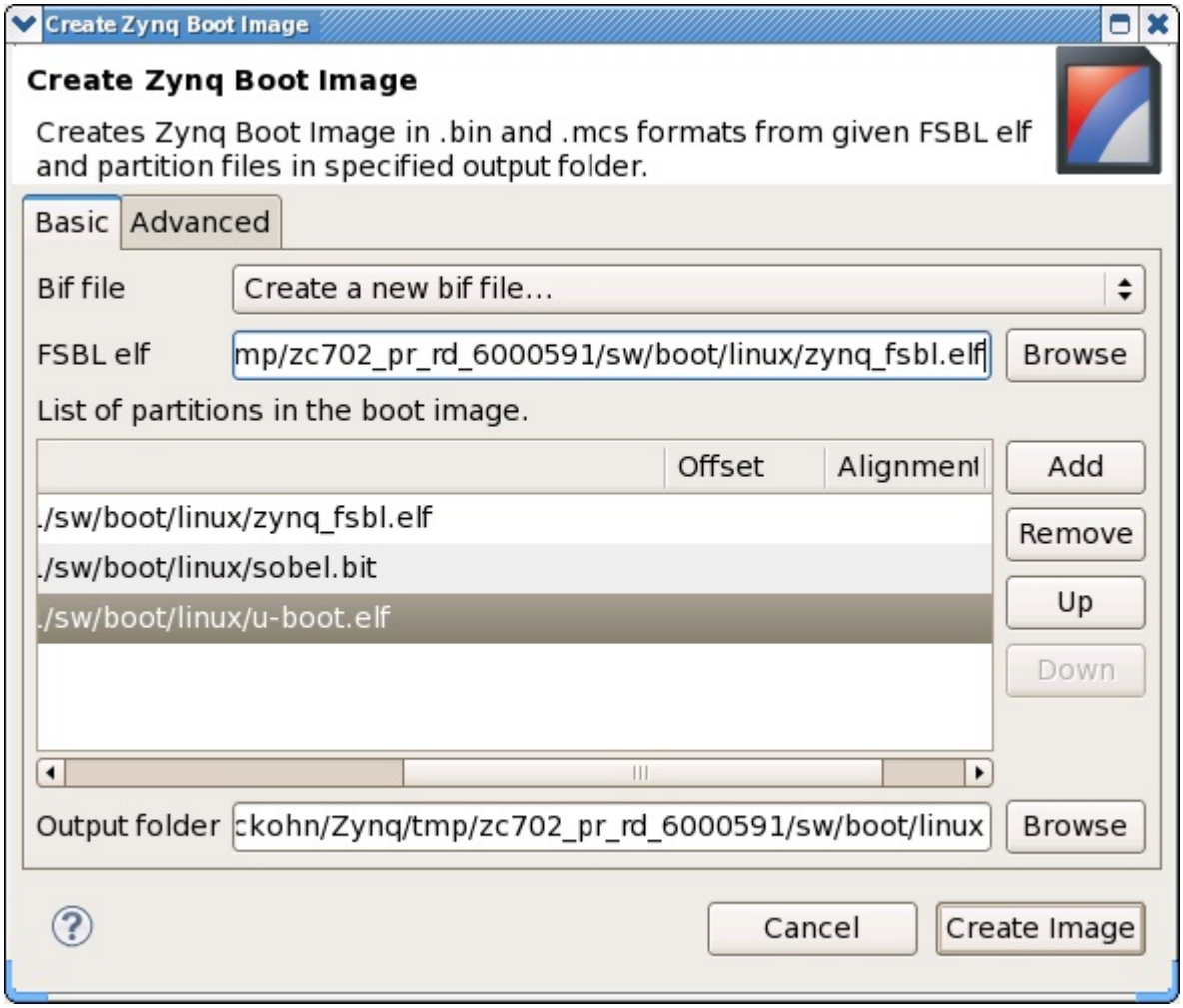
Tutorial

---

1. From the menu bar, select **Xilinx Tools > Create Boot Image**.

2. In the **Create Zynq Boot Image** dialog, next to the **FSBL elf** text field, **Browse** to the zc702_pr_rd/sw/boot/linux directory, select zynq_fsbl.elf, and click **OK**.
3. Next to the **List of partitions in the boot image** text field, click **Add**, browse to the zc702_pr_rd/sw/boot/linux directory, select sobel.bit, and click **OK**.
4. Click **Add** again, browse to the zc702_pr_rd/sw/boot/linux directory, select u-boot.elf and click **OK**.
5. Next to the **Output folder** text field, **Browse** to the zc702_pr_rd/sw/boot/linux directory and click **OK**.

6. Click **Create Image**.



7. Navigate to the zc702_pr_rd/sw/boot/linux directory and rename the generated boot image from u-boot.bin to BOOT.bin.

21.8 7 Running the Reference Design in Hardware

This section describes how to set up the ZC702 board and additionally required hardware, create standalone and Linux SD-card images based on the provided reference design, boot up the device using SD boot mode, and finally run and operate the different software applications. The reference design supports two video resolutions: 1080p (default) and 720p at 60 frames per second. Note that the video input and video output resolutions need to match. Section 6.3 shows how to change the resolution of the standalone application; Section 7.3 shows how to change the video resolution in the Linux devicetree.

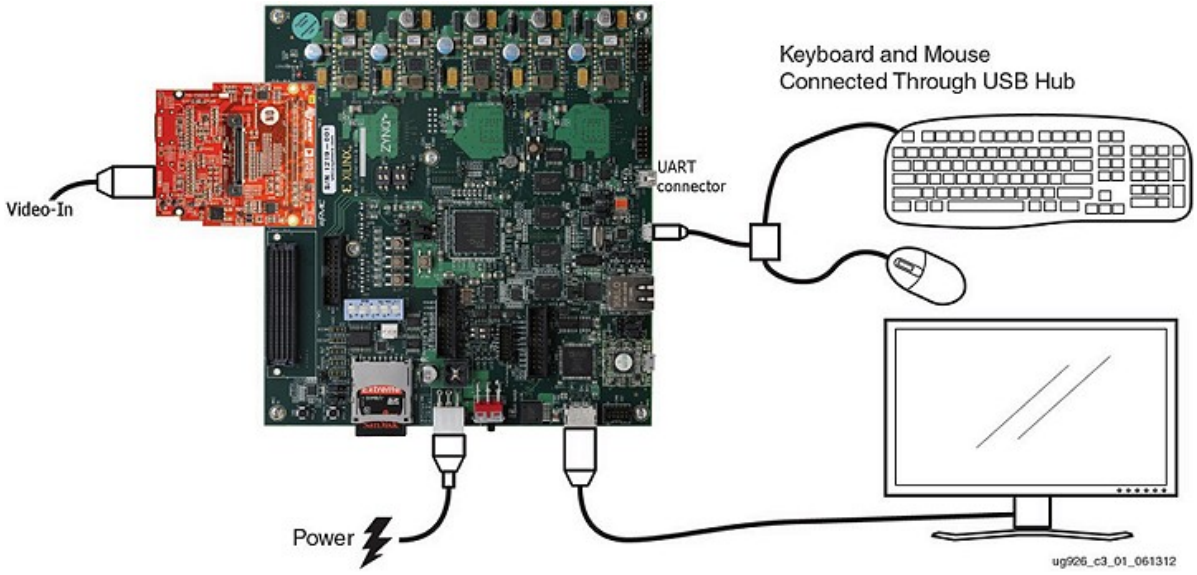
21.8.1 7.1 Setting up the ZC702 Evaluation Board

Refer to UG926 , Chapter 1 for kit contents and default jumper and switch settings. Reference design specific settings are described below:

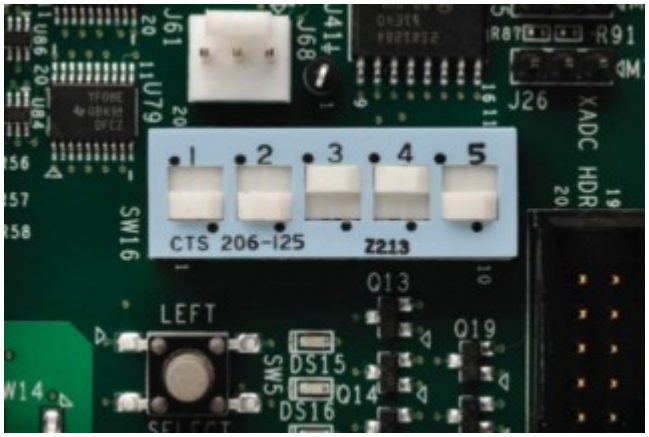
Tutorial

---

1. Connect a power supply to connector J60 on the ZC702 board.
2. Connect a 1080p60-capable monitor to the HDMI port on the ZC702 board.
3. Connect a USB keyboard and USB mouse to a USB hub, which in turn is connected to the Micro USB port J1 on the ZC702 board (Not required for standalone or Linux command line application).
4. Connect the FMC-IMAGEON daughter card to the FMC2 slot on the ZC702 board and an HDMI cable to the HDMI IN port on the FMC card to enable external video input (Optional).



5. Set the boot mode switch SW16 to SD boot mode.



6. Connect a Laptop or PC to the Mini USB port J17 on the ZC702 board for UART communication (Optional for Linux Qt application). Use the following serial port settings:
   Baud Rate: 115200
   Data: 8 bit
   Parity: None
   Stop: 1 bit
   Flow Control: None

21.8.2 7.2 Running the Standalone Software Application

The standalone SD-card image requires the following components:

BOOT.bin	Standalone boot image (6.7)
sepia.bin	Partial Sepia bitstreams (4.9)
sobel.bin	Partial Sobel bitstreams (4.9)

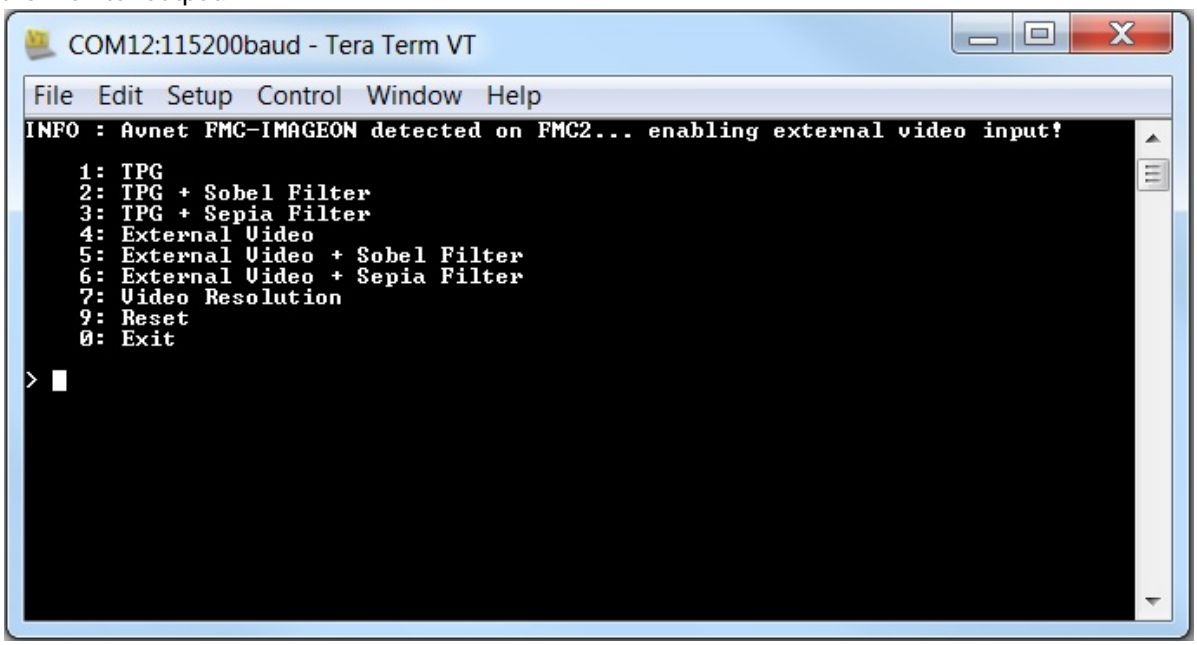
Shortcut: A pre-built standalone SD-card image is available at zc702_pr_rd/sd/standalone.

Tutorial

---

1. Copy the standalone SD-card image to the top-level directory of a FAT-32 formatted SD card.

2. Insert the card into the SD card slot on the ZC702 board.
3. Power on the ZC702 board.
4. Connect your terminal emulator software (e.g. TeraTerm) to the serial port assigned to the Silicon Labs CP210x USB to UART Bridge.
5. The standalone application will start automatically.
6. To operate the standalone application, enter the desired number in the terminal window and observe the monitor output.



21.8.3 7.3 Running the Linux Software Application(s)

The Linux SD-card image requires the following components:

BOOT.bin	Linux boot image (6.8)
devicetree.dtb	Device tree blob (5.2)
devicetree.dts	Device tree source (5.2)
filter_cmd	Linux command line application (6.4)

filter_qt	Linux Qt application (6.5)
images/	Directory with pictures used by Linux Qt application (6.5)
init.sh	Setup script (5.3)
qt_lib.img	Qt/Qwt libraries image file (6.5)
run_filter.sh	Wrapper script (7.3)
sepia.bin	Partial Sepia bitstreams (4.9)
sobel.bin	Partial Sobel bitstreams (4.9)
uImage	Linux kernel image (5.2)
uramdisk.image.gz	Linux root file system (5.3)

Shortcut: A pre-built Linux SD-card image is available at zc702_pr_rd/sd/linux.

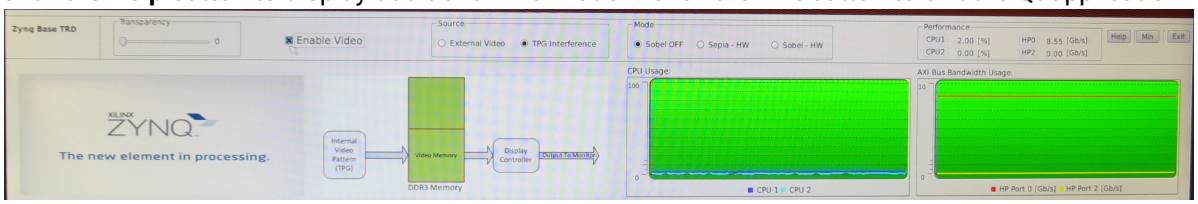
Tutorial

---

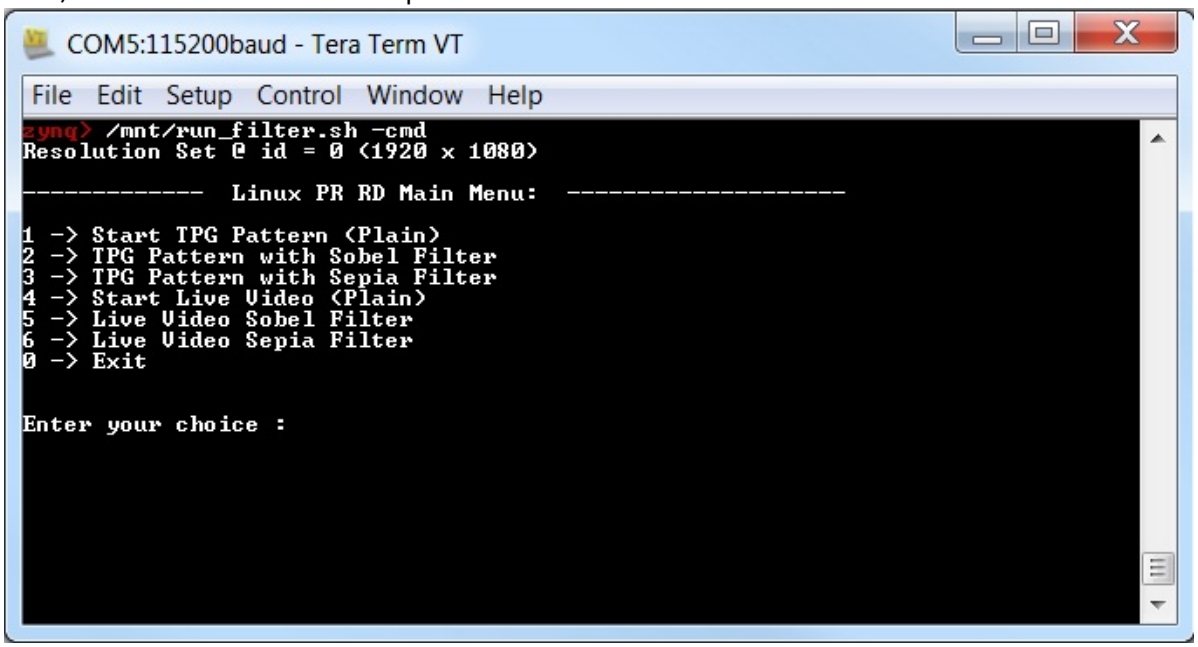
1. Copy the Linux SD-card image to the top-level directory of a FAT-32 formatted SD card.
2. Insert the card into the SD card slot on the ZC702 board.
3. Power on the ZC702 board.
4. Connect your terminal emulator software (e.g. TeraTerm) to the serial port assigned to the Silicon Labs CP210x USB to UART Bridge.
5. The Qt application will start automatically after Linux is booted.
6. Use the mouse to navigate the Qt GUI. The GUI has the following features (from left to right):
   Transparency slider
   Enable Video button
   Source Select radio buttons
   Filter Mode radio buttons
   Textual **Performance Monitor**: CPU usage and AXI HP0/2 memory throughput
   Help, **Min**, **Exit** control buttons
   Zynq logo
   Data flow graph based on Source and Mode selection

Graphical **CPU usage** monitor
Graphical **AXI HP0/2 memory throughput** monitor

7. Click the **Min** button to hide the Zynq logo, Data flow graph, and the graphical CPU / AXI HP monitors. Click the **Help** button to display additional information. Click the **Exit** button to exit the Qt application.

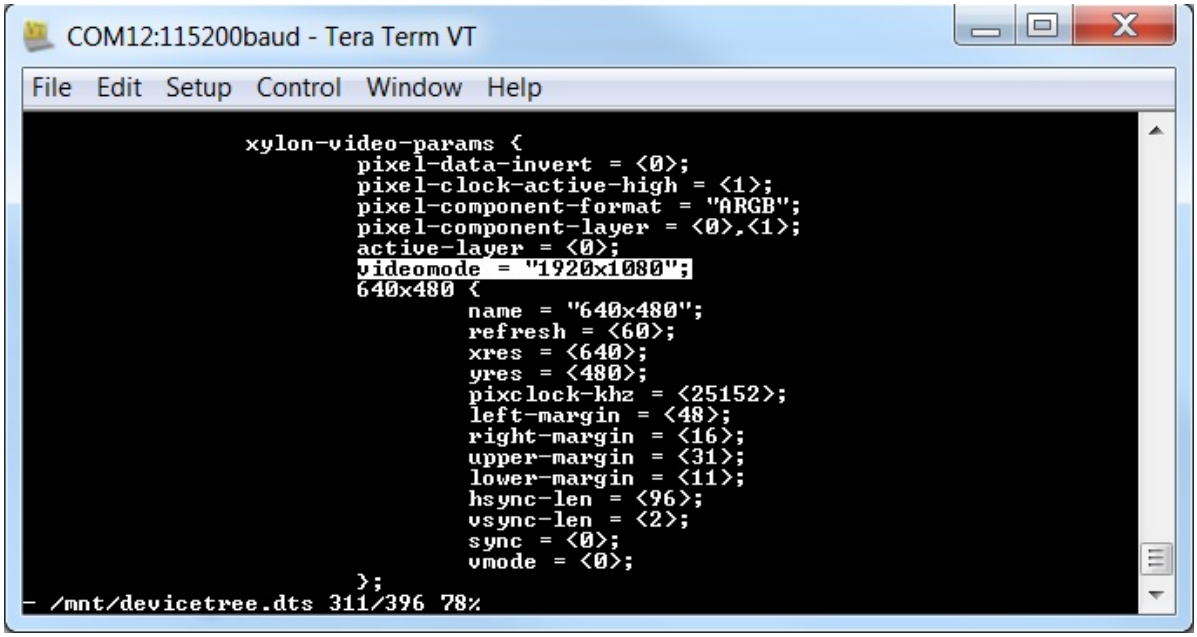


8. To restart the Qt application, change directory to /mnt, then type ./run_filter.sh -qt at the zynq prompt in either the terminal window or on the monitor display. Alternatively, to start the command line application, type ./run_filter.sh -cmd at the zynq prompt in the terminal window. The file run_filter.sh is a wrapper script that runs the corresponding Qt or command line executable and sets the required Qt environment variables.
9. To operate the command line application, enter the desired number in the terminal window, press enter, and observe the monitor output.



10. The default video resolution is set to 1080p. To change the video resolution to 720p, use a text editor (e.g. vi) to open the file /mnt/devicetree.dts. Search for the line starting with videomode and set the

resolution to "1280x720".



11. Save the file.
12. Re-compile the devicetree by typing dtc -I dts -O dtb -o /mnt/devicetree.dtb /mnt/devicetree.dts at the zynq prompt.
13. Type halt at the zynq prompt to shutdown Linux.
14. Power-cycle the ZC702 board or press the POR (power-on-reset) button. The new video resolution will be active after re-boot.

References:

1. XAPP1159,Partial Reconfiguration of a Hardware Accelerator on Zynq-7000 SoC Devices
2. UG925Zynq-7000 EPP ZC702 Base Targeted Reference Design
3. XAPP890,Zynq SoC Sobel Filter Implementation Using the Vivado HLS Tool
4. UG821,Zynq-7000 EPP Software Developers Guide
5. UG926,Zynq-7000 EPP ZC702 Evaluation Kit

Additional Documentation:

1. UG744Partial Reconfiguration of a Processor Peripheral Tutorial
2. UG873 Zynq-7000 SoC Concepts, Tools and Techniques

22 Data Movers

22.1 Data Movers

This page contains all Data Movers resources for Zynq-7000 SoC.

22.2 Resources

Zynq Training
Xilinx Zynq-7000 SoC Solution Center

22.3 User Guides

Zynq-7000 SoC Technical Reference Manual

22.4 App Notes & Reference Designs & White Papers

Zynq-7000 SoC- Where can I find Data Mover Examples?
- Overview of Zynq-7000 SoC data mover examples

Zynq-7000 SoC ZC702 Base Targeted Reference Design
- Functional description of the ZC702 base targeted reference design, including IP/logic implemented in programmable logic (PL) and base TRD package directory structure.

PCIe Targeted Reference Design
- Summarizes the PCIe TRD modes of operation and features including PCIe connectivity and Cortex A9 processing and offload.

System Monitoring using the Zynq-7000 SoC Processing System with the XADC AXI Interface
- App Note describes how a Xilinx XADC can be used for system monitoring applications.

Implementing Analog Data Acquisition using the Zynq-7000 SoC Processing System with the XADC AXI Interface
- App Note describes how the Xilinx XADC acquires analog data using its dedicated Vp/Vn analog input.

Zynq-7000 SoC Accelerator For Floating-Point Matrix Multiplication using Vivado HLS
- App Note describes how to use Vivado HLS to develop a floating-point matrix multiplication accelerator with an AXI4-Stream interface and connect it to the ACP of the ARM CPU.

PCI Express Endpoint-DMA Initiator Subsystem

- App Note demonstrates Vivado subsystem for endpoint-initiated DMA data transfers through PCI Express.

Partial Reconfiguration of a Hardware Accelerator on Zynq-7000 SoC Devices
- App Note describes the tool flow, concepts and techniques for using partial reconfiguration on Zynq-SoC through the DevC and PCAP.

PS and PL Ethernet Performance and Jumbo Frame Support with PL Ethernet in the Zynq-7000 SoC
- App Note describes using the PS based gigabit Ethernet MAC (GEM) through the EMIO interface with the 1000BASE-X physical interface using high speed serial transceivers in the PL.

Zynq-7000 Example Design - Cache coherent CDMA transfers from block RAM to OCM
- This example design allocates 4K of block RAM attached to the CPU via M_AXI_GP0

22.5 Tech Tips & How To's

Zynq-7000 SoC - Precision Timing with IEEE1588 v2 Protocol Tech Tip

Related Links

- Zynq-7000 SoC Getting Started
- Zynq-7000 SoC Demonstrations & Boards and Kits
- Zynq-7000 SoC Development & Debug
- Zynq-7000 SoC Boot & Config
- Zynq-7000 SoC Operating Systems
- Zynq-7000 SoC Power Management
- Zynq-7000 SoC Performance and Benchmarks
- Zynq-7000 SoC Software Acceleration
- Zynq-7000 SoC Real-Time & Data Movement
- Zynq-7000 SoC Peripherals, IP & Drivers
- Zynq-7000 SoC Security & Safety

23 Programming QSPI from U-boot ZC702

In this article, we shall be discussing how to program the QSPI from the U-boot running on the Cortex A9 on Xilinx ZC702 Development board. Here, we will show how to build the uboot executable, and how

to configure the Zynq Processing Sub-system (PS), place the Image into DDR and boot uboot via XSCT in JTAG. Finally, how to use the uboot commands to program the image from DDR into QSPI

Table of Contents

23.1 Step 1: Building the U-boot executable:

The recommended flow when creating any OS image is to use the Petalinux tool. However, here we shall be obtaining the xilinx branch of the u-boot from github and compiling
manually. For the complete OSL flow see the article here

Note: The uboot uses the Devicetre Complier (DTC) during compilation, so this is needed too. Also, the arm-xilinx-linux-gnueabi- compiler is needed too.

```
git clone https://git.kernel.org/pub/scm/utils/dtc/dtc.git
cd dtc
make
Add this to your PATH. For example
export PATH=$PATH:/<add the path here>/dtc/dtc
To test try dtc -help.
cd .. git clone git://github.com/Xilinx/u-boot-xlnx.git cd u-boot-xlnx
export CROSS_COMPILE=arm-xilinx-linux-gnueabi-
make zynq_zc702_config
make
```

The u-boot (to be renamed u-boot.elf) will be placed at u-boot-xlnx directory.

23.2 Step 2: Creating XSCT script:

```
connect
source ps7_init.tcl
targets -set -filter {name =~ "APU"}
ps7_init
ps7_post_config
targets -set -filter {name =~ "ARM Cortex-A9 MPCore #0"}
dow -data BOOT.BIN 0x08000000
dow u-boot.elf
con
```

Copy the contents above into a TCL file, and source this from XSCT (This is a SDK utility).
Note: I used the ps7_init.tcl. This can be generated in the SDK
This will boot uboot on a serial port (baud 15200)

23.3 Step 3: Using U-boot commands to program the QSPI

```
sf probe 0 0 0
sf erase <image file size in bytes (hex)>
sf write 0x08000000 <offset in hex> <image file size in bytes (hex)>
```

Change the Boot Mode and POR_B to test.

23.4 Related Links

- UBOOT & http://www.wiki.xilinx.com/U-boot
- Building Uboot & http://www.wiki.xilinx.com/Build+U-Boot

24 Zynq Ethernet Performance

Table of Contents

24.1 XAPP1082 Benchmarking Results

This section includes Zynq Ethernet Performance associated with XAPP1082 releases.

24.1.1 XAPP1082 v3.0

2014.4
2015.1
2015.2
2015.3

24.1.2 XAPP1082 v4.0

2015.4
2016.1
2016.2
2016.3
2017.2
2017.4

24.2 Understanding Ethernet Performance

The raw line rate of Ethernet is 1.25Gbps. This is commonly referred to as Gigabit Ethernet after accounting for encoding overheads. The performance of various Ethernet applications is at different layers is lesser than the throughput seen at the software driver or at the Ethernet interface. This is due to the additional headers and trailers inserted in packet by each layer of the networking stack. Ethernet is only a medium to carry traffic; various protocols like TCP or UDP implement their own header formats.

To estimate theoretical Ethernet performance, following is considered-

- Ethernet Overhead : 14B Ethernet Header + 4B Ethernet Trailer (FCS)
- 8B of preamble with 12B of Inter Frame Gap on wire
- 20B TCP Header + 20B IP Header

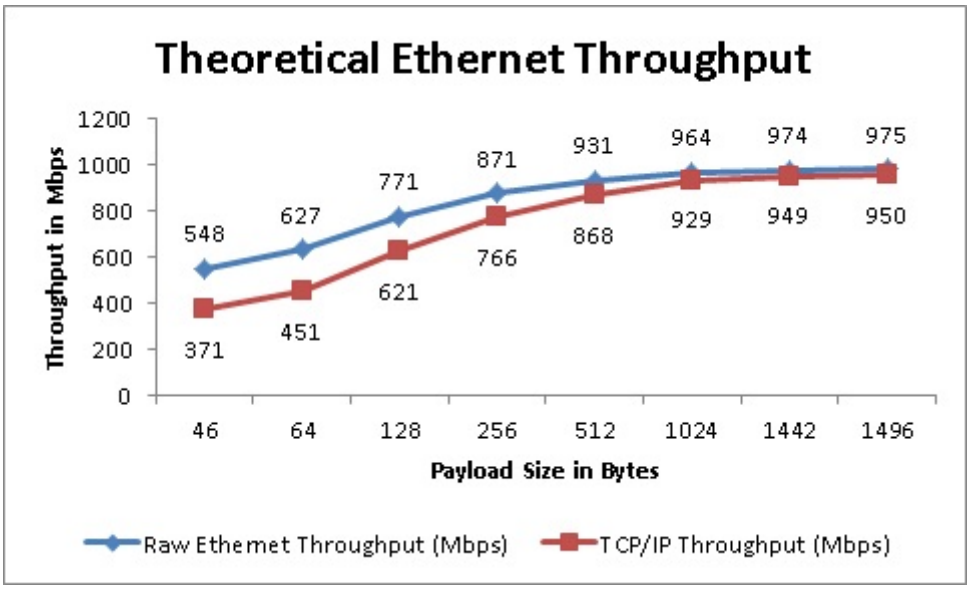
Based on this we have,

Raw Ethernet Throughput =	*Payload * 1000*
	(*Payload + 18 + 20*)

TCP/IP Frame Throughput =	*Payload * 1000*
	(*Payload + 18 + 20 + 40*)

Plotting this for a few payload size values as shown below, we notice

1. Protocol overheads prevent the network cards from reaching wire speed
2. Throughput improves with payload size; with increasing payload, impact of a fixed size header/trailer reduces



Note: This estimate doesn't account for software stack latencies.

Ethernet Benchmarking

This section describes Ethernet benchmarking results obtained with netperf.

24.2.1 Using Netperf

Netperf provides a network benchmarking tool which can measure throughput and also report CPU utilization.
To use netperf on Zynq Linux, the netperf source can be downloaded and built for ARM Linux using cross-compiler tool chain.
Following netperf commands are useful for benchmarking-

For Zynq as server- (refer XAPP1082 for note on CPU affinity)

```
bash> taskset 2 ./netserver
```

For peer PC as client-

The options '-c' and '-C' report CPU utilization for self and remote CPUs.

```
bash> netperf -H <Zynq_IP_address> -c -C
bash> netperf -H <Zynq_IP_address> -c -C -t TCP_MAERTS
bash> netperf -H <IP address> -c -C -- -m <64|128|256|512|1024> -D
```

The first command generates traffic from peer PC towards Zynq - essentially measuring Zynq receive performance.
This runs the TCP_STREAM test by default.
The second command running TCP_MAERTS test measures Zynq transmit performance.
The third command provides option for message size variation with '-D' option.

It should be noted that message size in netperf refers to size of TCP payload in Ethernet packet - the total size of packet on wire will include additional overheads associated with various layers. Please note that results provided (in v1.0 of XAPP1082) are run without -D option where Linux coalesces multiple smaller messages into larger TCP segment thereby showing improved performance at smaller message size values; if actual payload size variation is required for TCP tests, *netserver* and *netperf* should be run with an additional '-D' switch along with message size variation (this is updated in v2.0 of XAPP1082).
Use of '-D' switch disables Nagle's algorithm; however TCP being a byte stream service some additional options like changing socket size using -s option may be required to ensure smaller size frames on wire.
Benchmarking is done under near ideal conditions where no other applications are run to interfere with results.

24.2.2 Impact of Checksum Offload

Checksum is used to maintain data integrity in TCP/UDP protocols. Normally, this checksum calculation is handled by protocol stack which consumes significant processor bandwidth. This operation tends to be slower when operating larger size Ethernet frames at high line rate. Checksum offloading moves this checksum calculation in outbound (transmit) direction and checksum verification for inbound (receive) direction to hardware. This frees up processor for use in other functions.

It is noticed that,

1. Checksum Offload improves throughput
2. Checksum Offload improves CPU utilization

24.3 XAPP1082 - Zynq-7000 Ethernet Performance

Table of Contents

Benchmarking results are available at http://www.wiki.xilinx.com/Zynq+Ethernet+Performance

24.3.1 1. Introduction

The focus of this application note is on Ethernet peripherals in the Zynq®-7000 SoC. This application note describes using the processing system (PS) based gigabit Ethernet MAC (GEM) through the extended multiplexed I/O (EMIO) interface with the 1000BASE-X physical interface using high-speed serial transceivers in programmable logic (PL). This application note also describes the implementation of PL-based Ethernet supporting jumbo frames.

The designs provided with this application note enable the use of multiple Ethernet ports, and provide kernel-mode Linux device drivers. In addition, this document includes Ethernet performance measurements with

and without checksum offload support enabled.

This page discusses the following-

1. Hardware and software design build steps
2. Understanding & Benchmarking Ethernet performance

The design details (block diagrams) are provided in XAPP1082

24.3.2 2. XAPP1082 v 4.0

XAPP1082 v4.0 introduces:-

- SGMII support to PS-EMIO and PL-Ethernet designs
- Supports Vivado 2017.4 (IPI Design)
- Petalinux 2017.4 SDK
- macb driver support
- Xilinx PHY driver supports for 1000Base-X and SGMII

Four designs are described in this application note. The designs support Vivado IP Integrator tool flow.

a) PS Ethernet (GEM1) that is connected to a 1000BASE-X physical interface in PL through an EMIO interface.
b) PL Ethernet implemented as soft logic in PL and connected to the 1000BASE-X physical interface in PL.
c) PS Ethernet (GEM1) that is connected to a SGMII physical interface in PL through an EMIO interface.
d) PL Ethernet implemented as soft logic in PL and connected to the SGMII physical interface in PL.

The steps for building designs "a" and "b" mentioned above, are same as in "Vivado" section of XAPP v3.0.
The steps for building designs "c" and "d" are mentioned below.

24.3.2.1 Building PS-EMIO design in SGMII mode

To rebuild the hardware design, execute the following (after setting up Vivado environment).
1. Open a Linux terminal or Vivado tcl shell in windows
2. Navigate to hardware/vivado/scripts/ps_emio_eth for PS EMIO Ethernet design
$ vivado -source ps_emio_eth_sgmii.tcl

Rest of the steps remain same as covered in section XAPP v3.0.

24.3.2.2 Building PL Ethernet design in SGMII mode

To rebuild the hardware design, execute the following (after setting up Vivado environment).

1. Open a Linux terminal or Vivado tcl shell in windows
2. Navigate to hardware/vivado/scripts/pl_eth for PL Ethernet design

$ vivado -source pl_eth_sgmii.tcl

Rest of the steps remain same as covered in section XAPP v3.0.

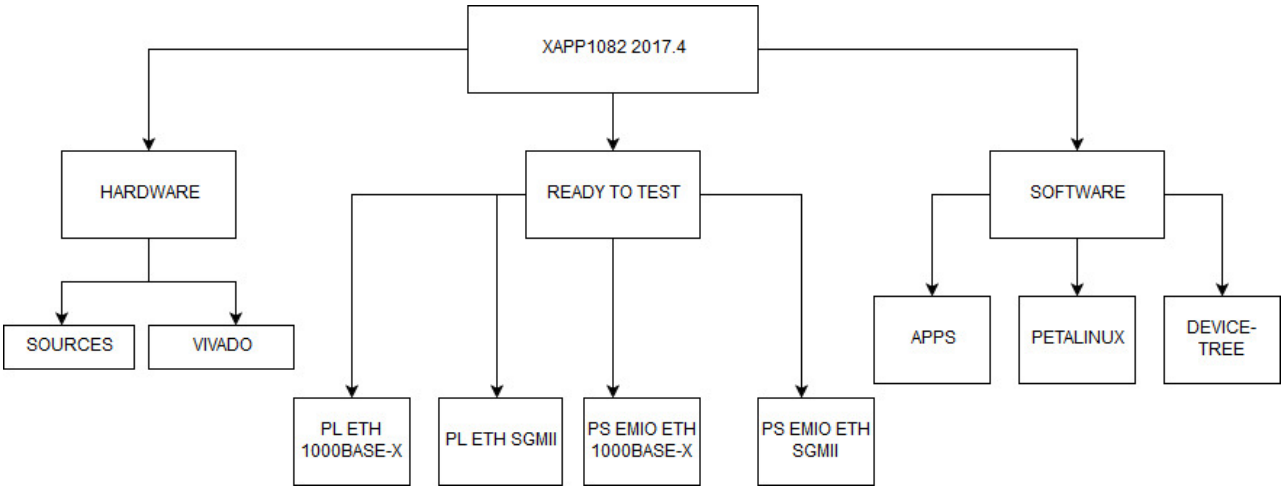
24.3.2.3 2.1 PetaLinux Installation

Prerequisites
This section lists the requirements for the PetaLinux Tools Installation

- Download Petalinux 2017.4 SDK software from Xilinx website download section.
- Refer to section 3.4 for PetaLinux installation instructions.

24.3.2.4 2.2 Directory structure

The XAPP1082 4.0 is released with the source code, Xilinx Vivado and Petalinux projects and an SD card image that enables the user to run the demonstration.
It also includes the binaries necessary to configure and boot the Zynq-7000 SoC board.
This wiki page assumes the user has already downloaded the XAPP package and extracted its contents to the XAPP home
directory referred to as XAPP_HOME in this wiki.



24.3.2.5 2.3 PS-EMIO Ethernet

24.3.2.5.1 2.3.1 PS-EMIO BSP installation for 1000Base-X

PS-EMIO Ethernet project provides installable BSP, which includes all necessary design sources and configuration files, including pre-built and tested hardware and software images, ready for download to your board or for booting in the QEMU system simulation environment.

24.3.2.5.1.1 2.3.1.1 Create PS EMIO Ethernet project from PetaLinux BSP

Run petalinux-create command on the console
petalinux-create -t project -s <path-to-bsp>

```
bash> cd $PETALINUX
bash> petalinux-create -t project -s $XAPP_HOME/software/petalinux/bsp/
xapp1082_ps_emio_eth_1000x.bsp
```

24.3.2.5.1.2 2.3.1.2 Configure PetaLinux

```
bash> cd xapp1082_ps_emio_eth
bash> petalinux-config
```

Save the changes and exit.
It will download the remote kernel and checkout xilinx-v2017.4 tag.
NOTE: Above step may take a longer time depending on the network bandwidth.

24.3.2.5.1.3 2.3.1.3 Configure the kernel

Check the Xilinx PHY driver from kernel configuration

```
bash> petalinux-config -c kernel
```

Device Drivers> Network device support > PHY Device support and infrastructure >
<*> Drivers for xilinx PHYs
Save the changes and exit.

24.3.2.5.1.4 2.3.1.4 Apply FSBL patch

Refer to the AR 66006 for configuring the SFP and SI5324 using I2C in FSBL.
Also user can copy the files present in fsbl_patch_files folder to configure the clock and SFP.

```
bash> cd components/bootloader/zynq_fsbl
bash> cp -f $XAPP_HOME/software/petalinux/fsbl_patch_files/* .
bash> cd -
```

24.3.2.5.1.5 2.4.1.5 Modifications to system-user.dtsi

For PS EMIO 1000BASE-X ETHERNET,

Modify the system-user.dtsi by navigating to the the file using 'vi project-spec/meta-user/recipes-bsp/device-tree/files/system-user.dtsi',

```
/include/ "system-conf.dtsi"
/ {
};
&gem1 {
local-mac-address = [00 0a 35 00 00 01];
phy-mode = "rgmii-id";
status = "okay";
xlnx,ptp-enet-clock = <0x69f6bcb>;
phy-handle = <&phy1>;

phy1: phy@1 {
compatible = "Xilinx PCS/PMA PHY";
device_type = "ethernet-phy";
xlnx,phy-type = <5>;
reg = <1>;
};
};
```

Save and exit.

24.3.2.5.1.6 **2.3.1.6 Build**

Build images using PetaLinux

```
bash> petalinux-build -v
```

24.3.2.5.1.7 **2.3.1.7 Create Zynq Boot image (BOOT.bin)**

```
bash> cd images/linux
bash> petalinux-package --boot --fsbl zynq_fsbl.elf --fpga ps_emio_sfp.bit --u-boot
(1000base-x)
```

24.3.2.5.1.8 **2.3.1.8 SD Images**

SD Deployable binaries:-
a) BOOT.bin
b) image.ub
Copy BOOT.BIN and image.ub from $PETALINUX/ xapp1082_ps_emio_eth_1000x/images/linux to SD partition and run the setup.

24.3.2.5.2 2.3.2 PS-EMIO BSP installation for SGMII

PS-EMIO Ethernet project provides installable BSP which includes all necessary design sources and configuration files, including pre-built and tested hardware and software images, ready for download to your board or for booting in the QEMU system simulation environment.
The design supports with the auto-negotiation for speeds of 10/100/1000 Mbps and full duplex mode.

24.3.2.5.2.1 2.3.2.1 Create PS EMIO Ethernet project from PetaLinux BSP

Run petalinux-create command on the command console
petalinux-create -t project -s <path-to-bsp>

```
bash> cd $PETALINUX
bash> petalinux-create -t project -s $XAPP_HOME/software/petalinux/bsp/
xapp1082_ps_emio_eth_sgmii.bsp
```

24.3.2.5.2.2 2.3.2.2 Configure Petalinux

```
bash> cd xapp1082_ps_emio_eth
bash> petalinux-config
```

Save the changes and exit.
It will download the remote kernel and checkout xilinx-v2017.4 tag.
NOTE: Above step may take a longer time depending on the network bandwidth.

24.3.2.5.2.3 2.3.2.3 Configure the kernel

Check the Xilinx PHY driver from kernel configuration

```
bash> petalinux-config -c kernel
```

Device Drivers> Network device support > PHY Device support and infrastructure >
<*> Drivers for xilinx PHYs
Save the changes and exit.

24.3.2.5.2.4 2.3.2.4 Apply FSBL patch

Refer to the AR 66006 for configuring the SFP and SI5324 using I2C in FSBL.
Also user can copy the files present in fsbl_patch_files folder to configure the clock and SFP.

```
bash> cd components/bootloader/zynq_fsbl
bash> cp -f $XAPP_HOME/software/petalinux/fsbl_patch_files/* .
bash> cd -
```

24.3.2.5.2.5 **2.4.2.5 Modifications to system-user.dtsi**

For PS EMIO SGMII ETHERNET,
Modify the system-user.dtsi by navigating to the the file using 'vi project-spec/meta-user/recipes-bsp/device-tree/files/system-user.dtsi',

/include/ "system-conf.dtsi"
/ {
};
&gem1 {
local-mac-address = [00 0a 35 00 00 01];
phy-mode = "rgmii-id";
status = "okay";
xlnx,ptp-enet-clock = <0x69f6bcb>;
phy-handle = <&phy1>;

phy1: phy@1 {
compatible = "Xilinx PCS/PMA PHY";
device_type = "ethernet-phy";
xlnx,phy-type = <4>;
reg = <1>;
};
};

Save and exit.

24.3.2.5.2.6 **2.3.2.6 Build**

Build images using PetaLinux

```
bash> petalinux-build -v
```

24.3.2.5.2.7 **2.3.2.7 Create Zynq Boot image (BOOT.bin)**

```
bash> cd images/linux
bash> petalinux-package --boot --fsbl zynq_fsbl.elf --fpga ps_emio_sfp,bit --u-boot
(sgmii)
```

24.3.2.5.2.8 **2.3.2.8 SD Images**

SD Deployable binaries:-
a) BOOT.bin
b) image.ub
Copy BOOT.BIN and image.ub from $PETALINUX/ xapp1082_ps_emio_eth_sgmii/images/linux to SD partition and run the setup.

24.3.2.6 2.4 PL Ethernet

24.3.2.6.1 2.4.1 PL Ethernet BSP installation for 1000Base-X

PL Ethernet project provides installable BSP which includes all necessary design sources and configuration files, including pre-built and tested hardware and software images, ready for download to your board or for booting in the QEMU system simulation environment.
NOTE : Check-sum offload in enabled in the default configuration.

24.3.2.6.1.1 **2.4.1.1 Create PL Ethernet project from petalinux installable BSP**

Run petalinux-create command on the command console:
petalinux-create -t project -s <path-to-bsp>

```
bash> cd $PETALINUX
bash> petalinux-create -t project -s $XAPP_HOME/software/petalinux/bsp/
xapp1082_pl_eth_cso_1000x.bsp
```

24.3.2.6.1.2 **2.4.1.2 Configure petalinux**

```
bash> cd xapp1082_pl_eth_1000x
bash> petalinux-config
```

Save the changes and exit.
It will download the remote kernel and checkout xilinx-v2017.4 tag.
NOTE: Above step may take a longer time depending on the network bandwidth.

24.3.2.6.1.3 2.4.1.3 Configure the kernel

```
bash> petalinux-config -c kernel
```

Enable the Xilinx PHY driver and Disable the AXI DMA driver
Device Drivers> Network device support > PHY Device support and infrastructure >
<*> Drivers for xilinx PHYs
Device Drivers> DMA Engine Support> Xilinx DMA Engines >
<> Xilinx AXI DMA Engine
Save the changes and exit.

24.3.2.6.1.4 2.4.1.4 Apply FSBL patch

Refer to the AR 66006 for configuring the SFP and SI5324 using I2C in FSBL
Also user can copy the files present in fsbl_patch_files folder to configure the clock and SFP for SGMII.

```
bash> cd components/bootloader/zynq_fsbl
bash> cp -f $XAPP_HOME/software/petalinux/fsbl_patch_files/* .
bash> cd -
```

24.3.2.6.1.5 2.4.1.5 Modifications to system-user.dtsi

For PL ETHERNET 1000base-x,

Modify the system-user.dtsi by navigating to the the file using 'vi project-spec/meta-user/recipes-bsp/device-tree/files/system-user.dtsi',

/include/ "system-conf.dtsi"
/ {
};

&axi_ethernet {
local-mac-address = [00 0a 35 00 1e 52];
};

Save and exit.

24.3.2.6.1.6 2.4.1.6 Build

Build all images using PetaLinux

```
bash> petalinux-build -v
```

24.3.2.6.1.7 2.4.1.7 Create Zynq Boot image (BOOT.bin)

```
bash> cd images/linux
bash> petalinux-package --boot --fsbl zynq_fsbl.elf --fpga pl_eth_sfp.bit --u-boot
(1000base-x)
```

24.3.2.6.1.8 2.4.1.8 SD Images

SD Deployable binaries:-
a) BOOT.bin
b) image.ub
Copy BOOT.BIN and image.ub from $PETALINUX/ xapp1082_pl_eth_1000x/images/linux to SD partition and run the setup.
Note: Instructions on Petalinux BSP creation (rather Petalinux flow from scratch) is provided in Appendix1 below.

24.3.2.6.2 2.4.2 PL Ethernet BSP installation for SGMII

PL Ethernet project provides installable BSP which includes all necessary design sources and configuration files, including pre-built and tested hardware and software images, ready for download to your board or for booting in the QEMU system simulation environment.
The design supports with the auto-negotiation for speeds of 10/100/1000 Mbps and full duplex mode.
NOTE : Check-sum offload in enabled in the default configuration.

24.3.2.6.2.1 2.4.2.1 Create PL Ethernet project from PetaLinux installable BSP

Run petalinux-create command on the command console:
petalinux-create -t project -s <path-to-bsp>

```
bash> cd $PETALINUX
bash> petalinux-create -t project -s $XAPP_HOME/software/petalinux/bsp/
xapp1082_pl_eth_cso_sgmii.bsp
```

24.3.2.6.2.2 2.4.2.2 Configure Petalinux

```
bash> cd xapp1082_pl_eth_sgmii
bash> petalinux-config
```

Save the changes and exit. It will download the remote kernel and checkout xilinx-v2017.4 tag.
NOTE: Above step may take a longer time depending on the network bandwidth.

24.3.2.6.2.3 **2.4.2.3 Configure the kernel**

```
bash> petalinux-config -c kernel
```

Enable the Xilinx PHY driver and Disable the AXI DMA driver
Device Drivers> Network device support > PHY Device support and infrastructure >
<*> Drivers for xilinx PHYs
Device Drivers> DMA Engine Support> Xilinx DMA Engines >
<> Xilinx AXI DMA Engine
Save the changes and exit.

24.3.2.6.2.4 **2.4.2.4 Apply FSBL patch**

Refer to the AR 66006 for configuring the SFP and SI5324 using I2C in FSBL
Also user can copy the files present in fsbl_patch_files folder to configure the clock and SFP.

```
bash> cd components/bootloader/zynq_fsbl
bash> cp -f $XAPP_HOME/software/petalinux/fsbl_patch_files/* .
bash> cd -
```

24.3.2.6.2.5 **2.4.2.5 Modifications to system-user.dtsi**

For PL ETHERNET SGMII,

Modify the system-user.dtsi by navigating to the the file using 'vi project-spec/meta-user/recipes-bsp/device-tree/files/system-user.dtsi',

```
/include/ "system-conf.dtsi"
/ {
};

&axi_ethernet {
local-mac-address = [00 0a 35 00 1e 52];
};
```

Save and exit.

24.3.2.6.2.6 **2.4.2.6 Build**

Build all images using PetaLinux

```
bash> petalinux-build -v
```

24.3.2.6.2.7 **2.4.2.7 Create Zynq Boot image (BOOT.bin)**

```
bash> cd images/linux
bash> petalinux-package --boot --fsbl zynq_fsbl.elf --fpga pl_eth_sfp.bit (sgmii)
```

24.3.2.6.2.8 **2.4.2.7 SD Images**

SD Deployable binaries:-
a) BOOT.bin
b) image.ub
Copy BOOT.BIN and image.ub from $PETALINUX/ xapp1082_pl_eth_sgmii/images/linux to SD partition and run the setup.
Note: Instructions on PetaLinux BSP creation (rather PetaLinux flow from scratch) is provided in Appendix1 below.

24.3.2.7 2.5 Test Instructions

In the version 4.0, kernel builds the required drivers statically.So, drivers are already inserted as part of the kernel booting.
Refer the **section 3.7** to use the *ifconfig utility* for configuring the *interface **eth1** for PS EMIO and PL Ethernet.*

24.3.2.8 3. XAPP1082 v3.0

XAPP1082 v3.0 introduces:-

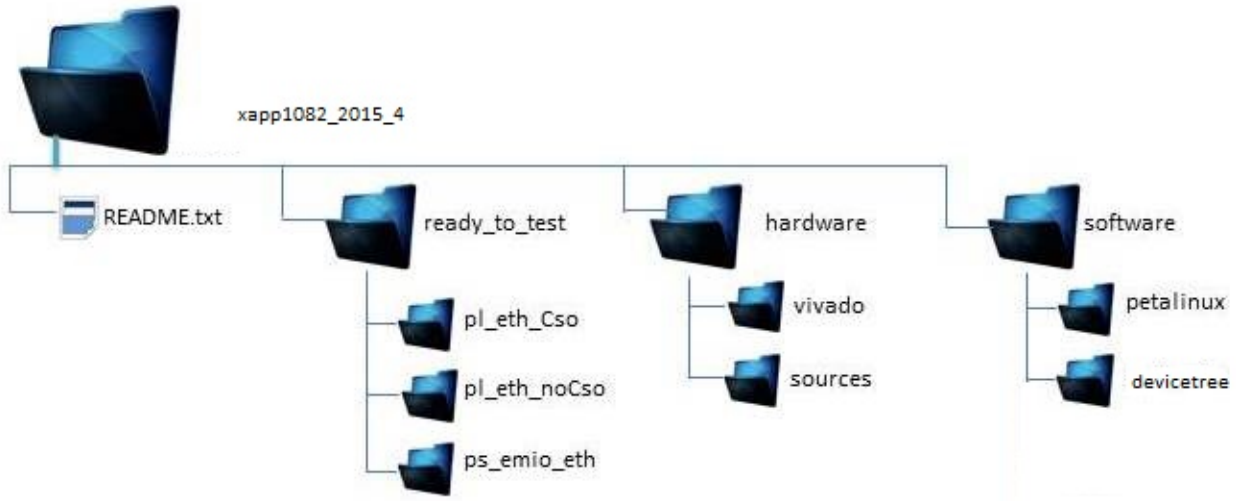
- Supports Vivado 2014.4 (IPI Design)
- PetaLinux 2014.4 SDK

24.3.2.9 3.1 Prerequisites

- PetaLinux 2014.4 SDK
- A Linux development PC with the distributed version control system Git installed. For more information, refer to Using Git and to UG821: Xilinx Zynq-7000 EPP Software Developers Guide.
- Other system utilities like make (3.82 or higher) and corkscrew if accessing git behind a firewall.

24.3.2.10 3.2 Directory structure

The XAPP1082 v3.0 is released with the source code, Xilinx Vivado and Petalinux projects and an SD card image that enables the user to run the demonstration.
It also includes the binaries necessary to configure and boot the Zynq-7000 SoC board.
This wiki page assumes the user has already downloaded the XAPP package and extracted its contents to the XAPP home
directory referred to as XAPP_HOME in this wiki.



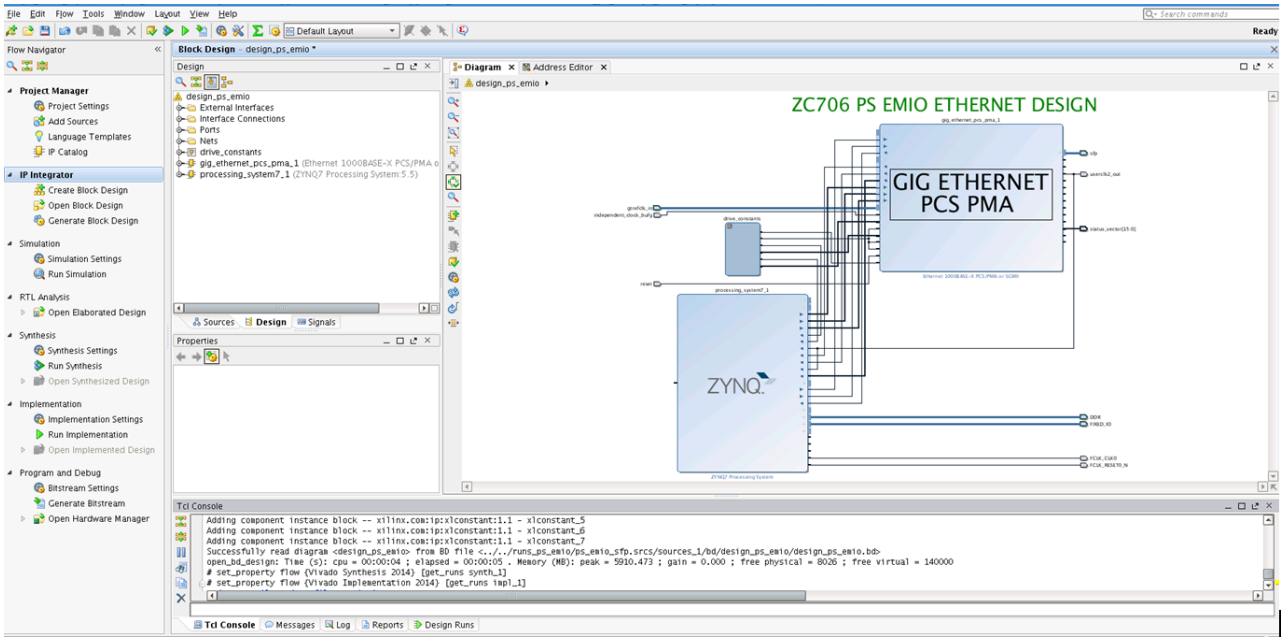
24.3.2.11 3.3 Vivado

Two designs are described in this application note. The designs support Vivado IP Integrator tool flow.
a) PS Ethernet (GEM1) that is connected to a 1000BASE-X physical interface in PL through an EMIO interface.
b) PL Ethernet implemented as soft logic in PL and connected to the 1000BASE-X physical interface in PL.

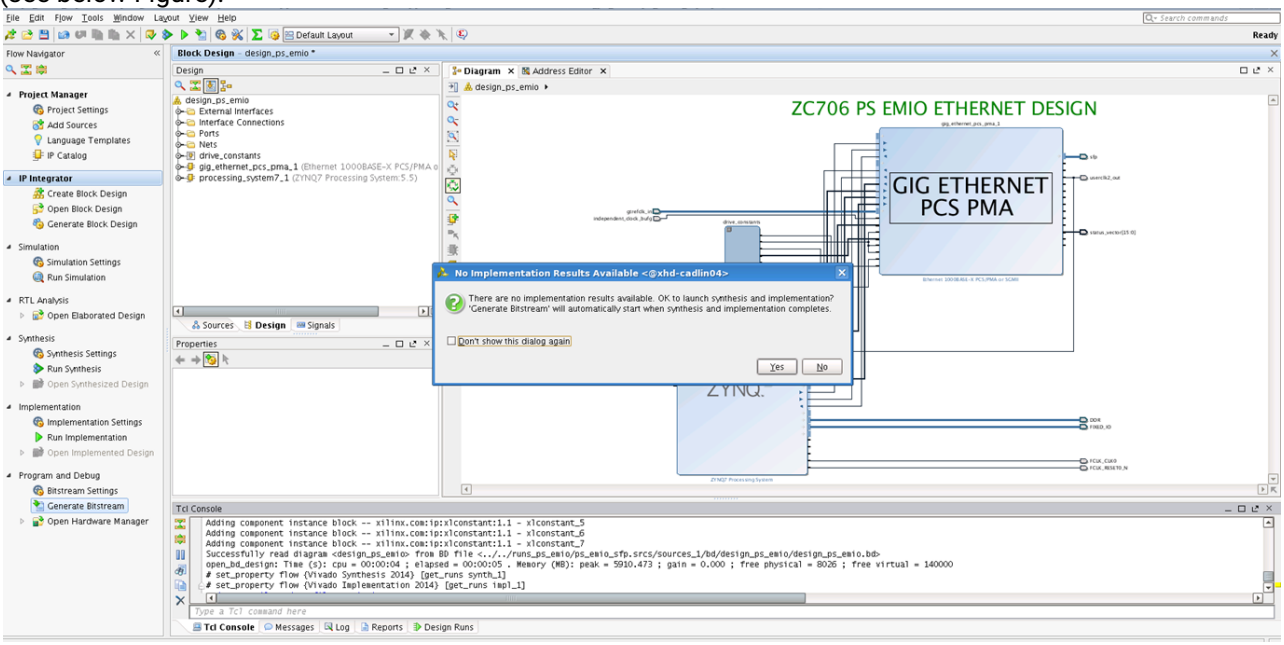
24.3.2.11.1 Building PS-EMIO design

To rebuild the hardware design, execute the following (after setting up Vivado environment).
1. Open a Linux terminal or Vivado tcl shell in windows
2. Navigate to hardware/vivado/scripts/ps_emio_eth for PS EMIO Ethernet design
$ vivado -source ps_emio_eth.tcl

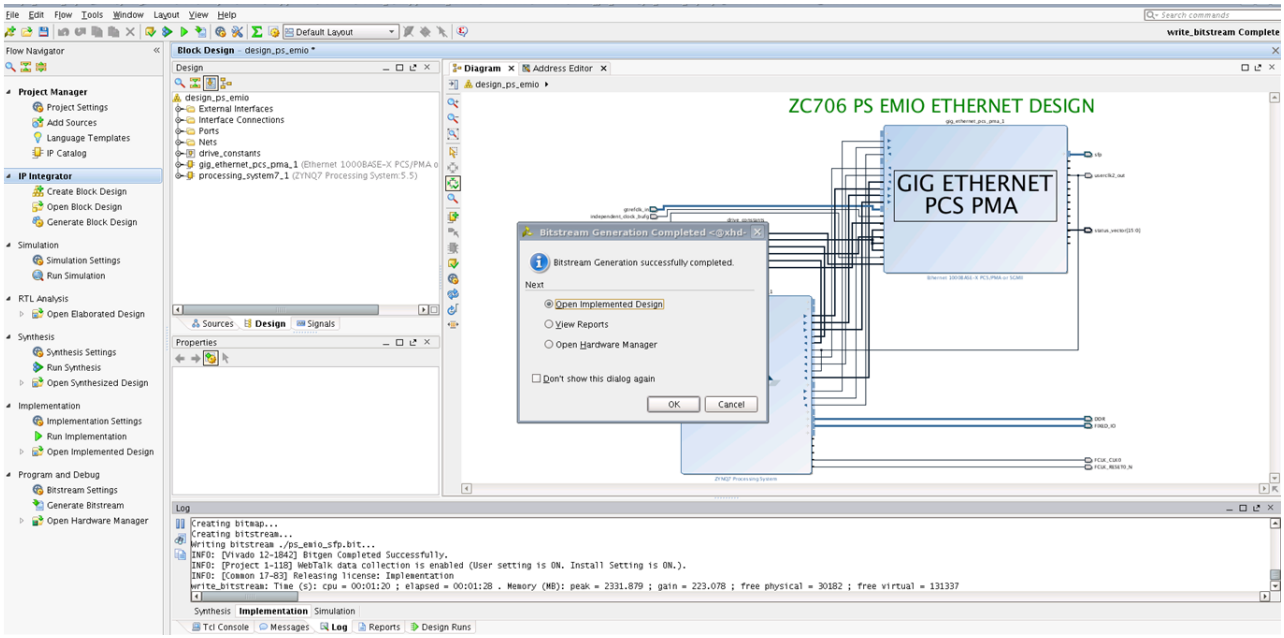
This step creates the project and opens the Vivado IDE with the design loaded (See below Figure). Relevant constraints file is also associated with the design.



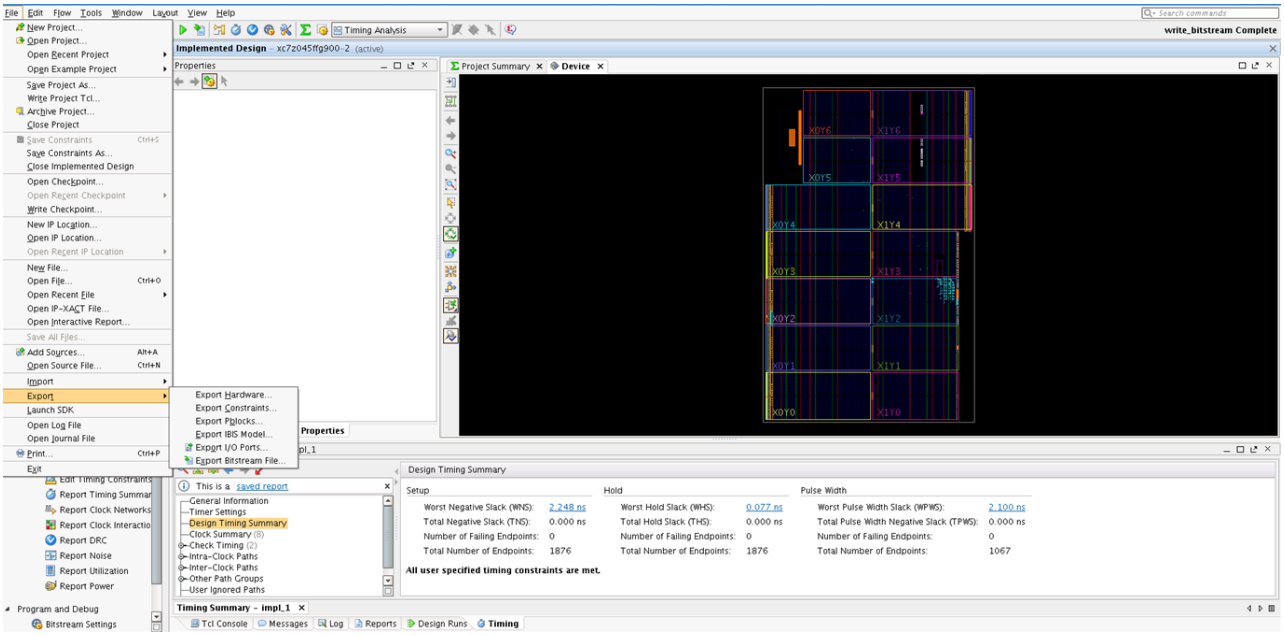
3. In the Flow Navigator Panel, click on 'Generate Bitstream' to implement the design and get a bitstream (see below Figure).



4. On completion of bitstream generation, open the implemented design (see below Figure).



5. Click on File --> Export --> Export Hardware to SDK (see below Figure)



6. Choose "Include bitstream" option, and click OK (see below Figure)

7. You can choose to launch SDK (File --> Launch SDK) so that hardware platform is already loaded into SDK.

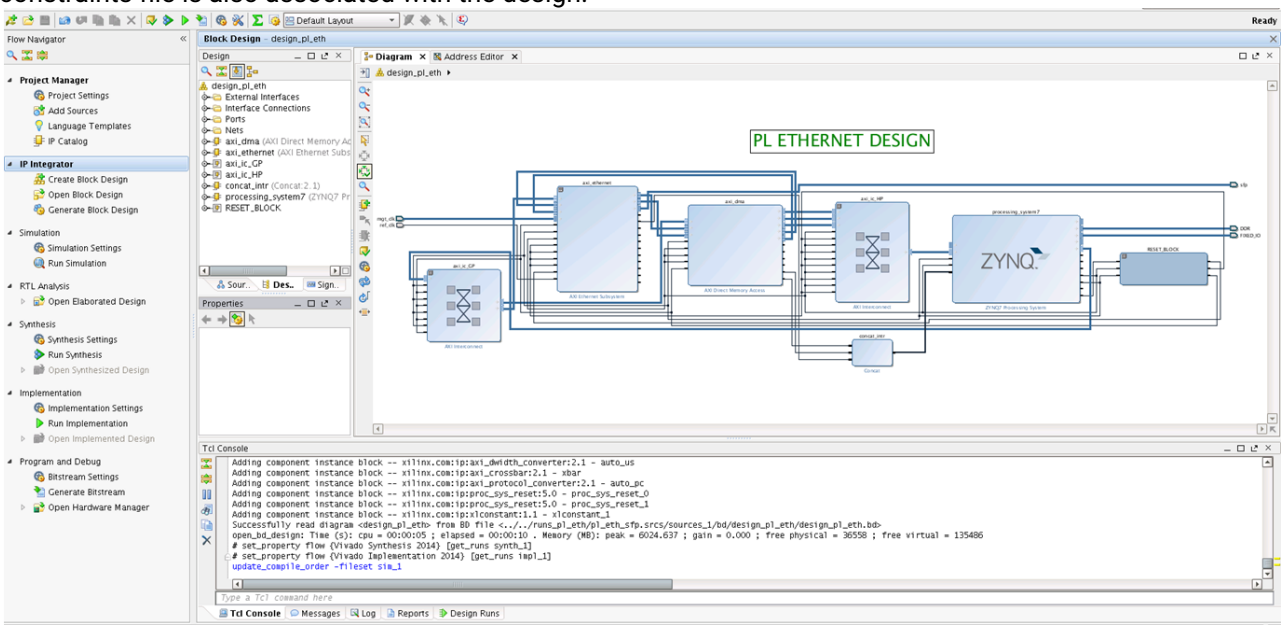
24.3.2.11.2 Building PL Ethernet design

To rebuild the hardware design, execute the following (after setting up Vivado environment).

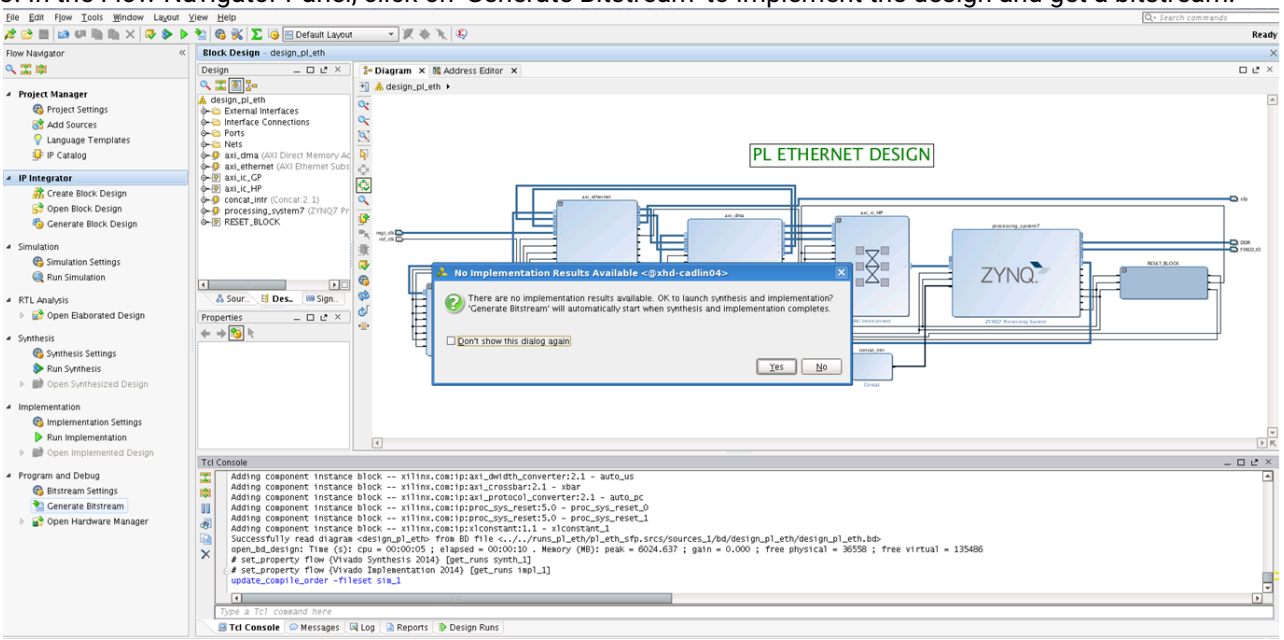
1. Open a Linux terminal or Vivado tcl shell in windows
2. Navigate to hardware/vivado/scripts/pl_eth for PL Ethernet design

$ vivado -source pl_eth.tcl

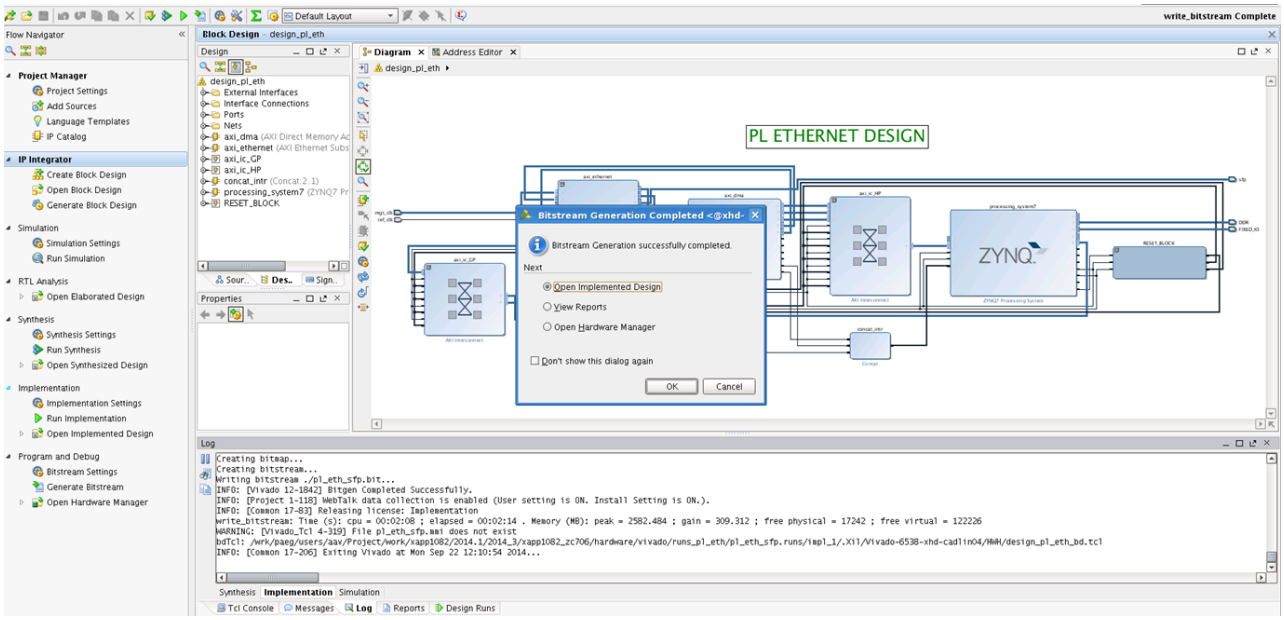
This step creates the project and opens the Vivado IDE with the design loaded (see below Figure). Relevant constraints file is also associated with the design.



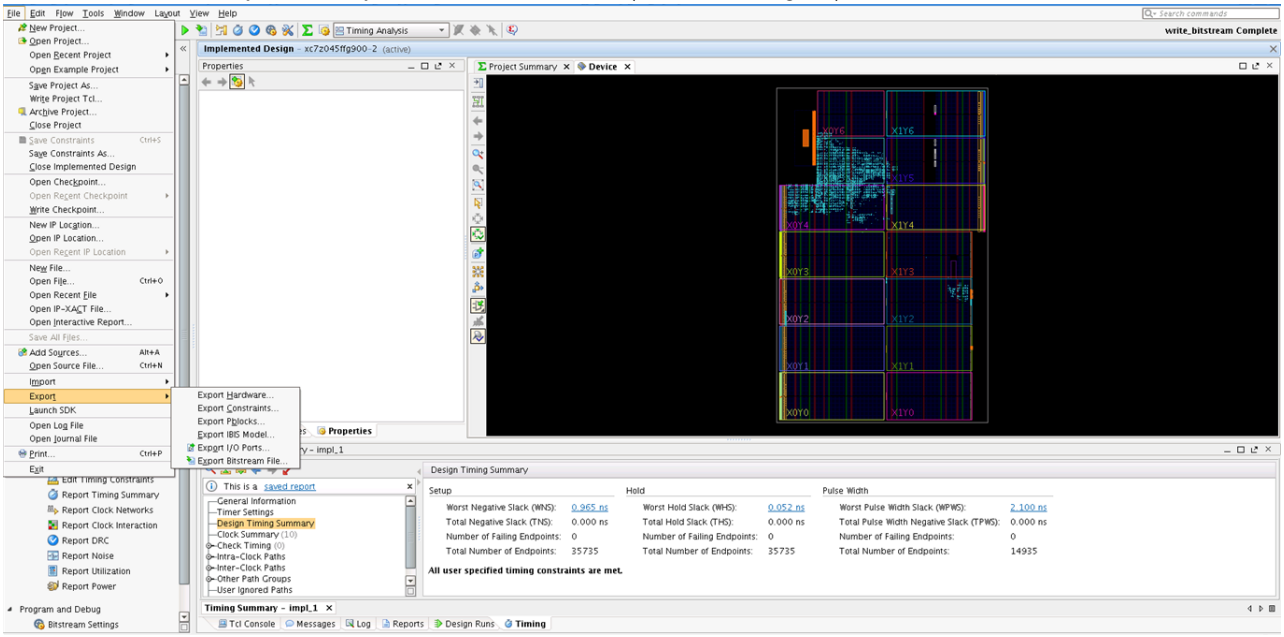
3. In the Flow Navigator Panel, click on 'Generate Bitstream' to implement the design and get a bitstream.



4. On completion of bitstream generation, open the implemented design (see below Figure).



5. Click on File --> Export --> Export Hardware to SDK (see below Figure)



6. Choose "Include bitstream" option, and click OK.

7. You can choose to launch SDK (File --> Launch SDK) so that hardware platform is already loaded into SDK.

24.4 XAPP1082 2017.4 Performance

This wiki page summarizes the performance of PS-EMIO (MACB diver) and PL Ethernet with CSO support for 1000BaseX and SGMII.

Build:

Vivado 2017.4
Kernel version 4.6 (2017.4)
These measurements are obtained against u-buntu high performance peer machine. The NIC on motherboard has been used with default offload options
(GSO, TSO) enabled. Performance is expected to differ when a different peer OS is used and peer NIC offload options are disabled.

Socket size in netperf (-s) is configured to 65536 for inbound tests.
PS EMIO Ethernet for 1000BaseX
Outbound: 801 Mbps
Inbound: 756 Mbps

PS EMIO Ethernet for SGMII
Outbound: 757 Mbps
Inbound : 758 Mbps

PL Ethernet for 1000BaseX with CSO offload
Outbound: 890 Mbps
Inbound: 745 Mbps

PL Ethernet for SGMII with CSO offload
Outbound: 824 Mbps
Inbound: 745 Mbps

NOTE : The above measurements are done without using taskset application which sets CPU affinity. So, there is ~5% throughput down compared previous version. The taskset application will be delivered in next release.

24.5 XAPP1082 2017.2 Performance

24.5.1 This wiki page summarizes the performance of PS-EMIO (MACB diver) and PL Ethernet with CSO support for 1000BaseX and SGMII.

Build:

Vivado 2017.2
Kernel version 4.6 (2017.2)
These measurements are obtained against u-buntu high performance peer machine. The NIC on motherboard has been used with default offload options
(GSO, TSO) enabled. Performance is expected to differ when a different peer OS is used and peer NIC offload options are disabled.

Socket size in netperf (-s) is configured to 65536 for inbound tests.
PS EMIO Ethernet for 1000BaseX
Outbound: 825 Mbps

Inbound: 751 Mbps

PS EMIO Ethernet for SGMII
Outbound: 849 Mbps
Inbound : 727 Mbps

PL Ethernet for 1000BaseX with CSO offload
Outbound: 883 Mbps
Inbound: 731 Mbps

PL Ethernet for SGMII with CSO offload
Outbound: 821 Mbps
Inbound: 753 Mbps

NOTE : The above measurements are done without using taskset application which sets CPU affinity. So, there is ~5% throughput down compared previous version. The taskset application will be delivered in next release.

24.6 Zynq Ethernet Performance 2014.4

24.6.1 XAPP1082 v3.0 2014.4

This wiki page summarizes the performance of PS-EMIO and PL Ethernet (with/without) CSO and jumbo frame support.
Build:
Vivado 2014.4
Kernel version 2014.4 (3.17)

These measurements are obtained against Fedora-20 high performance peer machine. The NIC on motherboard has been used with default offload options
(GSO, TSO) enabled. Performance is expected to differ when a different peer OS is used and peer NIC offload options are disabled.
Socket size in netperf (-s) is configured to 65536 for inbound tests.

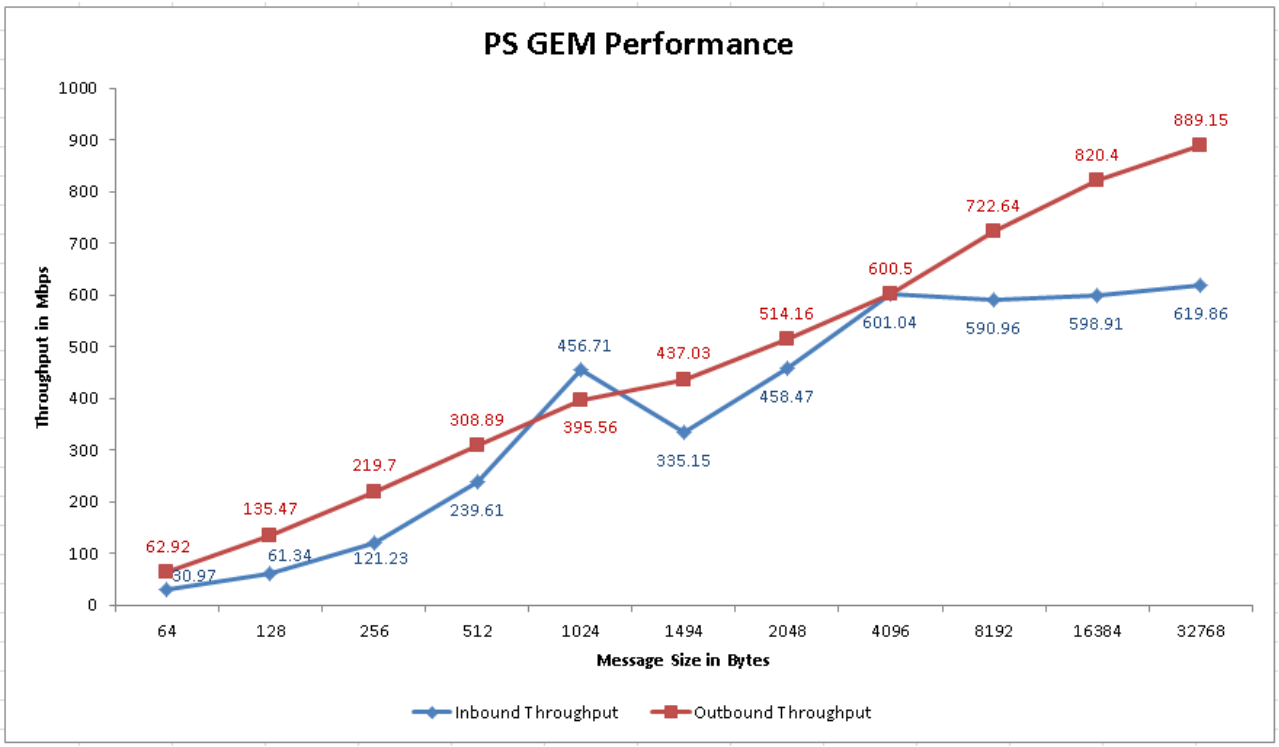


Figure-1 : PS GEM Performance

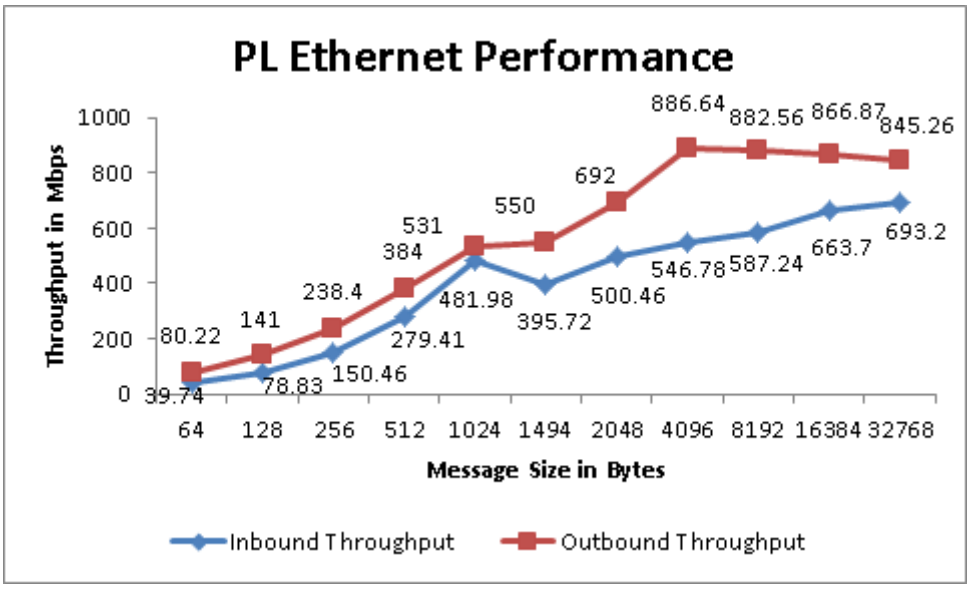


Figure-2 : PL Ethernet Performance

Note: Coalescing of packets may be observed even with '-D' option in netperf. To observe appropriate performance with smaller packets, one can also restrict MTU to smaller values. For MTU calculation include TCP/IP and MAC header in addition to message size. For example for 64B message size, set MTU to 130 (=14+20+32+64). The following commands can be used to set different MTU values-

```
ifconfig eth1 down
ifconfig eth1 mtu 130
ifconfig eth1 up
```

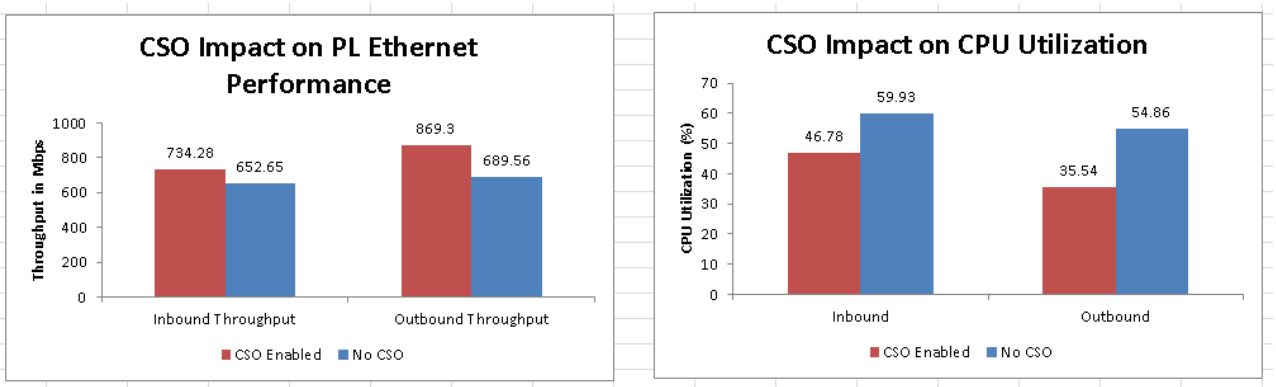


Figure-3 : PL Ethernet CSO impact on Throughput & CPU Utilization

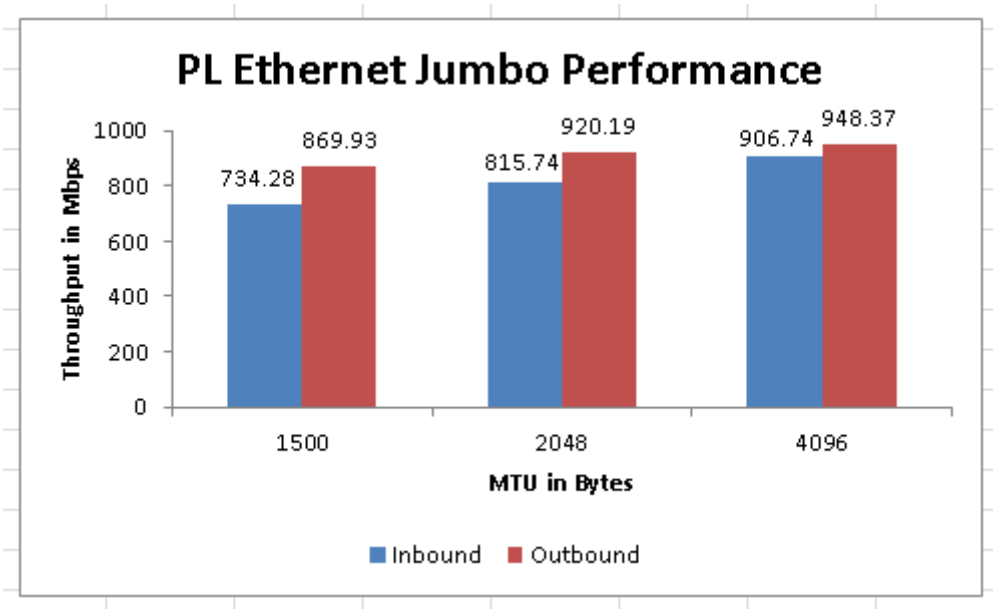


Figure-4 : PL Ethernet Jumbo Frame Performance

24.7 Zynq Ethernet Performance 2015.1

XAPP1082 v3.0 2015.1

This wiki page summarizes the performance of PS-EMIO and PL Ethernet (with/without) CSO and jumbo frame support.

Build:
Vivado 2015.1
Kernel version 3.18 (2015.1)

These measurements are obtained against Fedora-20 high performance peer machine. The NIC on motherboard has been used with default offload options
(GSO, TSO) enabled. Performance is expected to differ when a different peer OS is used and peer NIC offload options are disabled.
Socket size in netperf (-s) is configured to 65536 for inbound tests.

PS EMIO Ethernet
Outbound: 823.59 Mbps
Inbound: 631.52 Mbps

PL Ethernet with CSO offload
Outbound: 793.91 Mbps
Inbound: 704.74 Mbps

24.8 Zynq Ethernet Performance 2015.2

XAPP1082 v3.0 2015.2

In 2015.2, performance numbers are not updated on wiki because xilinx EMACPS driver is replaced with open source MACB driver and new PL AXI ethernet driver are provided in mainline for this release. So, we need to provide the PHY driver for 1000BASE-X as part of PHYLIB subsystem in the kernel, but PHY IP required changes which is part of 2015.3. 1000Base-X PHY driver will be provided in 2015.3 kernel release as part of PHYLIB subsystem. Performance numbers are provided in 2015.3.

24.9 Zynq Ethernet Performance 2015.3

XAPP1082 v3.0 2015.3

This wiki page summarizes the performance of PS-EMIO (MACB diver) and PL Ethernet with CSO support.
Build:
Vivado 2015.3
Kernel version 4.0 (2015.3)

These measurements are obtained against u-buntu high performance peer machine. The NIC on motherboard has been used with default offload options
(GSO, TSO) enabled. Performance is expected to differ when a different peer OS is used and peer NIC offload options are disabled.
Socket size in netperf (-s) is configured to 65536 for inbound tests.

PS EMIO Ethernet
Outbound: 850.06 Mbps

Inbound: 769.7 Mbps

PL Ethernet with CSO offload
Outbound: 640.56 Mbps
Inbound: 744.84 Mbps

24.10 Zynq Ethernet Performance 2015.4

XAPP1082 v4.0 2015.4
This wiki page summarizes the performance of PS-EMIO (MACB diver) and PL Ethernet with CSO support for 1000BaseX and SGMII.
Build:
Vivado 2015.4
Kernel version 4.0 (2015.4)

These measurements are obtained against u-buntu high performance peer machine. The NIC on motherboard has been used with default offload options
(GSO, TSO) enabled. Performance is expected to differ when a different peer OS is used and peer NIC offload options are disabled.
Socket size in netperf (-s) is configured to 65536 for inbound tests.

PS EMIO Ethernet for 1000BaseX
Outbound: 850.72 Mbps
Inbound: 769.7 Mbps

PS EMIO Ethernet for SGMII
Outbound: 842.96 Mbps
Inbound: 760.97 Mbps

PL Ethernet for 1000BaseX with CSO offload
Outbound: 641.56 Mbps
Inbound: 727.63 Mbps

PL Ethernet for SGMII with CSO offload
Outbound: 642.73 Mbps
Inbound: 722.70 Mbps

24.10.1 Note: For better performance numbers and stability improvement of PL Ethernet driver, please refer to xilinx-v2016.1 tag kernel driver.

PS Ethernet for 1000BaseX in Bi-directional mode using iperf commands
Refer to AR66670
Outbound: 269 Mbps
Inbound: 329 Mbps

PL Ethernet for 1000BaseX with CSO offload in Bi-directional mode using iperf commands
Refer to AR66446
Outbound: 316 Mbps
Inbound: 429 Mbps

24.11 Zynq Ethernet Performance 2016.1

XAPP1082 v4.0 2016.1

This wiki page summarizes the performance of PS-EMIO (MACB diver) and PL Ethernet with CSO support for 1000BaseX and SGMII.

Build:

Vivado 2016.1
Kernel version 4.0 (2016.1)
These measurements are obtained against u-buntu high performance peer machine. The NIC on motherboard has been used with default offload options
(GSO, TSO) enabled. Performance is expected to differ when a different peer OS is used and peer NIC offload options are disabled.

Socket size in netperf (-s) is configured to 65536 for inbound tests.
PS EMIO Ethernet for 1000BaseX
Outbound: 923.72 Mbps
Inbound: 778.7 Mbps

PS EMIO Ethernet for SGMII
Outbound: 915.96 Mbps
Inbound: 776.97 Mbps

PL Ethernet for 1000BaseX with CSO offload
Outbound: 924.87 Mbps
Inbound: 746.60 Mbps

PL Ethernet for SGMII with CSO offload
Outbound: 924.87 Mbps
Inbound: 746.60 Mbps

24.12 Zynq Ethernet Performance 2016.2

XAPP1082 v4.0 2016.2

This wiki page summarizes the performance of PS-EMIO (MACB diver) and PL Ethernet with CSO support for 1000BaseX and SGMII.

Build:

Vivado 2016.2
Kernel version 4.0 (2016.2)
These measurements are obtained against u-buntu high performance peer machine. The NIC on motherboard has been used with default offload options
(GSO, TSO) enabled. Performance is expected to differ when a different peer OS is used and peer NIC offload options are disabled.

Socket size in netperf (-s) is configured to 65536 for inbound tests.

and


utils are updated to 2016.2
Performance numbers are almost same as 2016.1 **release for both PS and PL Ethernet.**

24.13 Zynq Ethernet Performance 2016.3

24.13.1 2016.3

XAPP1082 v4.0 2016.3

This wiki page summarizes the performance of PS-EMIO (MACB diver) and PL Ethernet with CSO support for 1000BaseX and SGMII.

Build:

Vivado 2016.3
Kernel version 4.6 (2016.3)
These measurements are obtained against u-buntu high performance peer machine. The NIC on motherboard has been used with default offload options
(GSO, TSO) enabled. Performance is expected to differ when a different peer OS is used and peer NIC offload

options are disabled.

Socket size in netperf (-s) is configured to 65536 for inbound tests.
PS EMIO Ethernet for 1000BaseX
Outbound: 893.12 Mbps

Inbound: 741.56 Mbps

PS EMIO Ethernet for SGMII
Outbound: 841 Mbps
Inbound : 754 Mbps

PL Ethernet for 1000BaseX with CSO offload
Outbound: 905 Mbps
Inbound: 742 Mbps

PL Ethernet for SGMII with CSO offload
Outbound: 894 Mbps
Inbound: 748 Mbps

NOTE : The above measurements are done without using taskset application which sets CPU affinity. So, there is ~5% throughput down compared previous version. The taskset application will be delivered in next release.

24.14 Zynq-7000 AP SoC Performance – Gigabit Ethernet achieving the best performance

Table of Contents

24.14.1 Document History

Date	Version	Author	Description of Revisions
06/15/2015	0.1	Upender Cherukupally	Release 1.0

24.14.2 Overview

This techtip describes the challenges in achieving the best Ethernet performance and best design practices to achieve the better performance using the Zynq-7000 AP SoC. This techtip explains briefly on the various solutions available for the achieving the better performance using the Zynq-7000 AP SoC, steps to re-create, compile and run the design where ever it is possible. This paper also explains various ways to implement the TCP/IP protocols and discusses the advantages on each implementations like TCP/IP offload engine, software implementations of stack like lwIP and Linux Ethernet sub-system.

Zynq-7000 AP SoC has an in-built dual Giga bit Ethernet controllers which can support 10/100/1000 Mb/s EMAC configurations compatible with the IEEE 802.3-2008 standard. The Programming Logic (PL) sub system of the Zynq-7000 AP SoC can also be configured with additional soft AXI EMAC controllers if the end application requires more than two Giga bit Ethernet Controller. Following is the example block diagram of the Zynq-7000 AP SoC with GEMACs using the ZC706 Development board

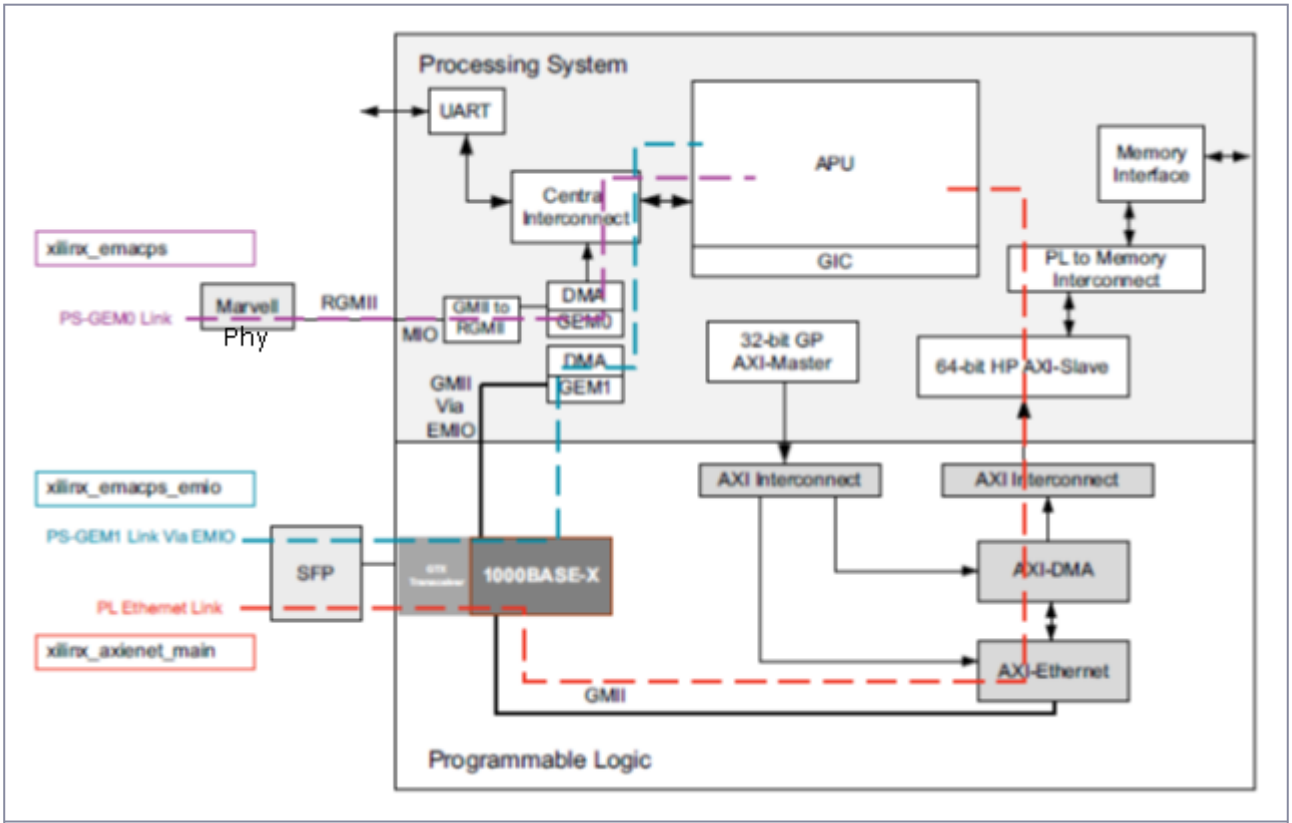


Figure 1: Gigabit Ethernet Design block diagram using Zynq-7000 AP SoC

Above example scenario shows all the possible gigabit Ethernet MAC configurations using the ZC706 board.

1. PS-GEM0 is connected to the Marvell PHY through the reduced gigabit media independent interface (RGMII), which is the default setup for the ZC706 board.
2. PS Ethernet (GEM1) that is connected to a 1000BASE-X physical interface in PL through an EMIO interface and
3. PL Ethernet implemented as soft logic in PL and connected to the 1000BASE-X physical interface in PL.

The PS-GEM1 and the PL Ethernet share the same 1000 BASE-X PHY so only one will be available at a given point of time on this board among these two configurations. PS GEM0, PS GEM1 supports max frame size 1522 bytes. There can be frame size up to 16k which called jumbo frame. The AXI EMAC in PL can be configured to have the Jumbo frame support which actually reduces the number of transfer for a large data size which is of greater/multiples of the jumbo frame size. The Jumbo frame support in AXI EMAC is the major difference between PS and PL EMACs.

This techtip explains the following sections:

1. Gigabit Ethernet solutions using the Zynq-7000 AP SoC and application data path,
2. What is Ethernet performance,

3. Types of TCP/IP stack implementations,
4. Solutions readily available using the Zynq-7000 AP SoC,
5. Techniques which can be applied and achieve the maximum possible Ethernet data performance

24.14.3 Implementation

Implementation Details	
Design Type	PS only
SW Type	Zynq-7000 AP SoC Linux & Zynq-7000 AP SoC Baremetal
CPUs	2 ARM Cortex-A9: SMP Linux and Baremetal configurations
PS Features	• DDR3 • Cache • L1 and L2 Cache • OCM • Generic Interrupt Controller • USB 2.0 OTG Controller
Boards/Tools	ZC702 Kit & ZC706 Kit
Xilinx Tools Version	Vivado & SDK 2015.1 or latest
Other Details	-

Files Provided	
ZC702_ZC706_ReadyToUseImages	Contain folders: Source, SD Card Images required to follow the procedure below

24.14.4 Ethernet Performance

With the Ethernet frame of size 1538 bytes, there can be a maximum of 81275 frames per second on 1000 Mbps (or 1Gbps), half-duplex (one way) Ethernet pipe. In other words, this would mean, that the Ethernet network interface would be subjected to a frame transaction in every 12.3 us. So, in order to sustain the line rate of 1 Gbps, the software on the host CPU must finish its work of handling & processing of a frame with 12.3 us and then be available to handle the next arriving frame. Such a time bound execution will ensure that software is working in tandem with the Ethernet hardware. For any reason, if the software is not able to handle and process a frame or packet within the stipulated time (of 12.3 us), then, it will not be able to strike the much needed equilibrium with the Ethernet hardware to sustain the line rate of 1 Gbps. The lack of tandem between software and hardware will create either a backpressure (if doing RX) or a starvation (if doing TX) on the hardware. A sustained backpressure will eventually force the Ethernet hardware to overrun and tail-drop the frames from the Ethernet wire, whereas starvation will instantly cause the hardware to under-run and the wire would be underutilized.
The following sections explains:

1. Features of the Zynq-7000 AP SoC for implementing the Ethernet based solution to achieve the best performance
2. End to end data flow of the Ethernet data and best design practices in hardware as well as software design flows
3. List of all the available and ready to use reference designs for Ethernet based solutions using the Zynq-7000 AP SoC

24.14.5 Ethernet Data movement in Zynq-7000 AP SoC

The Gigabit Ethernet MAC Controller on Zynq Processing System comprises of three blocks.

1. MAC Controller
2. FIFO(Packet Buffer)
3. Ethernet DMA Controller

The Ethernet DMA controller is attached to the FIFO to provide a scatter-gather capability for packet data storage in a Zynq processing system. The Ethernet DMA uses separate transmit and receive lists of buffer descriptors, with each descriptor describing a buffer area in memory
Receive Path:
The data received by the controller is written to pre-allocated buffer descriptors in system memory. These buffer descriptor entries are listed in the receive buffer queue. The Receive-buffer Queue Pointer register of the Ethernet DMA points to this data structure on initialization and uses it to continuously and sequentially to copy the Ethernet packet received in the Ethernet FIFO to Memory address specified in the receive buffer queue
Rx Ring buffers and Tx Ring buffers location can be in DDR or OCM and access latencies of these memories, the speed at which the instructions executes for packet processing will also improves the overall performance

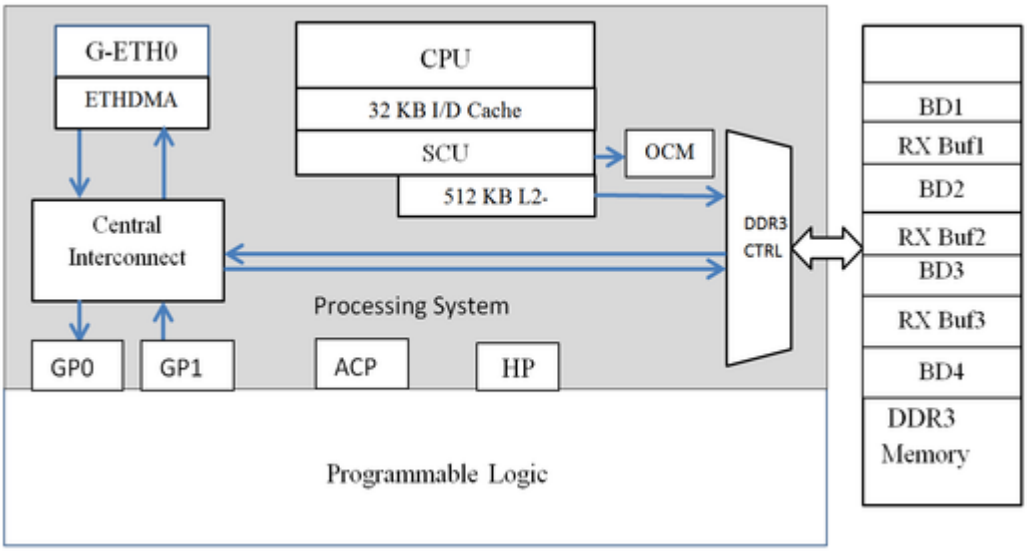


Figure 2: Ethernet Data movement in Zynq-7000 AP SoC

When an Ethernet Packet is received by the MAC, the Ethernet DMA uses the address in the RX Buffer descriptor to push the packet that has been buffered in the Packet Buffer on Ethernet interface to DDR3 memory, via the central interconnects.
Data Receive Path: ETH0 -> ETH0 DMA (32-bit) -> Central Interconnect -> DDR3 Memory Controller (64-bit AXI).

Transmit Path
In case of transmit the Ethernet DMA uses the address in the TX Buffer descriptor to pull data from DDR3 Memory, through the central interconnect and finally to the ETH0 Interface.
Data Transmit path: DDR3 Memory Controller (64-bit AXI) -> Central Interconnect -> ETH0 DMA (32-bit) -> ETH0

The Ethernet application software access the application data through the cache, so from the DDR memory to the cache access latency will also be added to the overall the access time from the source of the data to the destination at the application software. To improve further the application data buffers can be stored in OCM.
The inbuilt features like built in DMA engines, MAC address filtering and intermediate buffers in both transmit and receive path helps in achieving the better performance using the PS EMAC controllers. The PS EMAC controllers allow multiple packets to be buffered in both transmit and receive directions. This allows the DMA to withstand far greater access latencies on the AXI and make more efficient use of the AXI bandwidth to:

1. Discard packets with error on the receive path before they are partially written out of the DMA thus saving AXI bus bandwidth and driver processing overhead
2. Retry collided transmit frames from the buffer, thus saving AXI bus bandwidth,
3. Implement transmit IP/TCP/UDP checksum generation offload

Ethernet Protocols Implementations

There are following three general approaches for implementing the using the TCP/IP protocol software:

1. If a generic/ fully featured operating system like Linux is used, then the series of instructions and execution flow may take bit longer time as part of the packet handling, processing, context switching, user space / kernel space transactions, data copy (load store) and interrupt thrashing. And there would be definitely still more work that CPU would be required to do before it can come back to Ethernet hardware to pick up or provide the next frame such as user space to kernel space transitions and mode switches etc. This wiki page explains few techniques to improve the performance in the implementations line Linux stacks. This techtip also gives the pointers to the example reference designs and documentation which uses the Linux for TCP/IP solutions.
2. On the other hand, the tailored, customized and relatively lightweight operating system like freeRTOS and lightweight (lwIP) TCP/IP implementations is used then the sequence of execution flow may showcase improvised numbers for handling & processing of the network frames or packets.
3. A micro kernel or similar tight loop software, with no Internet Protocol stack & having the sole functionality of handling the network packets and incorporating a very minimal custom logic for packet processing might be able to sustain the good line rate.

In all above cases, such software or a part of it, which is associated with the handling and processing of the network packets would be eventually locked up in the sufficiently sized instruction cache of the CPU. So, in the absence of any cache thrashing, any interrupt thrashing, any context switching and any data copy, the CPU will work optimally, perhaps to hold onto the line rate of 1000 Mbps

24.14.5.1 Linux Networking SW TCP/IP stack implementation

The TCP/IP or UDP/IP protocol implementation also plays a major role in overall Ethernet performance. Following is the Linux SW stack implementation

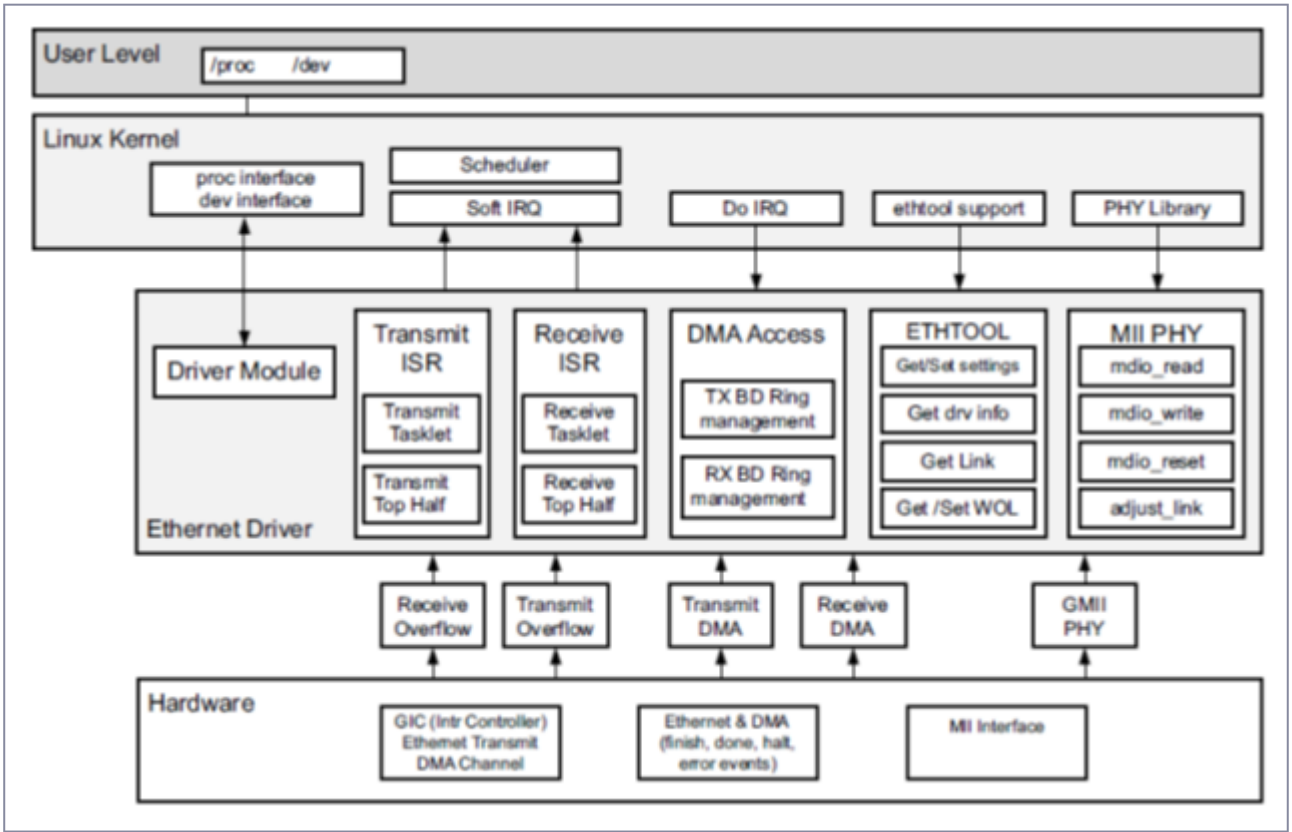


Figure 3: Linux TCP/IP SW stack implementation

Though the Linux kernel is based on monolithic architecture and works on sys call interface which involves the mode switches between user and kernel, it tries to optimize the system where ever it is possible to do so. Following are the few techniques used to achieve better performance
Memory allocation is a key factor in the performance of any TCP/IP stack. Most other TCP/IP implementations have a memory implementation mechanism that is independent from the OS. However, the Linux implementation took a different approach by using the slab cache method, which is used for other internal kernel allocation and this method has been adapted for socket buffers. With slab allocation, memory chunks suitable to fit data objects of certain type or size are pre-allocated.

The slab allocator keeps track of these chunks, known as caches, so that when a request to allocate memory for a data object of a certain type is received, it can instantly satisfy the request with an already allocated slot. Destruction of the object does not free up the memory, but only opens a slot which is put in the list of free slots by the slab allocator. The next call to allocate memory of the same size will return the now unused memory slot. This process eliminates the need to search for suitable memory space and greatly alleviates memory fragmentation.
Linux has the capability of deferring interrupt-level work to kernel threads to decrease latency problems. The goal of the OS should be to minimize context switches while a packet is processed by the stack. Linux uses softirqs to handle most of the internal processing

24.14.5.1.1 Minimal data copying

To achieve better performance the implementation should minimize the amount of copying to move a packet of data from the application, down through the stack to the transmission media. Linux provides scatter gather DMA support where the socket buffers are set up to allow for the direct transmission of lists of TCP segments. At the user level, when data is transferred through a socket, copying can be avoided and data can be mapped directly into the user space from the kernel space.

24.14.5.1.2 Endianness

TCP/IP or UDP/IP packet format follows the big endian notation and if the CPU core is in little endian then the endian conversion should happen at the software layers in both TX and Rx paths to interpret the data.

24.14.5.1.3 Design example on how to use PL AXI Ethernet implementations/ jumbo frames support in Zynq-7000 AP SoC

Jumbo frames are used in high data intensive applications. The packet size is 16384 bytes. The larger frame size improves the performance by reducing the number of the fragments for a given data size.
The XAPP-1082 provides details on how to use the jumbo frames support available in AXI EMAC for improved performance.
Following is the block diagram of the design provided with XAPP1082 using the ZC706 development kit:

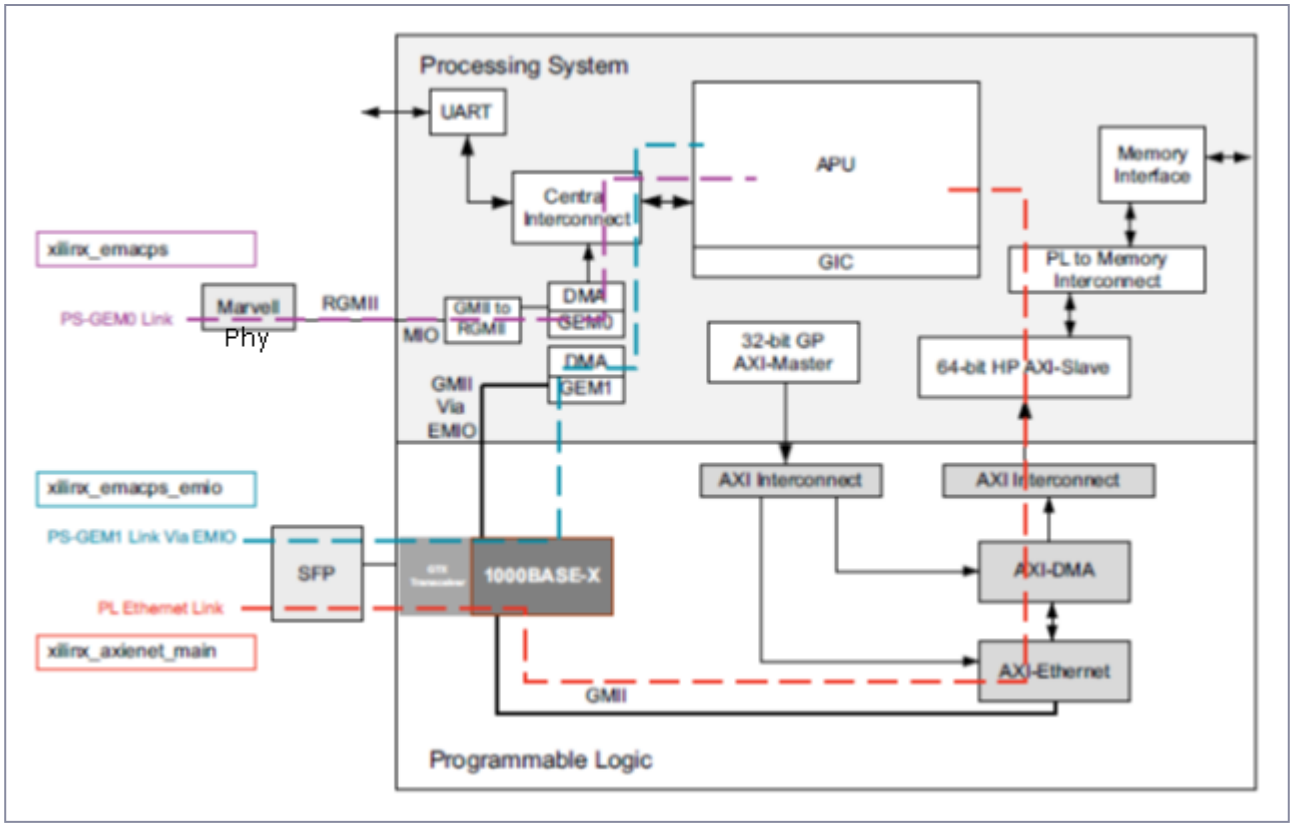


Figure 4: Block diagram of the design implemented as part of XAPP1082

The PS GEM1 and PL AXI Ethernet shares the 1000Base-X PHY so only either PSGEM1 or PLAXI Ethernet can be used at given point of time.
The complete design details and design files can be obtained from XAPP1082
http://www.xilinx.com/support/documentation/application_notes/xapp1082-zynq-eth.pdf

Steps to create the PS EMIO Ethernet solution, PL Ethernet solution and setting up the Embedded Linux for Zynq, refer the following link:
http://www.wiki.xilinx.com/Zynq+PL+Ethernet

24.14.5.1.4 Following are some of the techniques which can be applied to get the better performance from user space

These commands can be applied once the Linux kernel/XAPP1082 image is booted on Zynq-7000 AP SoC

- 1.Tuning the task priorities using the 'nice' system call form the user space
- Run the command ps –all to get the list of tasks and their PID, identify the network tasks and change the priorities using the nice sys call. The example format is shown below
- root@linux#nice –n -3 program; renice level PID
- 2.CPU affinity for the interrupt handlers/task: This will make sure the minimal cache operations as complete application/task is attached to a single core.
- root@linux#echo 01 > /proc/irq/19/smp_affinity
- 3.To share the load between the two Cortex A9s Taskset2 utility can be used while launching the Ethernet based applications
- 4.Window Size is also a configurable options for better performance. In a connection between a client and a server, the client tells the server the number of bytes it is willing to receive at one time from the server; this is the client's receive window, which becomes the server's send window. Likewise, the server tells the client how many bytes of data it is willing to take from the client at one time; this is the server's receive window and the client's send window. There are chances that window size will drop down to zero dynamically if the receiver is not able to process the data as fast as sender is sending the data. Larger the size then more chances to get the better performance. In Linux environment the Window size settings can be tuned by following the steps explained in the following links:
- http://www.cyberciti.biz/faq/linux-tcp-tuning/
- (linux-kernel/Documentation/networking/ip-sysctl.txt) http://www.cyberciti.biz/files/linux-kernel/Documentation/networking/ip-sysctl.txt
- While using the iperf bench marking the -w option can be used to specify the window size.

24.14.5.2 Bare metal lwIP TCP/IP stack

lwIP TCP/IP stack can be used in RAW API/standalone mode and Netconn API/Socket API using the RTOS features.

1. RAW Mode: This mode involves direct usage of EMAC driver calls, there is no middle level OS in communication with HW

2. Socket Mode: This mode uses the RTOS feature like message queues, threads to achieve the parallelism in software for both RX and TX

24.14.5.2.1 Design example on how to implement a bare metal TCP/IP stacks with and without the RTOS

The light weight TCP/IP stack implementations like lwIP and uIP can also significantly improves the Ethernet performance. The lwIP port for Zynq-7000 AP SoC with RAW API and Socket API modes is available with the XAPP-1026. This appnote provides all the best possible configurations of the lwIP TCP/IP stack to achieve the better performance with various Ethernet based applications like TFTP, Webserver, telnet in bare metal implementation.
The complete design details and design files can be obtained from XAPP1026 XAPP1026 (http://www.xilinx.com/support/documentation/application_notes/xapp1026.pdf)

As explained in the above appnote the lwIP TCP/IP stack is available for the designers as library as part of the SDK. To achieve the better performance designers can choose the following options of the lwIP library in SDK settings. lwIP TCP/IP performance settings for better performance:

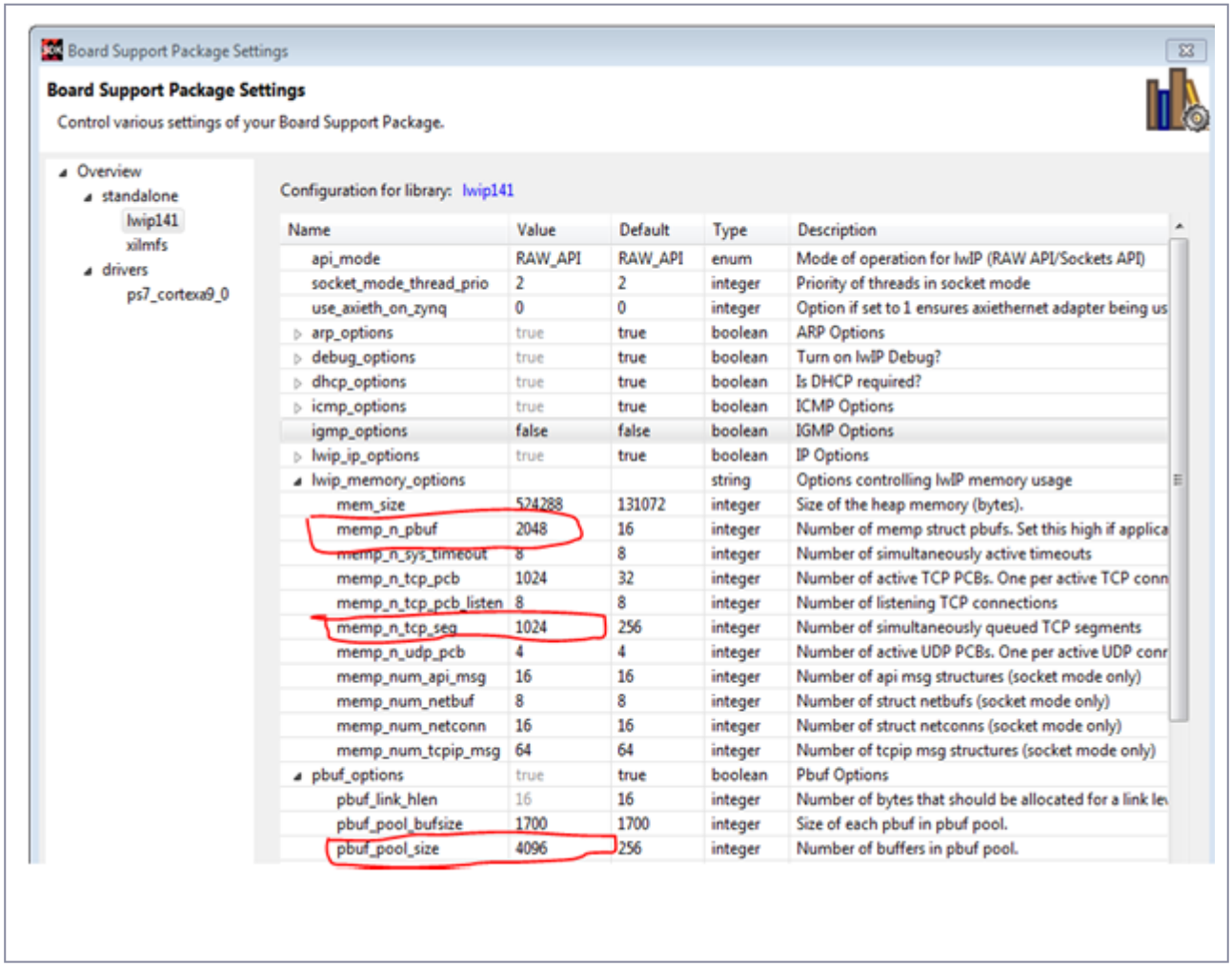


Figure 5: lwIP TCP/IP performance settings

If xilkernel RTOS is used in the design then following options best suits to achieve the better performance

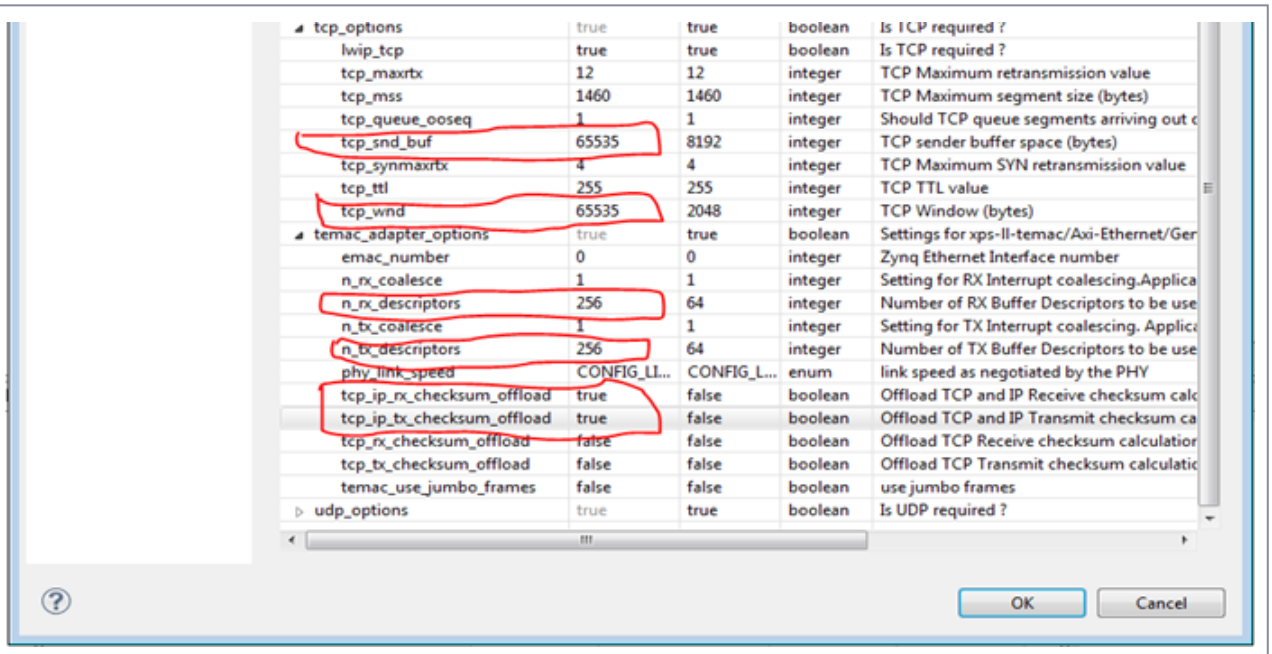


Figure 6 : lwIP stack settings for socket mode/when used along with RTOS

Conclusion

This techtip explained the Gigabit Ethernet solutions using the Zynq-7000 AP SoC, application data path, Ethernet performance, types of TCP/IP stack implementations, solutions readily available using the Zynq-7000 AP SoC, techniques which can be applied and achieve the maximum possible Ethernet data performance.

25 Zynq 7000 Tips and Tricks

This page contains a collection of tips and tricks specific to Zynq 7000, oriented around bare metal (non-Linux) designs in a Microcontroller environment.

25.1 Table of Contents

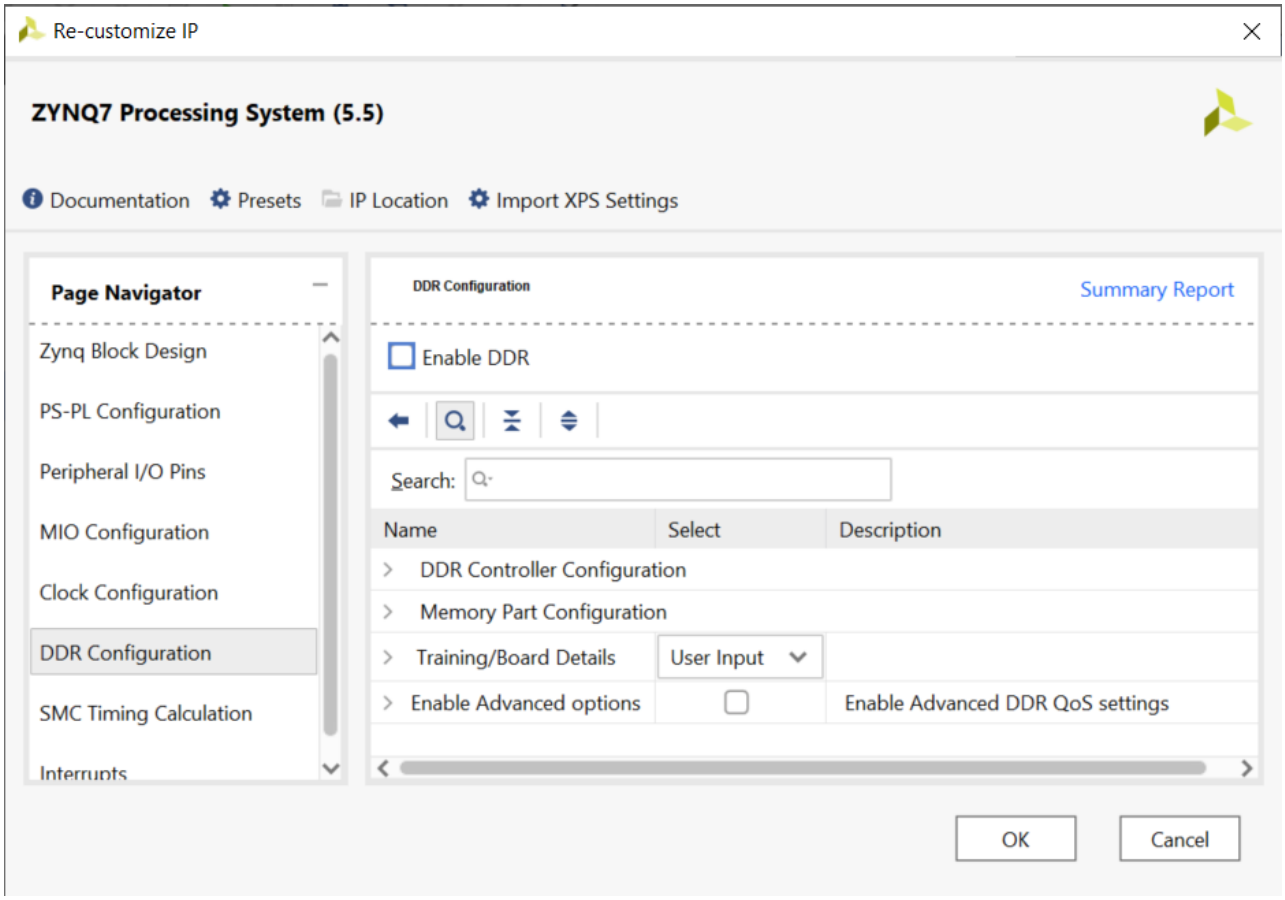
25.2 Introduction

Zynq 7000 has been in production for many years and still continues to be used in many lower cost designs. This page contains a number of small recipes that might be helpful in these designs. This page should be viewed more as a lab notebook with helpful hints assuming the user has a good knowledge of Zynq 7000 and the associated tools. It is not a tutorial for any of the required tools and the solution is not provided in any prebuilt images, so some work is required for success. A source code repository is provided at https://github.com/Xilinx-Wiki-Projects/software-prototypes with some of the primary changes required to the FSBL and SSBL based on the 2022.1 tools. The default FSBL source code is located at 2022.1 FSBL. The source code changes that follow might be more tactical, assuming a more refined/maintainable solution can be based on them and this is left as an exercise for the user.

25.3 DDR-less System

Most Zynq 7000 systems use DDR for system memory but some users want to use Zynq 7000 as a MicroBlaze replacement or as a typical Microcontroller with only internal memory and a boot media. The following DDR-less prototype system was created and tested for a specific system use case with QSPI flash as the boot media, with both unsigned and RSA signed images. Other system modes, such as other boot media types, encrypted images, etc. might not be supported by the code snippets illustrated in the following subparagraphs.

There are documented methods for DDR-less systems, such as Zynq-7000 AP SoC Boot - Booting and Running Without External Memory Tech Tip, which are more complex and that complexity might not be required for many systems. There is also a documented method, Answer Record 56044 , which this solution is based on. A DDR-less system is not supported out of the box but only requires small changes to be supported. Some features of the system are impacted by a DDR-less system including security. The following screen shot illustrates a Vivado IP Integrator system with the “Enable DDR” checkbox unchecked for no DDR.



A number of smaller FSBL changes are required to support various features without DDR. A key to making the following FSBL changes is to turn on debug in the build as all of the image details are output during boot image processing. See the Zynq 7000 FSBL wiki page at Zynq 7000 FSBL for build details. The following boot log snippet illustrates details when debug is turned on and allows boot image details to be correlated to FSBL processing with respect to image addresses.

```
Partition Number: 2
Header Dump
Image Word Len: 0x00001FC2
Data Word Len: 0x00001FC2
Partition Word Len:0x00002180
Load Addr: 0xFFFF0100
Exec Addr: 0xFFFF0100
Partition Start: 0x000C0000
Partition Attr: 0x00008010
Partition Checksum Offset: 0x00000000
Section Count: 0x00000001
Checksum: 0xFFE8FABA
Application
RSA Signed
PCAP:StatusReg = 0x40000F30
PCAP:device ready
```

```
PCAP:Clear done
PCAP register dump:
PCAP CTRL 0xF8007000: 0x4C00E07F
PCAP LOCK 0xF8007004: 0x0000001A
PCAP CONFIG 0xF8007008: 0x00000508
PCAP ISR 0xF800700C: 0x00033004
PCAP IMR 0xF8007010: 0xFFFFFFFF
PCAP STATUS 0xF8007014: 0x50000F30
PCAP DMA SRC ADDR 0xF8007018: 0xFC300001
PCAP DMA DEST ADDR 0xF800701C: 0xFFFF0101
PCAP DMA SRC LEN 0xF8007020: 0x00002180
PCAP DMA DEST LEN 0xF8007024: 0x00002180
PCAP ROM SHADOW CTRL 0xF8007028: 0xFFFFFFFF
PCAP MBOOT 0xF800702C: 0x8000C000
PCAP SW ID 0xF8007030: 0x00000000
PCAP UNLOCK 0xF8007034: 0x757BDF0D
PCAP MCTRL 0xF8007080: 0x30800110

DMA Done !
Authentication Done
```

By default, FSBL uses conditional compilation with XPAR_PS7_DDR* and fails to boot without DDR. The following code snippet illustrates a minor change to FSBL in the main() function in *main.c* where the conditional compilation with XPAR_PS7_DDR* is removed for a large block of code and a new conditional compilation block is added around the DDR initialization only.

```
#if 1 // was #ifdef XPAR_PS7_DDR_0_S_AXI_BASEADDR

    /* No DDR is a valid operating mode with a few other changes.
     */
#ifdef XPAR_PS7_DDR_0_S_AXI_BASEADDR
    /*
     * DDR Read/write test
     */
    Status = DDRInitCheck();
    if (Status == XST_FAILURE) {
        fsbl_printf(DEBUG_GENERAL,"DDR_INIT_FAIL \r\n");
        /* Error Handling here */
        OutputStatus(DDR_INIT_FAIL);
        /*
         * Calling FsblHookFallback instead of Fallback
         * since, devcfg driver is not yet initialized
         */
        FsblHookFallback();
    }
#endif
```

The LoadBootImage() function in the *image_mover.c* source file of the FSBL is altered to remove DDR address validation for image loading by adding conditional compilation with XPAR_PS7_DDR_0_S_AXI_BASEADDR. A hardware system built without DDR in Vivado removes the definition of this symbol in *xparameters.h*. The following code snippet illustrates the addition of conditional compilation to remove the DDR address validation such that images using internal memory can be loaded.

```
        /*
         * Load address check
         * Loop will break when PS load address zero and partition is
         * un-signed or un-encrypted
         */
#ifdef XPAR_PS7_DDR_0_S_AXI_BASEADDR
        if ((PSPartitionFlag == 1) && (PartitionLoadAddr < DDR_START_ADDR)) {
            if ((PartitionLoadAddr == 0) &&
                    (!((SignedPartitionFlag == 1) ||
                            (EncryptedPartitionFlag == 1)))) {
                break;
            } else {
                fsbl_printf(DEBUG_GENERAL,
                        "INVALID_LOAD_ADDRESS_FAIL\r\n");
                OutputStatus(INVALID_LOAD_ADDRESS_FAIL);
                FsblFallback();
            }
        }

        if (PSPartitionFlag && (PartitionLoadAddr > DDR_END_ADDR)) {
            fsbl_printf(DEBUG_GENERAL,
                    "INVALID_LOAD_ADDRESS_FAIL\r\n");
            OutputStatus(INVALID_LOAD_ADDRESS_FAIL);
            FsblFallback();
        }
#endif
```

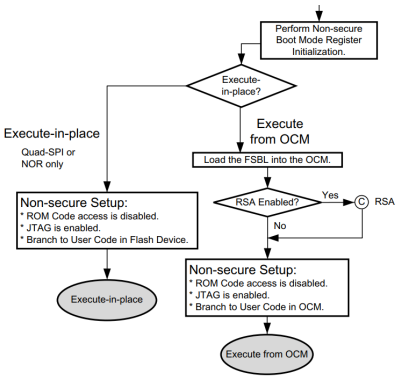
Partitions are normally moved from the boot media to DDR. A DDR-less use case should only move images located in internal memory (OCM or BRAM). The LoadBootImage() function in the *image_mover.c* source file of the FSBL is altered in the following code snippet which illustrates an example of adding conditions to prevent some images from being moved by not calling the PartitionMove() function.

```
        /*
         * Without DDR the authentication is done in place in linear QSPI, only
         * move partitions from a linear boot device for the non-XIP program
         * partitions, XIP partitions for flash and the associated RAM have an
         * exec address of flash or zero as bootgen does not know we are doing
         * XIP since we are doing RSA also
         */
         if ((PLPartitionFlag && !SignedPartitionFlag) || (!PLPartitionFlag &&
                ((PartitionExecAddr != 0 && ((PartitionExecAddr <
QSPI_FLASH_BASEADDR) ||
                        (PartitionExecAddr > QSPI_FLASH_HIGHADDR))) ||
                ((PartitionLoadAddr >= 0xFFFF0000) && (PartitionExecAddr == 0)) ||
                ((PartitionLoadAddr == 0x40000000) && (PartitionExecAddr == 0))))) {

            Status = PartitionMove(ImageStartAddress, HeaderPtr);
            if (Status != XST_SUCCESS) {
                fsbl_printf(DEBUG_GENERAL,"PARTITION_MOVE_FAIL\r\n");
                OutputStatus(PARTITION_MOVE_FAIL);
```

```
                FsblFallback();
            }
        }
```

25.3.1 Execute In Place (XIP)

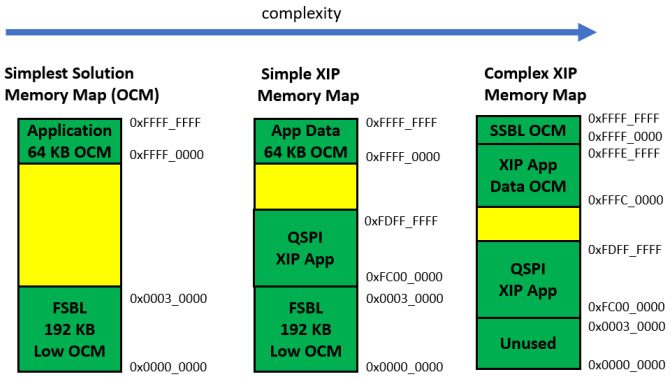
Due to the limited amount of internal memory, both in On Chip Memory (OCM) and Block RAM (BRAM) in the PL, users might require more memory for the application and XIP could be a solution. XIP with Zynq 7000 is less common and can be a bit challenging when reviewing the documentation and FSBL image processing. An application is built for XIP by linking the text sections for the linear QSPI flash memory and the data sections for OCM. As illustrated below in the capture from the TRM (UG585), FSBL with both RSA and XIP is not supported by the BootROM such that the prototype system described on this page only supports user applications (not FSBL) built in XIP mode.



A key to making an XIP image run is to make sure that the application image ends up in flash at the same address where the image is linked. This is done by using the *offset* property in the BIF file when building the boot image with BootGen and making sure the partition containing the text is before any data partition in the address space and boot image as described in Boot Images paragraph of this page.

25.3.2 OCM

FSBL by default is linked for all of OCM, but can be linked such that it occupies the lowest 192 KB of OCM leaving 64 KB of OCM for applications. A small application that fits into 64 KB of OCM is the easiest solution. The next easier solution is an XIP application with data which fits into the 64 KB of OCM. And the most complex solution is an XIP application with data that is too large to fit into the 64 KB of OCM. The prototype system described assumes the most complex solution with a Second Stage Boot Loader (SSBL). The following diagram illustrates the memory map for the described systems.



25.3.3 Block RAM (BRAM)

The PL contains BRAM that can be used together with AXI infrastructure to allow the CPU to execute from the BRAM or use it for data. FSBL cannot normally load applications into BRAM as the PL is not fully operational until late in FSBL operation at the handoff to an application. The PL bitstream is loaded first by FSBL such that the PL BRAM can be up and running prior to other image processing. The following code snippet illustrates adding conditional compilation for when a BRAM is in the design to call *ps7_post_config()* in the *FsblHookAfterBitStreamDload()* function of the *fsbl_hooks.c* source file to cause the PL BRAM to be ready to load applications.

```
u32 FsblHookAfterBitstreamDload(void)
{
    u32 Status;

    Status = XST_SUCCESS;

#ifdef XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR
    /* When a BRAM is in the PL, enable it as soon as the PL is loaded
     * so that an app can be loaded into it by FSBL
     */
    ps7_post_config();
#endif
```

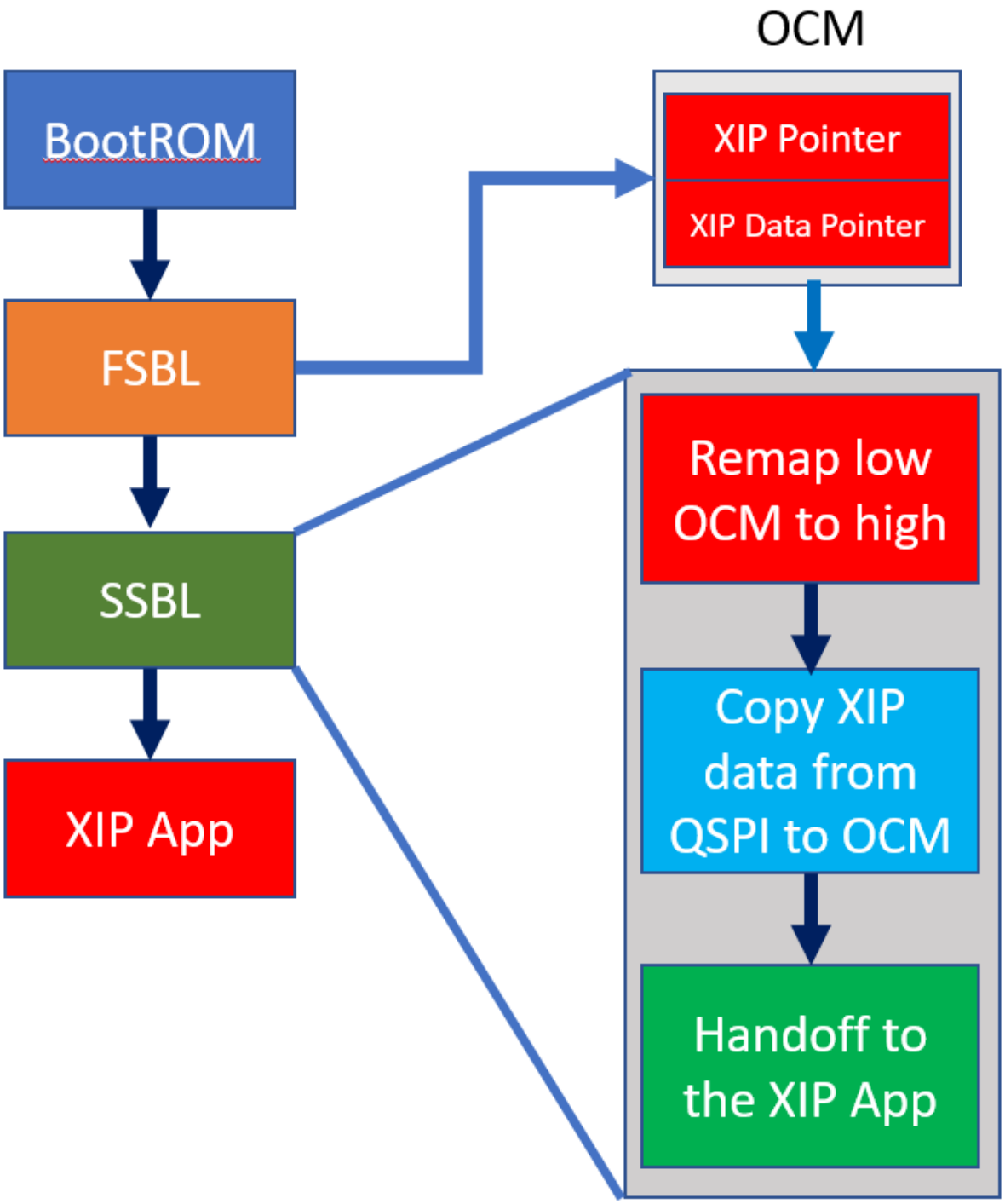
25.3.4 Secondary Stage Boot Loader (SSBL)

For XIP applications that require more than 64 KB of OCM, a more complex solution is required with a Secondary Stage Boot Loader (SSBL) which FSBL starts after it is complete. The SSBL is linked and runs from the high 64 KB of OCM allowing the XIP application data to consume the lower 192 KB of OCM. The SSBL copies the XIP application data section from QSPI to OCM and then starts the XIP application running from QSPI memory. This solution also assumes the system never needs FSBL again as it is overwritten by the XIP application data.

The XIP application data and the XIP application execution address in QSPI memory are needed by the SSBL to load and start the XIP application. FSBL is altered to use OCM addresses for shared memory

communicating the previously described addresses between itself and SSBL. Using the shared memory helps prevent duplication of FSBL functionality in SSBL as security duplication would be complex.

SSBL remaps the low 192 KB of OCM, where FSBL was executing at address zero, up high at 0xFFFC_0000 to match the linked XIP application data section as explained in the Boot Images paragraph. SSBL must also prepare the CPU, such as disabling the caches and MMU, to start the XIP application. FSBL contains this processing in the handoff code which can be re-used by SSBL. The following illustration shows a high level description of the boot flow from power up to the execution of the XIP application.


OCM
BootROM
XIP Pointer
XIP Data Pointer
FSBL
SSBL
Remap low OCM to high
Copy XIP data from QSPI to OCM
XIP App
Handoff to the XIP App

25.3.4.1 SSBL Details

The following code snippets illustrate the major functionality required for an SSBL application. These changes must be combined with FSBL changes to create the shared memory variables which contain the XIP application data address and execution address.

SSBL maps the lower 192 KB of OCM from the low address of zero up to the high address of 0xFFFC_0000. The remapping must be done prior to loading the XIP application data which is linked for high OCM. The following code snippet illustrates mapping the memory high.

```
    /* Map OCM from low to high as the primary app data is linked for high OCM so
that XIP QSPI works with Bootgen partitions
     */
    Xil_Out32(0xF8000008, 0xDF0D);          // Unlock the SLCR
    u32 ocm_cfg = Xil_In32(0xF8000910);     // Always read the register to keep bit 4
unchanged, otherwise mapping back low won't work
    Xil_Out32(0xF8000910, (ocm_cfg | 0xF)); // Map all OCM high
    Xil_Out32(0xF8000004, 0x767B);          // Re-lock the SLCR
```

SSBL copies the XIP application data from the QSPI to the high OCM. The following code snippet illustrates SSBL using the shared data (FLASH_DATA_PTR_ADDR) which FSBL created to get the address of the XIP application data and copy it to the high OCM as 0xFFFC_0000.

```
    /* Copy the data for the program from QSPI to high OCM as the program is running
     * XIP from QSPI but needs RAM for data
     */
    src = *(u32 *)FLASH_DATA_PTR_ADDR;
    dest = (u32 *)0xFFFC0000;
    for (i = 0; i < LOW_OCM_SIZE / 4; i++)
        *dest++ = *src++;
```

Before handing off execution to the XIP application it must make the CPU ready just as FSBL does in the handoff code. The SsblHandoffExit() function is copied from the FsblHandoffExit() in the fsbl_handoff.S source file into the project and then called at the end of SSBL. The following code snippet illustrates the code of the handoff to get the CPU ready for a new application to be started.

```
SsblHandoffExit:
        mov  lr, r0 /* move the destination address into link register */

        mcr  15,0,r0,cr7,cr5,0      /* Invalidate Instruction cache */
        mcr  15,0,r0,cr7,cr5,6      /* Invalidate branch predictor array */

        dsb
        isb                 /* make sure it completes */

    ldr r4, =0
        mcr  15,0,r4,cr1,cr0,0      /* disable the ICache and MMU */
```

```
        isb                 /* make sure it completes */

        bx      lr  /* force the switch, destination should have been in r0 */
```

The following code snippet illustrates calling the handoff function using the shared data (HANDOFF_PTR_ADDR) which FSBL created when the XIP application image was processed.

```
    SsblHandoffExit(*(u32 *)HANDOFF_PTR_ADDR);
```

25.3.4.2 FSBL Changes For SSBL

In the LoadBootImage() function of the *image_mover.c* source file, the SSBL required data should be added in the partition handling loop to allow the XIP application execution address (handoff pointer) and the data to be found. The following code snippet illustrates processing to find the XIP application and write the addresses to the lowest addresses of the high 64 KB of OCM where SSBL will use them.

```
#define FLASH_DATA_PTR_ADDR   0xFFFF0000
#define HANDOFF_PTR_ADDR      0xFFFF0004
#define QSPI_FLASH_BASEADDR   XPAR_PS7_QSPI_LINEAR_0_S_AXI_BASEADDR
#define QSPI_FLASH_HIGHADDR   XPAR_PS7_QSPI_LINEAR_0_S_AXI_HIGHADDR

        /* SSBL needs the address of the data to be copied from flash to OCM
         * and it needs the handoff address for the XIP program, so put them up
         * in high OCM just before the SSBL image
         */
        if ((PartitionExecAddr >= QSPI_FLASH_BASEADDR) &&
                (PartitionExecAddr <= QSPI_FLASH_HIGHADDR)) {
            fsbl_printf(DEBUG_GENERAL,"SSBL Handoff Address: 0x%08X\r\n",
PartitionStartAddr);
            Xil_Out32(HANDOFF_PTR_ADDR, PartitionStartAddr);
        }

        /* For QSPI XIP mode, there are two partitions in the boot image, the
         * first being the text region of the image that will run from QSPI,
         * then the second being the data that must be loaded into some RAM
         * such as OCM. The SSBL needs to know where the data is in QSPI so
         * that it can transfer it after FSBL is no longer running from OCM.
         */
        if ((PSPartitionFlag == 1) && ((PartitionLoadAddr >= 0xFFFC0000) &&
(PartitionLoadAddr < 0xFFFF0000))) {
            Xil_Out32(FLASH_DATA_PTR_ADDR, PartitionStartAddr);
        }
```

25.3.5 Boot Images

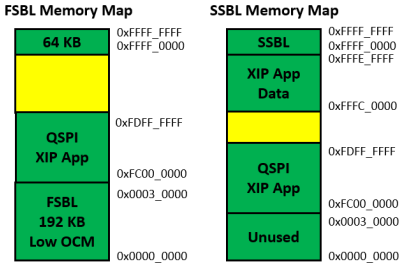
The BootGen tool generates a boot image consisting of a set of application images and a bitstream which is written to a boot media such as QSPI flash memory. A boot image contains partitions for the applications and the bitstream which can be individually controlled with respect to the location in the boot image. DDR

based systems allow application text and data to be linked and located in the same large continuous memory which creates a simple single partition. DDR-less systems using XIP require application text sections to be linked and located in the boot media while application data sections are linked and located for RAM and therefore create a more complex boot image due to discontinuous text and data memories.

The tools do not support Position Independent Code. As a result, an XIP application must be linked for the address where it will be executing, such as QSPI in linear address mode, with the data being linked for RAM. The memory sections of the application, text and data, are multiple memories such that they are discontinuous and BootGen generates a multi-partition image for the application. The partitions of the multi-partition image are in the order of address, so the text partition must be first (using a lower address) to allow a predictable address matched to the linked address.

BootGen allows images to be located at specific offsets in the boot image but it does not allow the individual partitions of a multi-partition image to be individually located. This is not an issue when the text section of the application is at a lower address than the data section, but is a challenge with data memory at a lower address than the text, such as with OCM mapped low or BRAM.

QSPI flash in linear mode starts at address 0xFC00_0000 and OCM can be mapped high in the address space starting at 0xFFFC_0000, so this works nicely for XIP mode. BRAM does not work easily for the data section because it is mapped at 0x4XXX_XXXX or 0x8XXX_XXXX in the address map. The following illustration shows the memory map when FSBL is executing, followed by the memory map when SSBL is executing to illustrate OCM moving from low addresses to high addresses.



25.3.6 RSA Authentication Support

RSA processing is done in both the BootROM and FSBL. The FSBL assumes DDR is present such that the image is copied into DDR prior to performing the RSA signature verification. A DDR-less FSBL requires changes to support RSA, so that it performs the RSA signature verification from QSPI. Security of the system should also be considered by users with the DDR-less system as the CPU spends more time accessing the external memory while performing the RSA algorithm. Boot time might be another consideration for fast booting systems and boot time was not a factor in the proposed solution.

FSBL changes to support RSA authentication from the QSPI include using the address of the image in QSPI for the authentication, and removal of the DDR as a temporary buffer for the PL bitstream. These are illustrated in the following code snippet of the image_mover.c source file and the LoadBootImage() function. Note the addition of conditional compilation for when DDR is in the system.

```
#ifdef XPAR_PS7_DDR_0_S_AXI_BASEADDR
        /*
         * Move partitions from boot device
         */
        Status = PartitionMove(ImageStartAddress, HeaderPtr);
        if (Status != XST_SUCCESS) {
```

```
            fsbl_printf(DEBUG_GENERAL,"PARTITION_MOVE_FAIL\r\n");
            OutputStatus(PARTITION_MOVE_FAIL);
            FsblFallback();
        }
#else
        PartitionStartAddr = ImageStartAddress;
        PartitionStartAddr += HeaderPtr->PartitionStart << WORD_LENGTH_SHIFT;
        PartitionStartAddr += FlashReadBaseAddress;
        ...
#endif
        if ((SignedPartitionFlag) || (PartitionChecksumFlag)) {

#ifdef XPAR_PS7_DDR_0_S_AXI_BASEADDR

            /* For DDR systems do processing of the bitstream in DDR
             */
            if(PLPartitionFlag) {
                /*
                 * PL partition loaded in to DDR temporary address
                 * for authentication and checksum verification
                 */
                PartitionStartAddr = DDR_TEMP_START_ADDR;
            } else {
                PartitionStartAddr = PartitionLoadAddr;
            }
#endif
```

The following snippet of code illustrates the change to the GetNAuthImageHeader() function in *image_mover.c* which causes the DDR-less system to allocate a temporary buffer in OCM rather than DDR for the authentication header.

```
u32 GetNAuthImageHeader(u32 ImageBaseAddress)
{
    u32 Status;
    u32 Offset;
#ifdef XPAR_PS7_DDR_0_S_AXI_BASEADDR
    u8 *HdrTmpPtr = (u8 *) DDR_TEMP_START_ADDR;
#else
    /* Create a non-stack temporary area for when there's no DDR
     */
    static u8 temp[TOTAL_HEADER_SIZE+RSA_SIGNATURE_SIZE];
    u8 *HdrTmpPtr = (u8 *)&temp;
#endif
```

25.3.6.1 Load Failures

For any security failures, such as a bad RSA signature, the FSBL normally causes a boot failure and makes the BootROM do multi-boot searching for the next valid image in the boot media. This might not make sense for all systems as it might be valid to boot into a secure application that recognizes the failure and then

updates the failing images. The following code snippet illustrates FSBL failing to authenticate an image and doing fallback to the BootROM.

```
                    OutputStatus(AUTHENTICATION_FAIL);
                    FsblFallback();
```

25.3.7 Boot Image Generation

Generating the boot image is critical for success. The following BIF file is an example for the prototype system which includes RSA keys, the FSBL, a PL bitstream, the SSBL application, a test application (hello world), and a second copy of the FSBL in BRAM for CPU warm reset. Images are placed at specific places in flash (offsets) to allow testing as the QSPI can only be erased in 128K sectors.

```
//arch = zynq; split = false; format = BIN
the_ROM_image:
{
    [pskfile]C:\psk.pem
    [sskfile]C:\ssk.pem
    [bootloader, authentication = rsa]fsbl-custom\Debug\fsbl-custom.elf
    [authentication = rsa, offset = 0x100000]hello-world/_ide/bitstream/
design_1_wrapper.bit
    [authentication = rsa, offset = 0x300000]ssbl\Debug\ssbl.elf
    [authentication = rsa, offset = 0x400000]hello-world/Debug/hello-world.elf
    [authentication = rsa, offset = 0x500000, load = 0x40000000]fsbl-
custom\Debug\fsbl-custom.elf
}
```

25.3.8 System Updates

For QSPI flash memory, DDR-less systems with XIP require that the system update software execute from another memory rather than QSPI. The QSPI memory is addressed in linear address mode for XIP which is a read only mode. The update application uses the QSPI driver and flash example to write to QSPI using I/O mode while running from OCM.

25.3.9 Vitis Debug

When programming flash memory or doing debug using an FSBL rather than using *ps7_init.tcl*, the DDR-less FSBL must be used for all Vitis operations that require FSBL. An FSBL for DDR will fail such that normal operation is not possible, causing failures such as a failure when trying to program flash memory.

25.4 RSA Authentication

25.4.1 RSA Without eFuses

It can be useful to test RSA security without the need to program eFuses on the board. This is not a full security test, but it is a useful test prior to programming eFuses and allows the FSBL RSA processing to be tested. A minor change is required to the FSBL to support this feature as illustrated in the code snippet below. The code snippet is inserted at the point in the *image_mover.c* source file and the LoadBootImage() function where the eFuse Status Register is being read. It uses conditional compilation to turn on this feature with RSA_WITHOUT_EFUSES.

```
#ifndef RSA_WITHOUT_EFUSES
  EfuseStatusRegValue = Xil_In32(EFUSE_STATUS_REG);
#else
  EfuseStatusRegValue = 0;

  extern u32 __rsa_ac_start;

  /* If RSA authentication certificate (AC) is in the boot image then setup the
   * RSA processing without eFUSEs otherwise don't force RSA to be used when the
   * user is not asking for it. This is only to allow early testing before eFUSEs
   * are programmed and is not a fully secure system.
   */
  if (PartitionHeader[0].ACOffset != 0) {
    /* The BootROM normally copies the PPK AC into OCM but only does it when the RSA
enable
     * efuse is programmed. There is a special linker section that already exists but
is
     * not normally used so copy the PPK AC into it rather than just at the end of
the
     * FSBL text section which seems to cause problems of overwriting the data
section.
     */
    fsbl_printf(DEBUG_GENERAL, "*** RSA Without eFUSE, Not a secure system! ***\n\r")
;

    /* Force RSA regardless of the eFUSE so no eFUSE programming is required for
early testing
     */
    EfuseStatusRegValue = EFUSE_STATUS_RSA_ENABLE_MASK;
    Status = MoveImage(ImageStartAddress + (PartitionHeader[0].ACOffset * 4),
    (u32)&__rsa_ac_start, TOTAL_HEADER_SIZE + RSA_SIGNATURE_SIZE);

    if (Status != XST_SUCCESS) {
      fsbl_printf(DEBUG_GENERAL,"RSA Without eFUSE, AC PPK Image failed\r\n");
      return XST_FAILURE;
    }
```

```
    /* Keep using the FSBL length variable but use the address of the special section
for
     * the fake FSBL length so other processing finds the AC PPK
     */
    FsblLength = (u32)&__rsa_ac_start;
  }
#endif
```

25.5 CPU Warm Reset

Resets with the Zynq 7000 architecture include POR and SRST which include rebooting through the BootROM. There are use cases for CPU warm resets without going through the BootROM such as reinitializing the software to a known state while keeping the PL in an operational state. A warm reset causes the CPU to run the FSBL again without going through the BootROM so that images are reloaded and restarted. A CPU warm reset of the type described in the following paragraphs is most suited for simpler systems which do not include AXI masters in the PL, which might be accessing PS resources.

25.5.1 AWDT

The Cortex A9 CPUs of the Zynq 7000 each include a private watchdog timer referred to as the AWDT in the TRM. The AWDT has the ability to reset the CPU. The AWDT must be configured to cause a CPU reset rather than a system reset.

25.5.2 Reset Effects

The following illustration from the TRM at Zynq 7000 SoC TRM (UG585) shows the details of the resets noting that the AWDT with a CPU only reset cannot be detected in a register after the warm reset.

Table 26-1: **Reset Effects**

Reset Name	Source	Portion of System that is Reset	RAMs Cleared	slcr.REBOOT_STATUS Bits set = 1
Power-On Reset (PS_POR_B)	Device pin	Entire chip, including debug (All) The PL must be re-programmed.	All	[POR]
Security Lock Down (requires a power-on reset to recover)	DevC		All	N/A
External System Reset (PS_SRST_B)	Device pin	All except debug and persistent registers. The PL must be re-programmed.	All	[SRST_RST]
System Software	SLCR		All	[SLC_RST]
System Debug Reset	JTAG		All	[DBG_RST]
System Watchdog Timer	SWDT		All	[SWDT_RST]
CPU0 and CPU1 Watchdog Timers (when slcr.RS_AWDT_CTRL{1,0} = 0)	AWDT		All	[AWDT{1,0}_RST]
CPU0 and CPU1 Watchdog Timers (when slcr.RS_AWDT_CTRL{1,0} = 1)	AWDT	CPU (s) only.	None	N/A
Debug Reset	JTAG	Debug logic.	None	N/A
Peripherals	SLCR	Selected peripherals or CPUs.	None	N/A

25.5.3 Watchdog Timer Reset Control

The following illustration from the TRM shows the bits of the control register which determines which kind of reset the AWDT causes.

Register (slcr) RS_AWDT_CTRL

Name	RS_AWDT_CTRL
Relative Address	0x0000024C
Absolute Address	0xF800024C
Width	32 bits
Access Type	rw
Reset Value	0x00000000
Description	Watchdog Timer Reset Control

Register RS_AWDT_CTRL Details

Field Name	Bits	Type	Reset Value	Description
reserved	31:2	rw	0x0	Reserved. Writes are ignored, read data is zero.

Field Name	Bits	Type	Reset Value	Description
CTRL1	1	rw	0x0	Select the target for the APU watchdog timer 1 reset signal. Route the WDT reset to: 0: the same system level as PS_SRST_B 1: the CPU associated with the watchdog timer
CTRL0	0	rw	0x0	Select the target for the APU watchdog timer 0 reset signal. Route the WDT reset to: 0: the same system level as PS_SRST_B 1: the CPU associated with the watchdog timer

The following code snippet illustrates altering the AWDT to cause a CPU reset rather than a system reset.

```
    /* Unlock the SLCR, then route the reset to the CPU only, not a system reset */

    Xil_Out32(0xF8000008, 0xDF0D);
    Xil_Out32(0xF800024C, 0x1);
```

25.5.4 AWDT Reset Detection

The Multiboot Register is persistent across CPU and SRST resets, and provides a number of bits for user specific purposes. Assuming an application sets a bit in the Multiboot Register, the FSBL can detect the CPU reset and perform application specific processing, such as not reloading the PL.

Register (devcfg) XDCFG_MULTIBOOT_ADDR_OFFSET

Name	XDCFG_MULTIBOOT_ADDR_OFFSET
Software Name	MULTIBOOT_ADDR
Relative Address	0x0000002C
Absolute Address	0xF800702C
Width	13 bits
Access Type	rw
Reset Value	0x00000000
Description	Multi-Boot Address Pointer.

Register XDCFG_MULTIBOOT_ADDR_OFFSET Details

MULTI Boot Addr Pointer Register: This register defines multi-boot address pointer. This register is power on reset only used to remember multi-boot address pointer set by previous boot.

Field Name	Bits	Type	Reset Value	Description
MULTIBOOT_ADDR	12:0	rw	0x0	Multi-Boot offset address

The following code snippet illustrates setting the MSB of the register which is done before the AWDT causes a CPU reset.

```
Xil_Out32(0xF800702C, Xil_In32(0xF800702C) | 0x80000000);
```

25.5.5 AWDT Driver

A bare metal driver is provided for the AWDT in Vitis and is named xscuwdt. An example application in a source file named *xscuwdt_polled_example.c* is provided to start the AWDT and cause a CPU reset. The application must be altered to configure the CPU reset, rather than a system reset, and allow the AWDT to timeout to cause the CPU reset.

25.5.6 FSBL

The FSBL must reside in OCM for a CPU reset to be performed. DDR-less systems might require the FSBL to be restored from QSPI to OCM to facilitate the CPU reset. In a non-secure system, the FSBL can be copied by SSBL from QSPI into low OCM. In a secure system, a copy of the FSBL can be loaded by the FSBL into BRAM so that an authenticated FSBL can be reloaded into OCM by SSBL.

25.5.7 Alternative Designs

Another potentially useful method to create a CPU reset is to use the A9_RST0 bit of the slcr.A9_CPU_RST_CTRL Register. More detailed investigation shows that this is not a method that can be used by a CPU to reset itself, but is designed to be used by another CPU such as the other Cortex A9.